MINICURSOS

WSCAD 2022

Florianópolis/SC

Minicursos do XXIII Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD)

Realização

Coordenação

Patrocinadores Diamante

Fomento

Patrocinadores Ouro

Patrocinadores Prata

XXIII SIMPÓSIO EM SISTEMAS COMPUTACIONAIS DE ALTO DESEMPENHO

DE 19 A 21 DE OUTUBRO DE 2022

FLORIANÓPOLIS – SC

MINICURSOS DO XXIII SIMPÓSIO EM SISTEMAS COMPUTACIONAIS DE ALTO DESEMPENHO

Organizadores

Carla Osthoff

Daniel Cordeiro

Porto Alegre

Sociedade Brasileira de Computação – SBC

2022

Copyright ©2022 da Sociedade Brasileira de Computação
Todos os direitos reservados

Política de Direitos Autorais da SBC
Os autores dos livros e capítulos publicados na SBC OpenLib retêm os direitos autorais de suas obras e autorizam a SBC a publicá-las de acordo com os termos da licença Creative Commons Attribution-NonComercial 4.0 International Public License (CC BY-NC 4.0). Dessa forma, fica permitido aos autores ou a terceiros a reprodução ou distribuição, em parte ou no todo, de material extraído dessas obras, de forma verbatim, adaptada ou remixada, bem como a criação ou produção a partir do conteúdo dessas obras, para fins não comerciais, desde que sejam atribuídos os devidos créditos às criações originais. Cópias das obras não devem ser utilizadas de nenhum modo que implique o endosso da SBC.

Dados Internacionais de Catalogação na Publicação (CIP)

S612 Simpósio em Sistemas Computacionais de Alto Desempenho (23. : 19 – 21 out. 2022 : Florianópolis)
Minicursos do XXIII Simpósio em Sistemas Computacionais de Alto Desempenho [recurso eletrônico] / organização: Carla Osthoff ; Daniel Cordeiro. Dados eletrônicos. – Porto Alegre: Sociedade Brasileira de Computação, 2022.
167 p. : il. : PDF ; 9.2MB

Modo de acesso: World Wide Web.
Inclui bibliografia
ISBN 978-85-7669-515-8 (e-book)

1. Computação – Brasil – Simpósio. 2. Sistemas computacionais. 3. Computação de alto desempenho. I. Osthoff, Carla. II. Cordeiro, Daniel. III. Sociedade Brasileira de Computação. IV. Título.

CDU 004(063)

Ficha catalográfica elaborada por Jéssica Paola Macedo Müller – CRB-10/2662
Biblioteca Digital da SBC – SBC OpenLib

Índices para catálogo sistemático:

1. Ciência e tecnologia dos computadores : Informática – Publicação de conferências, congressos e simpósios etc. ... 004(063)

Prefácio

Neste livro estão compilados os seis minicursos apresentados durante o XXIII Simpósio em Sistemas Computacionais de Alto Desempenho, realizado entre os dias 19 e 21 de outubro de 2022 em Florianópolis, SC. Em todos os minicursos destaca-se o viés prático que incentiva os participantes e os leitores a utilizarem alto desempenho de maneira efetiva. O capítulo 1 "**OneAPI: Uma Abordagem para a Computação Heterogênea Centrada no Desenvolvedor**", apresenta o padrão OneAPI, que é um padrão aberto para a aceleração de computação que adota uma abordagem single-source, na qual todo o código da aplicação pode ser especificado uniformemente usando a linguagem C++, independente de executar no host ou em aceleradores. O capítulo 2 "**Inteligência Artificial e Função como Serviço: Provisionando Aplicações com o AWS Lambda**", apresenta o AWS Lambda, que é uma das principais ferramentas do chamado serviço de computação sem servidor. Dentro desse contexto, são introduzidos os conceitos de Função como Serviço (FaaS), englobando o uso de AWS Lambda para provisionar aplicações de AI. O capítulo 3 "**Coisas para Fazer Antes de Paralelizar**", apresenta práticas que devem ser feitas para melhorar o desempenho de um código antes de se pensar em paralelização e revisa conceitos essenciais para a avaliação e comparação de programas, incluindo exemplos e demonstrações ao vivo de como aplicá-los. O capítulo 4 "**Fundamentos de Computação Acelerada com CUDA C/C++**", ensina as ferramentas e técnicas fundamentais para acelerar aplicações escritas em linguagens C/C++ para execução em arquiteturas massivamente paralelas com CUDA. O capítulo 5 "**From Sequence Assembly to Ancestry Testing: HPC Challenges for Bioinformatics**" fornece uma visão introdutória do processo de obtenção até a análise de dados biológicos, mostrando os principais algoritmos e banco de dados em cada etapa e uma discussão sobre as formas de paralelização destes algoritmos e os principais desafios que ainda carecem de uma solução. O capítulo 6, "**Ferramentas para Configuração e Gerenciamento de Cluster de Alto Desempenho em Nuvem Pública**", apresenta as melhores práticas adotadas pelo NACAD-COPPE/UFRJ para a implantação de um *cluster* de Alto Desempenho usando uma Nuvem Pública focando no uso de ferramentas para implantar e gerenciar clusters de Computação de Alto Desempenho (HPC) na Nuvem AWS.

Com este livro esperamos contemplar os usuários e pesquisadores da Área de Alto Desempenho que não puderam estar presentes no evento. Desejamos uma boa leitura.

Carla Osthoff (LNCC) e Daniel Cordeiro (EACH-USP)
Coordenadores dos minicursos do WSCAD 2022

Sumário

Minicurso I

OneAPI: uma Abordagem para a Computação Heterogênea Centrada no Desenvolvedor . . 1
Ricardo Menotti (UFSCAR) e Tiago da Conceição Oliveira (SENAI CIMATEC)

Minicurso II

Inteligência Artificial e Função como Serviço: Provisionando Aplicações com o AWS Lambda 36
Sayonara S. Araújo (UFC), Francisco R. P. da Ponte (UFC), Victória T. Oliveira (UFC), Wendley S. da Silva (UFC), Dario Vieira (EFREI), Miguel F. de Castro (UFC) e Emanuel B. Rodrigues (UFC)

Minicurso III

Coisas para Fazer Antes de Paralelizar 65
Alfredo Goldman (USP), Vitor Tessari Terra (USP) e Sarita Mazzini Bruschi (USP)

Minicurso IV

Fundamentos de Computação Acelerada com CUDA C/C++ 88
Arthur Francisco Lorenzon (UFRGS)

Minicurso V

From Sequence Assembly to Ancestry Testing: HPC Challenges for Bioinformatics 106
Mariana Carmin (UFPR), Cláudio Torres Júnior (UFPR), André Ricardo Abed Grégio (UFPR) e Marco Antonio Zanata Alves (UFPR)

Minicurso VI

Ferramentas para Configuração e Gerenciamento de Cluster de Alto Desempenho em Nuvem Pública 139
Albino A. Aveleda (UFRJ) e Alvaro L. G. A. Coutinho (UFRJ)

Capítulo 1

OneAPI: uma Abordagem para a Computação Heterogênea Centrada no Desenvolvedor

Ricardo Menotti (UFSCAR) e Tiago da Conceição Oliveira (SENAI CIMATEC)

Abstract

In heterogeneous computing, applications are designed so that their execution can be shared between different architectures. In general, part of the program is implemented to run on a host, and part is split into kernels that are submitted to one or more accelerators. OneAPI is an open standard for computing acceleration that takes a single-source approach, in which all application code can be uniformly specified using the C++ language, regardless of whether it runs on the host or on accelerators. In this short course, we will see how applications can be parallelized, implemented, and validated using only one multicore CPU as host, including the Bitonic Sort parallel sorting algorithm. Later, the same codes will be gradually changed to use different accelerators (integrated GPU, discrete GPU, and FPGA) in the Intel® DevCloud environment. To this end, the different execution models of each target architecture, the general form of parallel programming in this approach, and the programming nuances that favor each execution model will be presented.

Resumo

Na computação heterogênea as aplicações são projetadas para que sua execução seja compartilhada entre diferentes arquiteturas. Em geral, parte dela é implementada para executar em um host e parte é dividida em kernels para serem submetidos para um ou mais aceleradores. OneAPI é um padrão aberto para a aceleração de computação que adota uma abordagem single-source, na qual todo o código da aplicação pode ser especificado uniformemente usando a linguagem C++, independente de executar no host ou em aceleradores. Neste minicurso, veremos como aplicações podem ser paralelizadas, implementadas e validadas usando apenas uma CPU multicore como host, incluindo o algoritmo de ordenação paralela Bitonic Sort. Posteriormente, os mesmos códigos serão gradativamente alterados para usarem aceleradores diversos (GPU integrada, GPU

discreta e FPGA) no ambiente Intel® DevCloud. Para tal, serão apresentados os diferentes modelos de execução de cada arquitetura alvo, a forma geral de programação paralela nesta abordagem e as nuances na programação que favorecem cada modelo de execução.

1.1. Introdução

A computação heterogênea – termo que surgiu há alguns anos e que não será esquecido tão logo – chegou para ficar. Em suma, trata-se de sistemas paralelos nos quais um ou mais nós computacionais passuem diferentes formas de executar instruções. Mohamed Zahran (2017) destaca diferentes graus possíveis de heterogeniedade:

1. Núcleos idênticos podem apresentar desempenho distinto se contam com tensão dinâmica e escala de frequência;
2. Núcleos com características arquiteturais distintas que parecem executar instruções em sequência, mesmo se por trás disso houver algum tipo de paralelismo entre as instruções;
3. Nós de computação com diferentes modelos de execução, tais como GPUs, que processam múltiplos dados com instrução única (ou *thread*);
4. Aceleradores programáveis (FPGAs) que podem ser usados como hardware especializado, suprimindo completamente o modelo de execução por instruções.

Ao nível do hardware, podemos citar alguns desafios evidentes: **(i) hierarquia de memória** - os sistemas de memória se tornaram verdadeiros gargalos desde muito tempo e a heterogeniedade só agrava o problema, uma vez que diferentes núcleos certamente têm necessidade de acesso variadas (largura de banda, tempo de resposta, gasto energético, etc.). **(ii) interconexão** - como conectar os diferentes módulos e a hierarquia de memória de forma eficiente? **(iii) balanceamento de carga** - como distribuir a carga de trabalho de forma a obter o melhor desempenho com o menor consumo energético?

Ao nível do software, programar sistemas heterogêneos é extremamente desafiador a medida que novos modelos de programação revelam mais detalhes do hardware. O senso comum diz que muitos aspectos do hardware precisavam ser ocultados do programador para aumentar sua produtividade. Linguagens de alto nível costumam oferecer rotinas que são apenas chamadas para métodos otimizados escritos em C/C++. Porém, em sistemas heterogêneos, mesmo os programadores destas linguagens precisam tomar algumas decisões difíceis: Como decompor o aplicativo em *threads* adequados para o hardware em questão? Quais partes do programa não requerem alto desempenho e podem ser executados no modo de baixo consumo de energia? Os desafios são enormes, principalmente se considerarmos – além do desempenho – a escalabilidade e portabilidade do código (Zahran, 2017).

Herb Sutter (2011), já argumentava em seu artigo ***Welcome to the Jungle: Or, A Heterogeneous Supercomputer in Every Pocket*** que:

> *"In the twilight of Moore's Law, the transitions to multicore processors, GPU computing, and HaaS cloud computing are not separate trends, but aspects of a single trend – mainstream computers from desktops to 'smartphones' are being permanently transformed into heterogeneous supercomputer clusters. Henceforth, a single compute-intensive application will need to harness different kinds of cores, in immense numbers, to get its job done."*

O autor dizia que a programação tradicional/sequencial – a qual ele chama de "almoço grátis" – havia chegado ao fim e dava as boas vindas à "selva do hardware". Segundo ele, este processo se iniciou por volta de 2005, com o advento dos processadores *multicore*, e concluiu-se em torno de 2012, quando a Nintendo – fabricante do único console que ainda era *single core* (Wii) – anunciou que iria descontinuá-lo em detrimento de outro que já seria *multicore* (Wii U). Quando os sistemas *multicore* surgiram eles eram homogêneos, ou seja, possuiam núcleos idênticos. Na computação heterogênea os núcleos são diferentes.

Analisando a evolução desde aquela época até os dias de hoje, percebe-se que o cenário não é tão desolador e que ainda é possível programar muitas coisas sem se preocupar com a arquitetura do hardware. Com a ascenção das aplicações *web* e *mobile*, bibliotecas e *frameworks* usados para este fim podem abstrair do desenvolvedor detalhes do paralelismo usufruindo dele em níveis mais baixos. Também se programam aplicações e funcionalidades que não executam em paralelo em si, mas concorrem com outras partes ou aplicações, resultando em um paralelismo grosso, gerenciado em mais alto nível.

Porém, quando se trata da computação de alto desempenho, faz-se necessário adotar modelos de programação no qual o paralelismo deve ser explicitado pelo desenvolvedor, o que os torna mais dependentes da arquitetura alvo. Padrões como o OpenCL e SYCL foram criados para a computação heterogênea visando cobrir uma vasta gama de arquiteturas (CPUs, GPUs, DSPs e FPGAs). Embora seja possível desenvolver aplicações para todas elas usando um desses padrões, uma única implementação não costuma oferecer o melhor desempenho em todas elas e ajustes significativos precisam ser feitos para se obter melhores resultados quando se migra de uma arquitetura para outra.

No caso dos FPGAs, descrições usando padrões como o OpenCL e SYCL são usadas para se fazer síntese de alto nível (HLS[*]) e gerar automaticamente o hardware especializado. O processo pode resultar em arquiteturas subótimas, pois os padrões não permitem acessar diretamente o hardware como seria possível usando linguagens de descrição de hardware (HDL[†]). Apesar disso, a complexidade dos sistemas atuais e a abundância de recursos dos FPGAs mais recentes têm tornado seu uso atrativo (Firmansyah & Yamaguchi, 2019).

Desde que surgiram, os FPGAs esteviveram relativamente restritos a poucas aplicações da uma pequena parte da comunidade científica. Isso se deu principalmente pela dificuldade de se projetar sistemas especializados usando linguagens de descrição de hardware como VHDL e Verilog. Mais recentemente, os esforços por aperfeiçoar técnicas de síntese de alto nível e desenvolver ferramentas deste tipo, têm permitido um uso mais

[*] Do inglês: *High-Level Synthesis*

[†] Do inglês: *Hardware Description Languages* (e.g. Verilog, VHDL)

amplo dos FPGAs, pois elas já realizam a geração automática de aceleradores relativamente eficientes a partir de descrições em C/C++ (Stock, 2019), OpenMP (Ceissler et al., 2018), OpenCL (Gautier et al., 2016), SYCL (Keryell & Yu, 2018), entre outras.

Neste minicurso será abordado o uso da oneAPI – uma implementação do padrão SYCL com extensões – para a progração de aceleradores em geral (Intel Corp., 2020b). A principal vantagem desta abordagem deriva de se tratar de um padrão aberto, usado por outros fabricantes, e que pode ser usado também para programar outras platadormas (GPUs, multicores, etc.). Ao final do minicurso os participantes irão: (**i**) Aprender o básico sobre a programação SYCL (DPC++); (**ii**) Entender o ciclo de desenvolvimento para CPU, GPUs e FPGAs usando oneAPI; e (**iii**) Compreender os métodos comuns de otimização visando estes dispositivos.

O restante deste capítulo está organizado da seguinte forma. Na Seção 1.2 são descritas as arquiteturas abordades neste minicurso, plataformas disponíveis e métodos de desenvolvimento empregados atualmente. Na Seção 1.3 são descritos os conceitos de C++ importantes para a compreensão do padrão SYCL, bem como as principais classes usadas na implementação e comunicação com os *kernels* e os respectivos escopos para sua programação. Na Seção 1.4 é fornecido o embasamento teórico necessário para efetuar e compreender as práticas realizadas no minicurso.

1.2. Arquiteturas

A computação heterogênea vem se desenvolvendo rapida e consistentemente em termos de quantidade e variedade de dispositivos aceleradores (Zahran, 2017). Neste minicurso, iremos nos concentrar nos mais comuns: (i) processadores *multicore*; (ii) GPUs discretas/integradas; e (iii) FPGAs, que representam uma categoria bastante diferente das anteriores, como veremos adiante. Também vamos nos concentrar em técnicas usadas em um único processador ou nó computacional e, portanto, técnicas de computação distribuída – importantíssimas para viabilizar a execução de grandes aplicações – estão fora do escopo deste material.

1.2.1. Processadores *multicore*

Processadores modernos possuem uma infinidade de recursos dedicados à melhoria de desempenho. Eles vão desde hierarquias de memória – que buscam atenuar a diferença de frequência entre o processador e a memória principal – até a replicação de unidades funcionais em diferentes granularidades para executar operações em paralelo (Hennessy & Patterson, 2019).

Na Tabela 1.1 são apresentados dados da execução de uma multiplicação de matrizes usando diferentes linguagens e técnicas de otimização que exploram estes recursos (Leiserson et al., 2020)[‡]. Vamos usá-la com referência nesta seção para exemplificar alguns deles e suas técnicas serão exploradas da parte prática do minicurso.

A coluna **Ver.** numera as versões nas diferentes implementações. **Tempo (s)** apresenta o tempo de execução médio em segundos, para ao menos cinco execuções. **GFLOPS** é o número de operações de ponto flutuante executadas por segundo (em bi-

[‡]Códigos-fonte disponíveis em: https://github.com/neboat/Moore/tree/v1.0.1/matrix-multiply-study

Tabela 1.1: Aceleração de um programa que multiplica duas matrizes 4096 por 4096 (Leiserson et al., 2020).

Ver.	Implementação	Tempo (s)	GFLOPS	Abs.	Rel.	Pico (%)
1	Python	25.552,48	0,005	1	–	0,00
2	Java	2.372,68	0,058	11	10,8	0,01
3	C	542,67	0,253	47	4,4	0,03
4	Parallel loops	69,80	1,969	366	7,8	0,24
5	Parallel divide and conquer	3,80	36,180	6.727	18,4	4,33
6	plus vectorization	1,10	124,914	23.224	3,5	14,96
7	plus AVX intrinsics	0,41	337,812	62.806	2,7	40,45

lhões). As colunas **Abs.** e **Rel.** apresentam respectivamente os *speedups* absolutos (em relação à versão 1) e relativos (em relação à versão anterior). Por fim, a coluna **Pico (%)** apresenta o percentual do pico teórico do processador Intel Xeon E5-2666 usado, que é de 835 GFLOPS. Embora a comparação possa ser exagerada, é possível notar que muitos recursos não podem ser completametne aproveitados, mesmo usando bibliotecas e funções específicas para a otimização deles.

As facilidades de programação das linguagens modernas aumentam o número de operações realizadas, então programar o problema em C torna-o quase 50x mais rápido do que em Python e mais de 4x do que em Java. A partir da **versão 3**, usando a mesma linguagem, ainda é possível torná-lo 1300x mais rápido. A **versão 4** explora o paralelismo entre os 18 núcleos, mas só consegue acelerar 8x em relação à sequencial. A **versão 5** explora o uso de memórias *cache*, dividindo o problema em partes menores para que os dados sejam encontrados com mais frequência nelas. Finalmente, as **versões 6 e 7** usam conjuntos de instruções vetoriais, sendo a última delas obtida por meio do uso de tipos de dados intrínsecos.

Técnicas de otimização baseadas no acesso à memória consistem em explorar os princípios de localidade espacial e temporal das memórias *cache* para reduzir o tempo de acesso aos dados. Enquanto as memórias mais próximas ao processador (e.g. L1, L2) podem funcionar na mesma frequência dele, o acesso à memória principal pode custar dezenas ou centenas de ciclos de relógio, dependendo da arquitetura. Compiladores são capazes de tirar proveito deste tipo de otimização automáticamente, por exemplo, fazendo fusão e/ou intercâmbio de laços aninhados. Em outras situações pode ser necessário refatorar o código para melhorar o número de acertos nas *caches*, por exemplo, fazendo blocagem, como na **versão 5** da tabela.

Já as técnicas de paralelismo – que pode ser de dados ou de controle – são encontradas em diferentes granularidades. O paralelismo em nível de instruções é explorado de diversas maneiras atualmente. O suporte a este tipo de paralelismo é implementado em *hardware*, capaz de detectar oportunidades automaticamente. Recursos de execução fora de ordem são usados para maximizar o paralelismo em nível de instruções, fazendo com que instruções sem dependências possam ser adiantadas. A renomeação de registradores pode ser usada para eliminar dependências entre instruções.

Processadores também possuem múltiplas unidades funcionais para que possam

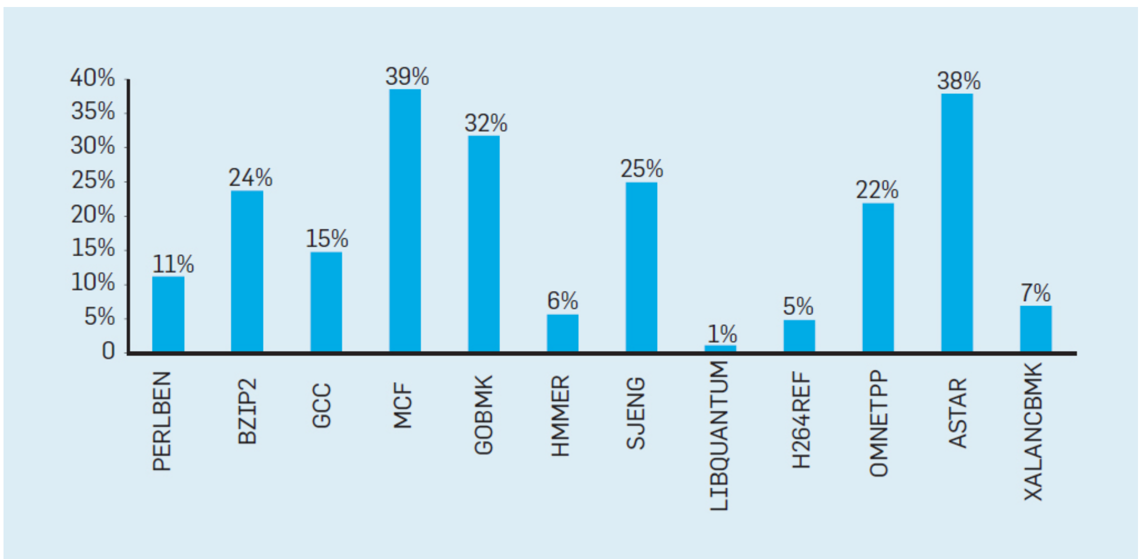


Figura 1.1: Percentual de instruções desperdiçadas em um processador Intel Core i7 para uma variedade de benchmarks de inteiros SPEC (Hennessy & Patterson, 2019).

despachar mais de uma instrução por ciclo de relógio, além de sobrepor sua execução na forma de *pipeline*. Mecanismos de predição de salto são usados para tentar antecipar as próximas instruções e sua execução é realizada de forma especulativa. Em caso de erro na predição, é preciso descartar sua execução e restaurar o estado anterior, causando desperdício de tempo e energia. Na Figura 1.1 são apresentados alguns percentuais destas perdas para *benchmarks* conhecidos (Hennessy & Patterson, 2019).

Talvez o exemplo de paralelismo mais usado atualmente seja o uso de múltiplos processos e *threads* que podem ser executadas em cada um dos núcleos dos processadores modernos. Isso é possível graças à replicação de unidades funcionais e suporte específico de *hardware* para sua sincronização. Para explorar estas funcionalidades são usadas linguagens ou bibliotecas de programação paralela, nas quais o programador é responsável por indicar explicitamente as áreas paralelas do código, como na **versão 4** da Tabela 1.1. Neste minicurso veremos como OneAPI pode ser usada para isso, embora esta não seja a única possibilidade abordada.

Por fim, processadores modernos possuem também instruções SIMD *(single instruction multiple data)* que executam a mesma operação lógica ou aritmética sobre vetores de dados. Instruções deste tipo podem ser geradas automaticamente pelo compilador (como a autovetorização usada na **versão 6**), facilitadas pelo uso de pragmas ou tipos intrínsicos de dados na programação (**versão 7**) ou pelo uso de bibliotecas específicas para este fim. Elas podem ainda ser programadas diretamente em *assembly* quando o código for específico demais para ser paralelizado automaticamente. Em geral, isso ocorre na presença de desvios condicionais em repetições ou acesso aos dados usando índices complexos[§].

[§] Para aprender mais sobre o assunto, veja: https://cvw.cac.cornell.edu/vector/

1.2.2. GPUs integradas

Disponíveis há um bom tempo, as unidades de processamento gráfico (GPUs) surgiram para auxiliar o processador principal na execução de aplicações gráficas intensas, tais como jogos e aplicações multimídia em geral, e posteriormente foram incorporadas aos processadores. Processadores gráficos integrados compartilham memória com o processador principal, mas podem apresentar bom desempenho em aplicações que não requerem processamento gráfico intenso. Além disso, por estarem integrados ao processador principal, consomem menos energia, o que os torna mais adequados para *laptops* e dispositivos móveis em geral.

Boa parte dos recursos empregados nos processadores atuais são também encontrados nos processadores gráficos, tais como *multithreading*, instruções SIMD, hierarquia de memórias *cache*, entre outros. Leiserson et al., 2020 obtiveram fotos anotadas dos processadores Intel com GPUs integradas na WikiChip[¶] e mediram a proporção da GPU em relação à área total do chip. Os percentuais encontrados para os processadores de quatro núcleos foram os seguintes: *Sandy Bridge* (18%), *Ivy Bridge* (33%), *Haswell* (32%), *Skylake* (de 40% até 60%, dependendo da versão), *Kaby Lake* (37%) e *Coffee Lake* (36%). Isso demonstra que o processamento gráfico tem ocupado áreas significativas do chip, assim como as memórias *cache*.

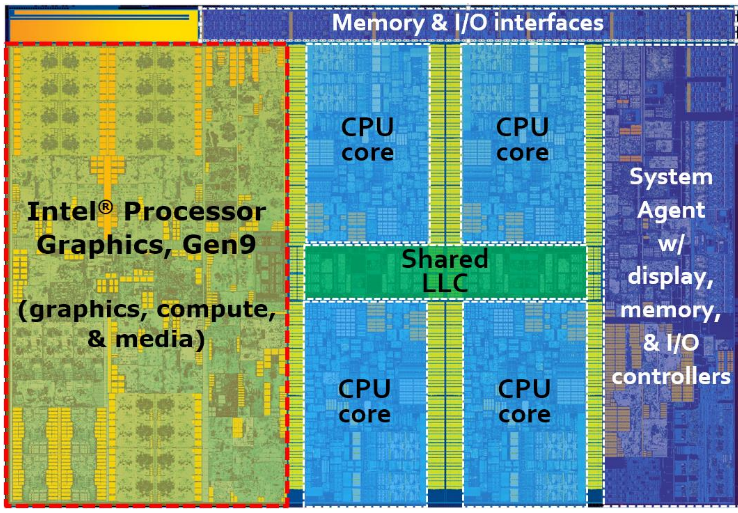


Figura 1.2: Processador Intel® Core™ i7 6700K de 4 núcleos com GPU Gen9 GT2 integrada. Nesta configuração a GPU tem 24 *slices* agrupados em 8 *subslices*, cada um com 8 unidades de execução (EU).

Na Figura 1.2 é apresentada a arquitetura da **Gen9 GT2 GPU**, integrada na microarquitetura *Skylake* (Junking, 2015) no processador Intel® CoreTM i7 processor 6700K. Esta arquitetura foi escolhida por estar bem documentada e disponível no ambiente DevCloud para o minicurso.

Na referida arquitetura, uma conexão bidirecional em anel é usada para conectar os núcleos do processador, a unidade gráfica e a comunicação externa do chip (memória,

[¶] https://en.wikichip.org/

display, barramento PCIe e um controlador EDRAM opcional). Cada núcleo do processador é conectado individualmente. A topologia favorece um projeto modular, já que não há um árbitro central que precisa controlar todas as conexões ao anel.

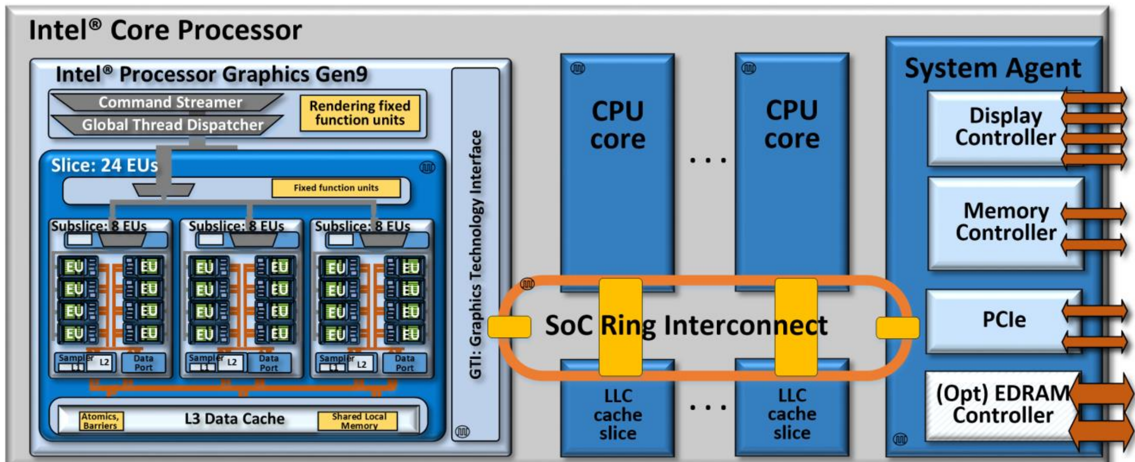


Figura 1.3: Interconexão em anel com múltiplos domínios de relógio.

Na Figura 1.3 é apresentada a topologia de comunicação. Além da linha de dados com 32 bits, *request*, *snoop* e *acknowledge* são usadas. Em algumas configurações de processadores, há um último nível de *cache* (LLC) compartilhada entre os núcleos e a GPU na qual cada núcleo é alocado a uma fatia dela, embora o esquema da *hash* para o endereço permita a cada um deles enxergar toda a LLC.

O projeto da GPU Gen9 também é modular, permitindo atender processadores de diversos segmentos. Os componentes de computação mais básicos são chamados de unidades de execução (EU). Estas unidades de execução são agrupadas em grupos de 8 EUs, chamados *subslices*, que são ainda agrupados em um, dois ou três *slices*, dependendo do modelo do processador.

Na Figura 1.4 é apresentada uma unidade de execução (EU) e seu agrupamento em um *subslice*. Cada EU faz uma combinação de *multithreading* simultâneo (SMT) e *multithreading* intercalado de granulação fina (IMT), além permitirem despachos múltipos em seus pipelines, com instruções SIMD de inteiros e de ponto flutuante e 128 registradores de propósito geral de 32 bits para cada *thread*, totalizando 28 Kbytes por EU.

Um par de FPUs SIMD está presente em cada EU, o que as torna capazes de executar 16 operações de 32 bits por ciclo. Uma das FPUs tem capacidade para operações de 64 bits de precisão dupla e funções matemáticas transcendentes. Dadas essas 8 EUs com 7 *threads* cada, um único *subslice* possui recursos de hardware dedicados para um total de 56 *threads* simultâneas.

Hierarquia de memória

A hierarquia de memória completa desta geração de processadores, bem como as larguras de banda para acesso, podem ser vista na Figura 1.5. A principal vantagem em compar-

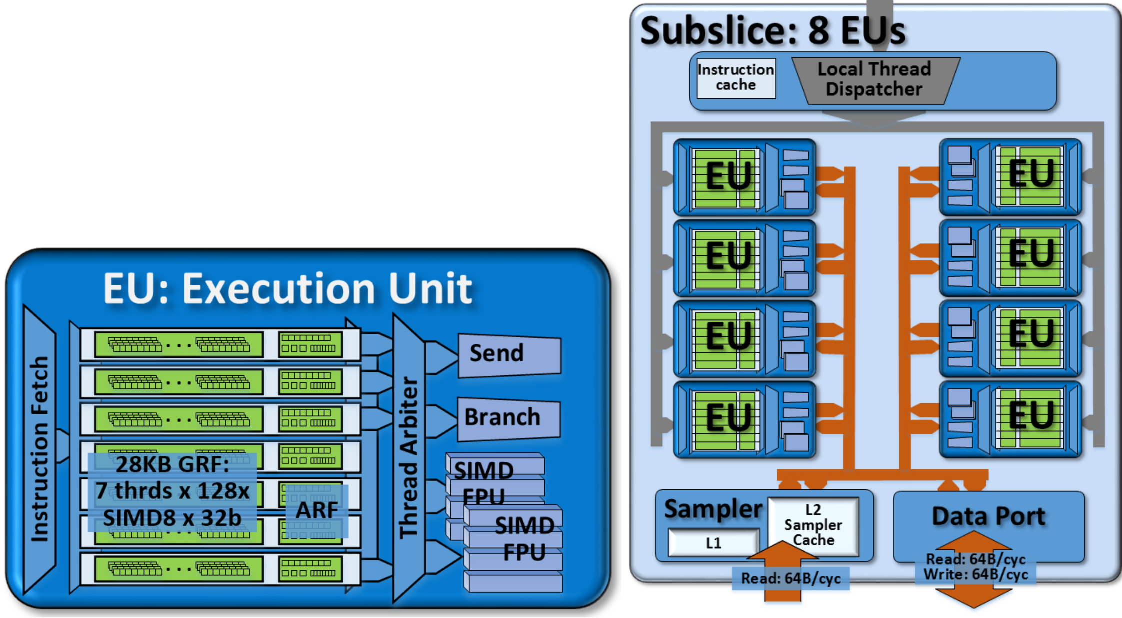


Figura 1.4: Um unidade de execução (EU) e seu agrupamento em um *subslice*.

tilhar memória com o processador é que se elimina a necessidade de transferir *buffers*, o que gasta tempo e energia, além de facilidades de virtualização de hardware.

Uma memória local compartilhada, situada junto com a L3, suporta dados gerenciados pelo programador para compartilhamento entre *threads* de hardware da UE dentro do mesmo *subslice*. A interface de barramento de leitura/gravação entre cada *subslice* e a memória local compartilhada tem 64 bytes de largura. Em termos de latência, o acesso à memória local compartilhada é semelhante ao acesso à *cache* de dados L3. No entanto, a memória local compartilhada possui mais bancos, o que pode permitir maior largura de banda em padrões de acesso não alinhados em 64 bytes ou não adjacentes na memória.

Na Tabela 1.2 são apresentas algumas capacidades da GPU integrada Intel® HD Graphics 530, um modelo da **Gen9** descrita nesta seção. Não se trata do última geração disponível no mercado, mas sim na DevCloud para realização deste minicurso.

Tabela 1.2: Capacidades de uma GPU integrada Intel® HD Graphics 530

Configurações		
Unidades de Execução (EU)	24	8 EUs × 3 *subslices* × 1 *slice*
Threads de *hardware*	168	24 EUs × 7 *threads*
Instâncias concorrentes de *kernel*‖	5376	168 *threads* × SIMD-32
Tamanho da cache (L3) de dados (Kbytes)	512	1 *slice* × 512 Kbytes/*slice*
Memória local compartilhada (Kbytes)	192	3 *subslices* × 64 Kbytes/*subslice*
***Throughput* (picos)**		
32b float (FLOP/ciclo)	384	24 EUs × (2 × SIMD-4 FPU) × (MUL+ADD)
64b double float (FLOP/ciclo)	96	24 EUs × SIMD-4 FPU × (MUL+ADD) × ½
32b integer (IOP/ciclo)	192	24 EUs × (2 × SIMD-4 FPU) × (ADD)

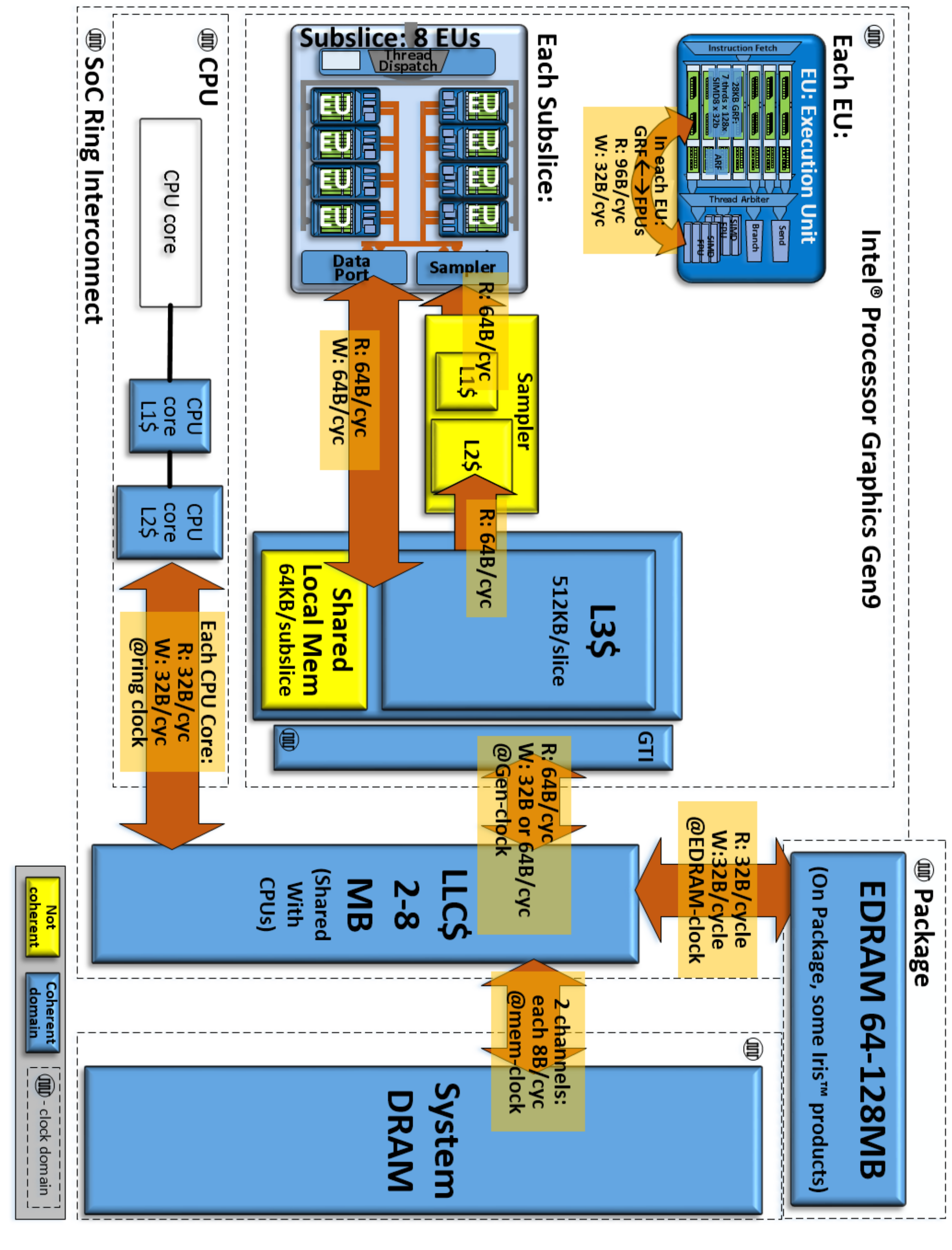


Figura 1.5: Hierarquia de memória com as larguras de banda para acesso.

1.2.3. GPUs discretas

GPUs discretas ou *offboard* possuel sua própria memória, costumam demandar mais energia e estão sujeitas ao gargalo de comunicação com o processador *host* imposto pelo barramento. No entanto, possuem uma capacidade de processamento muito maior, principalmente pela grande quantidade de núcleos e alta vazão de acesso à sua memória. A principal consideração ao usar este acelerador é calcular se o ganho de desempenho nos *kernels* acelerados compensa em face à sobrecarga de transferência de dados entre a CPU e a GPU (Boyer et al., 2013).

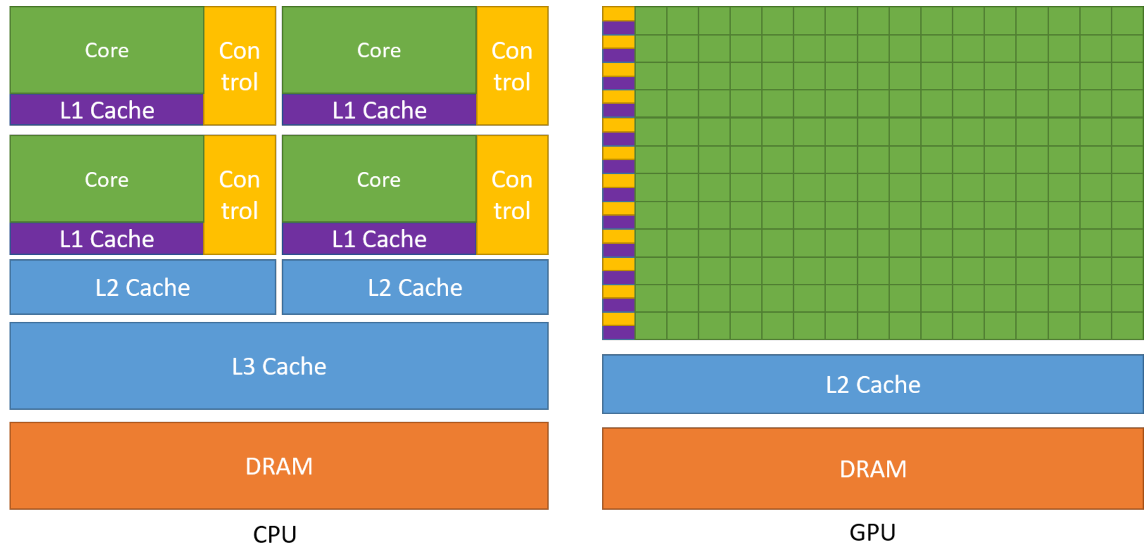


Figura 1.6: A GPU dedica mais transistores ao processamento de dados.

Na Figura 1.6 é apresentado um comparativo entre uma CPU e uma GPU evidenciando a diferença de área dedicada aos núcleos de processamento (NVIDIA, 2022). Um processador possui estruturas complexas de controle e *cache* para minimizar a latência de acesso à memória e executar rapidamente uma ou poucas *threads*. Por sua vez, a GPU é capaz de executar milhares de *threads*, desde que se comportem de maneira bastante homogênea. Isso a torna especialmente interessante para aplicações massivamente paralelas.

Neste minicurso, faremos experimentos com GPUs discretas da Intel, de acordo com a disponibilidade na DevCloud. Os participantes poderão posteriormente testar a possilidade de geração de código para GPUs da NVIDIA.

1.2.4. FPGAs

FPGAs são uma categoria bastante distinta de aceleradores, já que não possuem uma arquitetura fixa usando o modelo clássico de busca e execução de instruções. Enquanto procesadores, gráficos ou convencionais, possuem unidades funcionais fixas que executam um determinado conjunto de instruções, nos FPGAs recursos reconfiguráveis são usados para construir um *pipeline* customizado para a aplicação no qual os dados fluem.

[‖]e.g. OpenCL *work-items*

Podemos dividir as soluções computacionais em dois grandes grupos: (i) *hardware* dedicado, normalmente circuitos integrados de aplicação específica (ASIC); e (ii) *software* executando em processadores de propósito geral. Enquanto a primeira abordagem oferece excelente desempenho, ela se mostra totalmente inflexível. Podemos afirmar que a segunda abordagem constitui seu oposto, pois enquanto se pode alterar facilmente o *software* desenvolvido, o modelo de execução que busca instruções de um conjunto fixo delas na memória é bastante ineficiente em termos de desempenho.

Os FPGAs, principais dispositivos usados na computação reconfigurável, permitem uma solução intermediária entre as duas formas sobreditas, conforme se apresenta na Figura 1.7. Seus circuitos são construídos de forma a permitir que sejam reconfigurados inúmeras vezes, o que lhes confere a capacidade de operar com desempenho de *hardware* e flexibilidade de *software*.

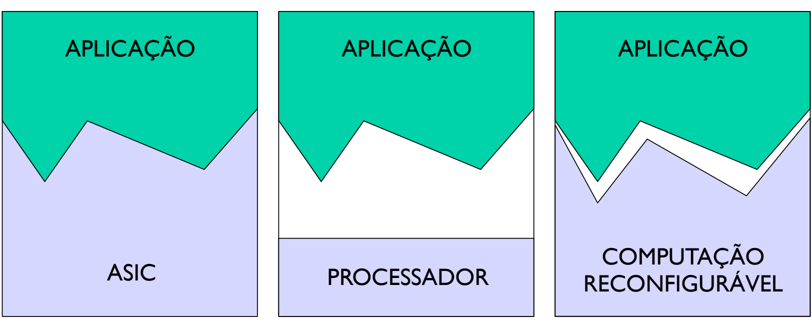


Figura 1.7: Computação reconfigurável comparada às soluções de *hardware* dedicado (ASIC) e *software* executando em processador de propósito geral (Menotti, 2010)

O principal recurso que permite a reconfiguração do circuito é a *Look-Up Table* (LUT) que consiste em um arranjo de multiplexadores, cujas entradas são alimentadas por bits de memória (em geral voláteis) que podem ser gravados para definir a função lógica desejada. Os controles dos multiplexadores são usados como entradas da função, fazendo com que os bits gravados sejam direcionados à saída de acordo com a entrada da função lógica, conforme apresentado na Figura 1.8.

As LUTs são combinadas com registradores para formar os chamados elementos lógicos, mas como isso é feito e até esta nomenclatura varia bastante de acordo com o fabricante e o modelo do FPGA. Além dos elementos lógicos reconfiguráveis, os FPGAs costumam ter blocos de memória internos (*block RAMs)* de tamanhos variados, blocos de DSP[**] (e.g. MAC[††]) para realizar cálculos matemáticos de forma otimizada e até mesmo um ou mais processadores (Crockett et al., 2014).

Como vimos na subseção anterior, os processadores se tornaram heterogêneos, o que lhes confere vantagens em termos de desempenho, apesar de dificultar sua programação. Os FPGA também evoluíram bastante desde a década de 80, quando foram inventados, atingindo atualmente uma dezena de milhões de elementos lógicos programáveis (Intel Corp., 2020a). Atualmente, eles são usados principalmente como aceleradores acoplados a um ou mais processadores *host*. Se por um lado isso elimina a necessidade

[**] *Digital Signal Processing*
[††] *Multiply And Accumulate*

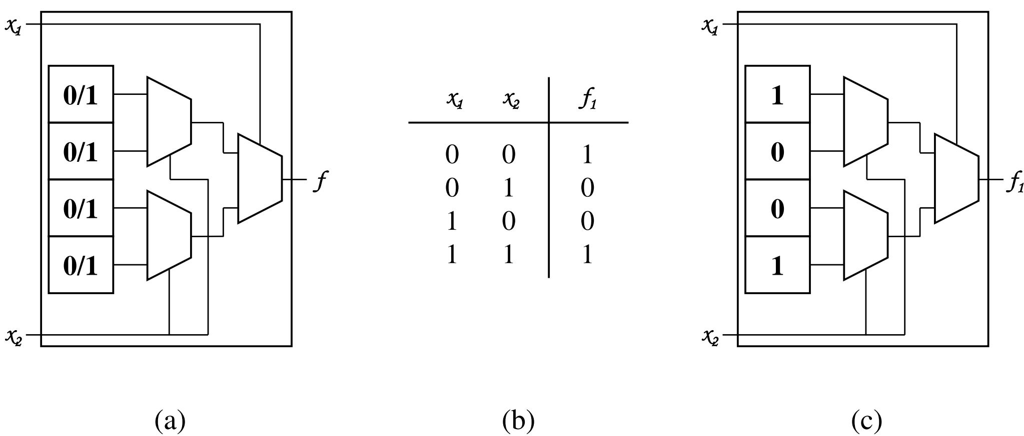


Figura 1.8: Implementação de uma função lógica em uma LUT (Menotti, 2010)

de se projetar todo o sistema em *hardware* – apenas as partes críticas são transferidas ao acelerador, deixando o restante a cargo do processador – por outro há a necessidade de comunicação, que atualmente demanda a implementação de esquemas complexo, inviáveis sem o uso de metodologias avaçadas (Mbakoyiannis et al., 2018).

Os últimos modelos de FPGAs produzidos pelos principais fabricantes ultrapassam 10 e 9 milhões de elementos lógicos respectivamente, mais de dois mil pinos de E/S de propósito geral (GPIO) e capacidades de comunicação de vários terabits/segundo (Intel Corp., 2020c; Xilinx Inc., 2020). Com esta imensa quantidade de recursos é possível devenvolver projetos realmente grandes, o que os torna inviáveis de serem completamente desenvolvidos com HDLs e demandam necessariamente por ferramentas de síntese de alto nível. As metodologias de desenvolvimento adotadas por eles oferecem várias ferramentas, permitindo que desenvolvedores de software e de hardware as escolham de acordo com sua preferência. Neste minicurso vamos tratar de uma metodologia mais voltada para desenvolvedores de software (Intel Corp., 2020a).

1.3. SYCL

SYCL[‡‡] é uma camada de abstração que permite escrever código para processadores heterogêneos usando C++ padrão em único código fonte para o *host* e o acelerador. Além de usar as técnicas conhecidas das linguagens orientadas a objetos, conceitos de C++ importantes (*templates*, *lambdas* e inferência de tipos) adquirem particular relevo neste contexto. Além desses conceitos, as principais classes usadas na implementação e comunicação com os *kernels* e os respectivos escopos para sua programação merecem ser descritos (Khronos Group, 2020b).

SYCL segue o modelo de execução, conjunto de recursos do *runtime* e recursos do dispositivo inspirados no padrão OpenCL. Este padrão impõe algumas limitações na gama completa de recursos da linguagem C++ que o SYCL é capaz de suportar. Apesar de garantir a portabilidade do código em uma ampla variedade de dispositivos, essas limitações restringem o código que pode ser escrito na sintaxe C++ padrão quando o alvo

[‡‡]Pronuncia-se “sickle” (en) / “sicou” (pt)

é um dispositivo SYCL. Em particular, o código do dispositivo SYCL, conforme definido pela especificação, não suporta chamadas de função virtuais, ponteiros de função em geral, exceções, informações de tipo em tempo de execução ou o conjunto completo de bibliotecas C++ que podem depender desses recursos ou dos recursos de um determinado compilador. No entanto, essas restrições podem ser amenizadas por extensões específicas (Khronos Group, 2020b).

O esquema de compilação de múltiplos passos usado pelo padrão SYCL evita que usuários tenham que aprender novas linguagens, favorece o reuso de software e a obtenção de implementações eficientes, pois permite usar recursos já consolidados da linguagem C++ e seus compiladores para gerar código para o *host* e dispositivos.

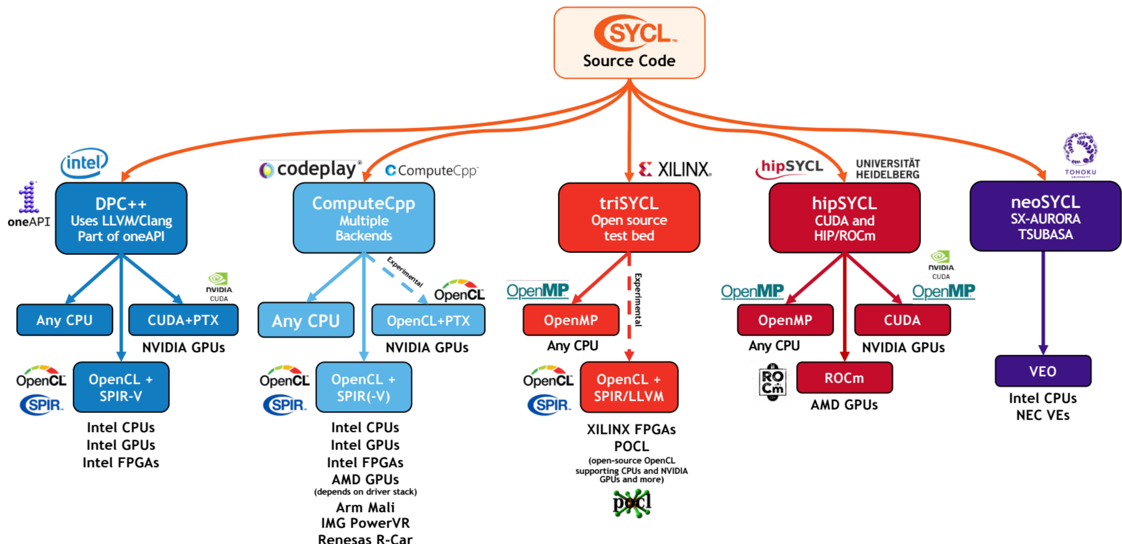


Figura 1.9: Algumas das implementações SYCL disponíveis atualmente

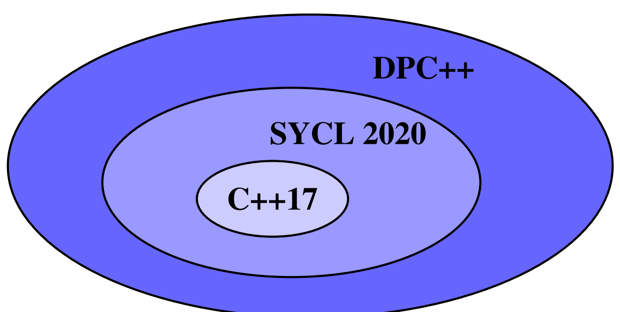


Figura 1.10: Relação de DPC++ com SYCL e C++

Na Figura 1.9 estão algumas implementações SYCL disponiveis atualmente. OneAPI é a implementação SYCL da Intel, a qual adiciona extensões, algumas delas incorporadas ao padrão na atualização da versão 1.2.1 para a 2020. A linguagem foi denominada Data Parallel C++ (DPC++), mas trata-se de C++ padrão, tecnicamente implementável como biblioteca de templates, pois não depende de novas palavras-chave ou alterações sintáticas em C++. No entanto, implementações eficazes de DPC++ ou SYCL requerem

suporte de compilação e tempo de execução específicos (Reinders et al., 2020). A relação entre as tecnologias é apresentada na Figura 1.10.

Para facilitar o entendimento, uma breve explicação de conceitos da linguagem C++ importantes para o contexto do minicurso são fornecidos a seguir. Usuários bem familiarizados com a linguagem e seus recursos podem saltar sem prejuízo para a Subseção 1.3.4

1.3.1. Templates

Templates são a forma como a linguagem C++ implementa a programação genérica, na qual parâmetros para classes e funções podem ser independentes de tipo. No exemplo da Figura 1.11, a função `GetMax` irá funcionar para qualquer tipo de dado que possa ser comparado com o operador <, bastando informar o tipo de dado nos colchetes angulares.

```
1  #include <iostream>
2
3  template <class T>
4  T GetMax (T a, T b) {
5      T result;
6      result = (a > b) ? a : b;
7      return (result);
8  }
9
10 int main () {
11     int i = 7, j = 5, k;
12     float l = 1.0, m = 5.5, n;
13     k = GetMax<int>(i, j);
14     n = GetMax<float>(l, m);
15     std::cout << k << std::endl;
16     std::cout << n << std::endl;
17     return 0;
18 }
```

Figura 1.11: Exemplo de uso de um template

A biblioteca de classes do padrão SYCL está baseada neste estilo de programação, o que a torna genérica. No exemplo a seguir um `buffer` de inteiros com duas dimensões é criado. O tipo de dado a ser armazenado nele é o primeiro parâmetro dentro dos colchetes angulares (< >) e o número de dimensões do *buffer* é o segundo.

```
1 // Cria um buffer de 2 dimensões com 8x8 inteiros
2 buffer<int, 2> my_buffer { range<2>{ 8,8 } };
```

1.3.2. Lambdas

Em C++11 e posterior, uma expressão *lambda* – frequentemente chamada de *lambda* – é uma maneira conveniente de definir um objeto de função anônimo (fechamento ou clausura) bem no local onde é invocado ou passado como um argumento para uma função. Em

vez de definir uma classe nomeada com um operador (), mais tarde fazer um objeto dessa classe e, finalmente, invocá-lo, podemos usar uma abreviação. Normalmente, *lambdas* são usados para encapsular algumas linhas de código que são passadas para algoritmos ou métodos assíncronos (Stroustrup, 2013).

O exemplo da Figura 1.12 aborda os principais aspectos desta funcionalidade da linguagem C++, relevantes para o nosso curso. A função *lambda* `f` (que não é mais anônima, já que demos um nome a ela!) é definida na linha 9. Indicamos dentro dos colchetes que a variável *i* será capturada por referência e `j` por valor. Depois declaramos que ela tem um argumento inteiro *a*, além de indicarmos com `->` que ela deve retornar um `double` (se omitíssemos isso, o compilador inferiria `int`). Finalmente temos a expressão ou trecho de código que deve ser executando quando ela for invocada.

```
1  #include <iostream>
2  #include <typeinfo>
3
4  int main()
5  {
6     int i = 1;
7     int j = 2;
8
9     auto f = [&i, j](int a) -> double { return i + j + a; };
10
11    i = 4;
12    j = 8;
13
14    std::cout << f(1)  << std::endl;
15    std::cout << typeid(f(42)).name()  << std::endl;
16 }
```

Figura 1.12: Exemplo de uma função *lambda*

Em SYCL elas são um forma conveniente de separar o código a ser executado no acelerador, comumente chamado de *kernel*.

1.3.3. Inferência de tipos

Outro recurso importante para facilitar a programação é a inferência automática de tipos de variáveis em tempo de compilação. O trecho de código a seguir é válido em C++17, pois o compilador é capaz de inferir o tipo do vetor `v` declarado a partir de sua inicialização, bem como de iterar automaticamente sobre ele usando a palavra chave `auto` para o elemento `e`. Experimente trocar o tipo de dado do vetor para ver o que acontece.

```
1  std::vector v {2, 7, 42};
2  for (auto e : v)
3      std::cout << e << std::endl;
```

Na versão C++11, o código precisaria ser o seguinte. Note que é preciso explicitar

os tipos de dados envolvidos, o que torna o código escrito muito mais complexo e dependente do tipo de dado a ser tratado. Em versões anteriores, nem seria possível declarar o vetor com inicialização dos dados.

```
std::vector<int> v {2, 7, 42};
for (std::vector<int>::iterator e = v.begin(); e != v.end();
    ++e)
    std::cout << *e << std::endl;
```

Como se pode ter uma ideia a partir deste exemplo, a linguagem C++ tem evoluído bastante ultimamente, a ponto de atualmente podermos considerá-la uma linguagem moderna – tal como Python e outras linguagens interpretadas – com a vantagem de oferecer alto desempenho e segurança de tipos (o que geraria um erro de execução em uma linguagem interpretada pode ser detectado em tempo de compilação em C++).

1.3.4. OneAPI e Data Parallel C++

OneAPI é a implementação da Intel para o padrão SYCL. Os programas **oneAPI** são escritos em **Data Parallel C++ (DPC++).** Ele é baseado nos benefícios de produtividade do C++ moderno e em construções familiares e incorpora o padrão SYCL para paralelismo de dados e programação heterogênea. DPC++ é uma linguagem de código fonte único (*single source*) onde o código do *host* e de aceleradores heterogêneos (*kernels*) podem ser misturados nos mesmos arquivos fonte. Um programa DPC++ é chamado no computador *host* e transfere a computação para um acelerador. Os programadores usam C++ familiar e construções de biblioteca com funcionalidades adicionais, como *queue* para direcionamento de trabalho, *buffer* para gerenciamento de dados e *parallel_for* para paralelismo direcionando quais partes da computação e dados devem ser descarregados.

Dispositivo

A classe `device` representa os recursos dos aceleradores em um sistema que utiliza Intel® oneAPI Toolkits. A classe de dispositivo contém funções de membro para consultar informações sobre o dispositivo, o que é útil para programas SYCL onde vários dispositivos são criados. A função `get_info` fornece informações sobre o dispositivo[§§], tais como: (i) Nome, fornecedor e versão do dispositivo; (ii) IDs de item de trabalho local e global; e (iii) Largura para tipos integrados, frequência de relógio, largura e tamanhos de *cache*, online ou offline.

```
queue q;
device my_device = q.get_device();
std::cout << "Device: " <<
    my_device.get_info<info::device::name>() << std::endl;
```

[§§]Consulte a lista completa em Khronos Group, 2020a

Seletor de dispositivo

A classe `device_selector` permite a seleção em tempo de execução de um dispositivo específico para executar *kernels* com base em heurísticas fornecidas pelo usuário. O código SYCL da Figura 1.9 mostra diferentes seletores de dispositivo (linhas 13 a 16). Ele pode ser usado para se descobrir possíveis aceleradores em um sistema.

```
//==============================================================
// Copyright © 2020 Intel Corporation
//
// SPDX-License-Identifier: MIT
// =============================================================
#include <CL/sycl.hpp>

using namespace cl::sycl;

int main() {
  //# Create a device queue with device selector

  gpu_selector selector;
  //cpu_selector selector;
  //default_selector selector;
  //host_selector selector;

  queue q(selector);

  //# Print the device name
  std::cout << "Device: " <<
    q.get_device().get_info<info::device::name>() << std::endl;

  return 0;
}
```

Figura 1.13: Código SYCL com diferentes seletores de dispositivos

Fila (Queue)

A classe `queue` submete grupos de comandos a serem executados pelo *runtime* SYCL. A fila é um mecanismo em que trabalho é submetido a um dispositivo. Uma fila mapeia para um dispositivo e várias filas podem ser mapeadas para o mesmo dispositivo, conforme apresentado na Figura 1.14.

```
q.submit([&](handler& h) {
    //COMMAND GROUP CODE
});
```

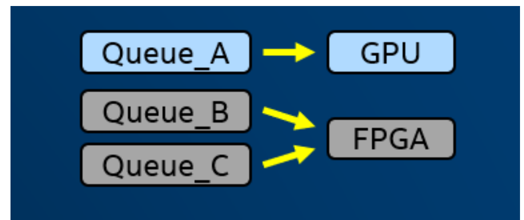


Figura 1.14: Filas associadas a dispositivos

Kernel

A classe `kernel` encapsula métodos e dados para a execução de código no dispositivo quando um grupo de comando é instanciado. O objeto do *kernel* não é explicitamente construído pelo usuário, é construído quando uma função de envio do *kernel*, como `parallel_for`, é chamada

```
1 q.submit([&](handler& h) {
2   h.parallel_for(range<1>(N), [=](id<1> i) {
3     A[i] = B[i] + C[i]);
4   });
5 });
```

Escolhendo onde os *kernels* do dispositivo rodam

O trabalho é submetido a filas e cada fila é associada a exatamente um dispositivo (por exemplo, uma GPU ou FPGA específico). Você pode decidir a qual dispositivo uma fila está associada (se desejar) e ter quantas filas desejar para despachar o trabalho em sistemas heterogêneos. Na Tabela 1.3 são apresentadas algumas possibilidades de construtores para a classe `queue`.

Dispositivo de destino	**Fila**
Crie uma fila destinada a qualquer dispositivo:	queue()
Crie uma fila destinada a classes pré-configuradas de dispositivos:	queue(cpu_selector{}); queue(gpu_selector{}); queue(intel::fpga_selector{}); queue(accelerator_selector{}); queue(host_selector{});
Crie um dispositivo específico de destino de fila (critérios personalizados):	class custom_selector : public device_selector {int operator()(... // Any logic you want! ... queue(custom_selector{});

Tabela 1.3: Alguns construtures da classe `queue`

Linguagem DPC++, *runtime* e escopos

A linguagem e o *runtime* DPC++ consistem em um conjunto de classes, modelos e bibliotecas C++. Em geral, podemos dividir o código em duas partes:

1. **Código que executa no host:** os recursos completos do C++ estão disponíveis no escopo da aplicação e do grupo de comandos.
2. **Código que executa no dispositivo:** no escopo do *Kernel*, existem limitações no C++ aceito.

Kernels paralelos

Kernels paralelos permitem que várias instâncias de uma operação sejam executadas em paralelo. Isso é útil para descarregar (offload) a execução paralela de um for-loop básico no qual cada iteração é completamente independente e em qualquer ordem. *Kernels* paralelos são expressos usando a função `parallel_for` Um simples loop 'for' em um aplicativo C++ é escrito como abaixo:

```
for(int i=0; i < 1024; i++){
    a[i] = b[i] + c[i];
});
```

A seguir está como você pode descarregá-lo para o acelerador:

```
h.parallel_for(range<1>(1024), [=](id<1> i){
    a[i] = b[i] + c[i];
});
```

Kernels Paralelos Básicos

A funcionalidade dos *Kernels* Paralelos Básicos é exposta por meio das classes `range`, `id` e `item`. A classe `range` é usada para descrever o espaço de iteração da execução paralela e a classe `id` é usada para índice uma instância individual de um *kernel* em uma execução paralela

```
h.parallel_for(range<1>(1024), [=](id<1> i){
    // CODE THAT RUNS ON DEVICE

});
```

O exemplo acima é suficiente se tudo o que você precisa é o índice (`id`), mas se você precisa do intervalo (`range`) em seu código de *kernel*, então você pode usar a classe `item` em vez da classe `id`, que você pode usar para consultar o `range` como mostrado abaixo. A classe `item` representa uma instância individual de uma função do *kernel* e expõe funções adicionais para propriedades de consulta do intervalo de execução

```
h.parallel_for(range<1>(1024), [=](item<1> item){
    auto i = item.get_id();
    auto R = item.get_range();
    // CODE THAT RUNS ON DEVICE

});
```

Kernels NDRange

Os *Kernels* Paralelos Básicos são uma maneira fácil de paralelizar um loop `for`, mas não permitem a otimização do desempenho no nível do hardware. O *Kernel* NDRange (N-dimensões) é outra maneira de expressar paralelismo que permite ajuste de desempenho de baixo nível, fornecendo acesso à memória local e mapeamento de execuções para unidades de computação no hardware. Todo o espaço de iteração é dividido em grupos menores chamados *work-groups*, *work-items* dentro de um *work-groups* são agendados em uma única unidade de computação no hardware. A Figura 1.15 ilustra o conceito.

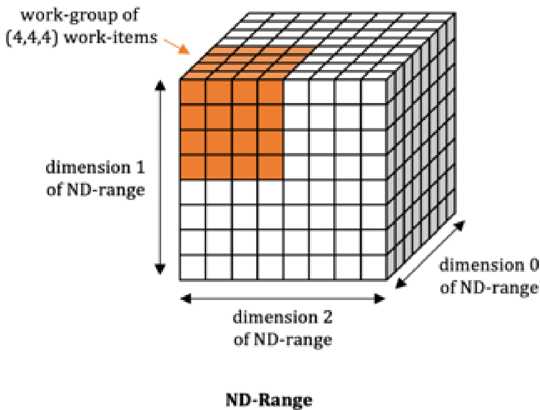


Figura 1.15: *Kernel* NDRange (N-dimensões) com *work-items* em destaque

O agrupamento de execuções do *kernel* em *work-groups* permite o controle do uso de recursos e de balanceamento de carga na distribuição de trabalho. A funcionalidade dos *kernels* NDRange é exposta por meio das classes `nd_range` e `nd_item`. A classe `nd_range` representa um intervalo de execução agrupado (*grouped execution range*) usando o intervalo de execução global e o intervalo de execução local de cada grupo de trabalho. A classe `nd_item` representa uma instância individual de uma função do *kernel* e permite consultar o intervalo e o índice do grupo de trabalho.

```
h.parallel_for(nd_range<1>(range<1>(1024),range<1>(64)),
  ↪ [=](nd_item<1> item){
    auto idx = item.get_global_id();
    auto local_id = item.get_local_id();
    // CODE THAT RUNS ON DEVICE
});
```

***Buffers*, *Accessors* e a anatomia do código SYCL**

Buffers encapsulam dados em uma aplicação SYCL em ambos dispositivo e *host. Accessors* é o mecanismo para acessar os dados do *buffer*.

Os programas que utilizam SYCL requerem a inclusão do cabeçalho `cl/sycl.hpp`. Recomenda-se empregar a instrução de namespace para evitar a digitação de referências repetidas do namespace `cl::sycl`.

```
1 #include <CL/sycl.hpp>
2 using namespace cl::sycl;
```

Programas SYCL são C++ padrão. O programa é invocado no computador *host* e transfere a computação para o accelerador. O programador usa *queues*, *buffers*, dispositivos e abstrações de *kernel* do SYCL para direcionar quais partes da computação e dados devem ser descarregados.

Como primeiro passo em um programa SYCL, criamos uma *queue*. Descarregamos a computação para um dispositivo enviando tarefas para uma fila. O programador pode escolher CPU, GPU, FPGA e outros dispositivos por meio do *selector*. Este programa usa o padrão aqui, o que significa que o *runtime* SYCL seleciona o dispositivo mais capaz disponível em tempo de execução usando o seletor padrão. Falaremos sobre os dispositivos, seletores de dispositivo e os conceitos de *buffers*, *accessors* e *kernels* nos próximos módulos, mas abaixo está um programa SYCL simples para você começar com os conceitos acima.

O dispositivo e o *host* podem compartilhar memória física ou ter memórias distintas. Quando as memórias são distintas, o descarregamento de computação requer copia de dados entre o *host* e o dispositivo. SYCL não requer que o programador gerencie as cópias dos dados. Ao criar *buffers* e *accessors*, o SYCL garante que os dados estejam disponíveis para o *host* e o dispositivo sem nenhum esforço do programador. O SYCL também permite ao programador controle explícito sobre a movimentação de dados quando é necessário obter o melhor desempenho.

Em um programa SYCL, definimos um *kernel*, que é aplicado a cada ponto em um espaço de índice. Para programas simples como este, o espaço de índice mapeia diretamente para os elementos do arranjo. O *kernel* é encapsulado em uma função *lambda* C++. A função *lambda* recebe um ponto no espaço de índice como uma matriz de coordenadas. Para este programa simples, a coordenada do espaço de índice é a mesma que o índice da matriz. O `parallel_for` no programa abaixo aplica o *lambda* ao espaço do índice. O espaço de índice é definido no primeiro argumento de `parallel_for` como um intervalo unidimensional de 0 a N-1.

O código da Figura 1.16 mostra a adição de vetor simples usando SYCL. Leia os comentários abordados na etapa 1 à etapa 6.

```
1  void dpcpp_code(int* a, int* b, int* c, int N) {
2    //Etapa 1: criar uma fila de dispositivos
3    //(o desenvolvedor pode especificar um tipo de dispositivo por
   ↪  meio do seletor de dispositivo ou usar o seletor padrão)
4    queue q;
5    //Etapa 2: criar buffers (representa tanto a memória do host
   ↪  quanto a do dispositivo)
6    buffer<int,1> buf_a(a, range<1>(N));
7    buffer<int,1> buf_b(b, range<1>(N));
8    buffer<int,1> buf_c(c, range<1>(N));
9    //Etapa 3: enviar um comando para execução (assíncrona)
10   q.submit([&](handler &h){
11   //Etapa 4: crie acessores (acessors) de buffer para acessar os
   ↪  dados do buffer no dispositivo
12   auto A = buf_a.get_access<access::mode::read>(h);
13   auto B = buf_b.get_access<access::mode::read>(h);
14   auto C = buf_c.get_access<access::mode::write>(h);
15   //Etapa 5: enviar um kernel (lambda) para execução
16   h.parallel_for(range<1>(N), [=](item<1> i){
17     //Etapa 6: escrever um kernel
18     //As invocações do kernel são executadas em paralelo
19     //O kernel é invocado para cada elemento do intervalo
20     //A invocação do kernel tem acesso ao id de invocação
21     C[i] = A[i] + B[i];
22     });
23   });
24 }
```

Figura 1.16: Um programa de adição de vetores em SYCL

Dependência implícita com acessores (*accessors*)

Acessores criam dependências de dados no gráfico SYCL que ordenam as execuções do *kernel*. Se dois *kernels* usam o mesmo *buffer*, o segundo *kernel* precisa aguardar a conclusão do primeiro *kernel* para evitar condições de corrida. Na Figura 1.17 é mostrado um grafo de dependência de dados, de acordo com os *buffers* que cada *kernel* acessa.

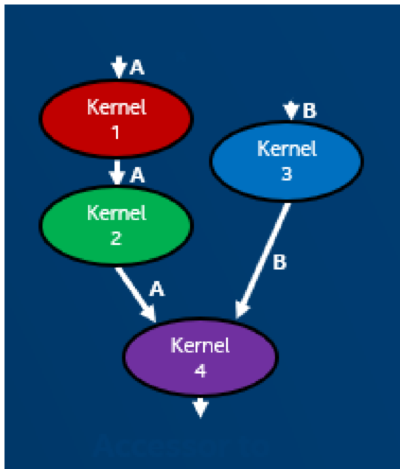


Figura 1.17: Grafo de dependência de dados entre *kernels*

Acessores do host e sincronização

O acessor de *host* é aquele que usa o destino de acesso do *buffer* do *host*. Ele é criado fora do escopo do grupo de comando e os dados aos quais ele dá acesso estarão disponíveis no *host*. Eles são usados para sincronizar os dados de volta ao host, construindo os objetos de acesso do *host*. A destruição do *buffer* é a outra maneira de sincronizar os dados de volta ao *host*.

O *buffer* assume a propriedade dos dados armazenados no vetor. A criação do acessor de *host* é uma chamada blocante e só retornará depois que todos os *kernels* SYCL enfileirados que modificam o mesmo *buffer* em qualquer fila concluam a execução e os dados estejam disponíveis para o *host* por meio desse acessor de *host*.

Unified Shared Memory (USM)

OneAPI possui também um modelo unificado de acesso à memória (USM[¶¶]) pelo *host* e dispositivo acelerador. Neste modelo, chamadas `malloc_shared` são usadas para alocar os dados, que são visíveis tanto no host quanto no dispositivo. O *runtime* fica resposável por transferi-los automaticamente.

Na Figura 1.18 é apresentada uma nova versão do código da Figura 1.16, modificada para o modelo USM. Note que as etapas 2 e 4 não são mais necessárias. Embora este

[¶¶]Do inglês: *Unified Shared Memory*

```
void dpcpp_code(int* a, int* b, int* c, int N) {
  //Etapa 1: criar uma fila de dispositivos
  //(o desenvolvedor pode especificar um tipo de dispositivo por
  ↪ meio do seletor de dispositivo ou usar o seletor padrão)
  queue q;
  //Etapa 3: enviar um comando para execução (assíncrona)
  q.submit([&](handler &h){
  //Etapa 5: enviar um kernel (lambda) para execução
  h.parallel_for(range<1>(N), [=](item<1> i){
    //Etapa 6: escrever um kernel
    //As invocações do kernel são executadas em paralelo
    //O kernel é invocado para cada elemento do intervalo
    //A invocação do kernel tem acesso ao id de invocação
    c[i] = a[i] + b[i];
    });
  });
}
```

Figura 1.18: Um programa de adição de vetores em SYCL

modelo seja mais simples, em certos casos pode valer a pena gerenciar explicitamente as transferências de dados para evitar transferências desnecessárias.

1.4. Práticas

Esta seção fornece o embasamento teórico dos algoritmos que serão usados no minicurso. Os códigos completos estão disponíveis no repostitório do curso***.

1.4.1. Filtro Sobel

O Filtro Sobel é um algoritmo de processamento de imagens usado para detecção de bordas que funciona por meio de um operador de diferenciação discreta, calculando uma aproximação do gradiente da função de intensidade da imagem. O operador usa duas máscaras 3×3 que são convoluídas com a imagem original para calcular as aproximações das derivadas, um para mudanças horizontais e outro para mudanças verticais. Se definirmos A como a imagem de origem, e Gx e Gy como duas imagens que em cada ponto contêm as aproximações derivadas horizontal e vertical, respectivamente, os cálculos são os seguintes:

$$G_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} \times A, G_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \times A$$

Depois, em cada ponto da imagem, as aproximações de gradiente resultantes podem ser combinadas para fornecer a magnitude do gradiente. Um exemplo de sua aplicação é apresentado na Figura 1.19:

***https://github.com/menotti/sycl-wscad-2022

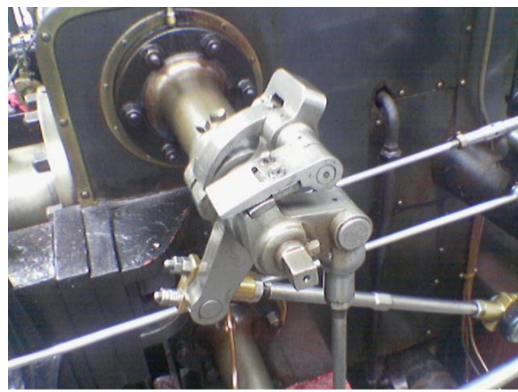
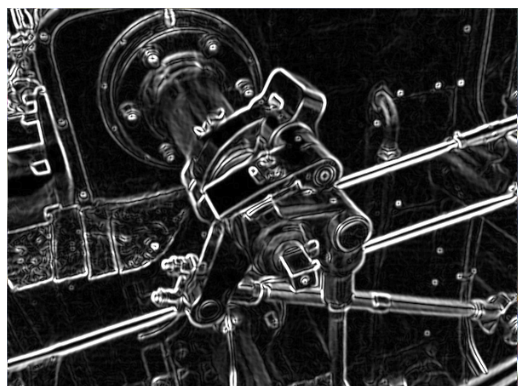



Figura 1.19: Exemplo da aplicação do filtro Sobel

$$\sqrt{{G_x}^2 + {G_y}^2}$$

1.4.2. Transformada de Hough

A transformada de Hough é usada em aplicações de visão computacional. Depois que uma imagem foi processada com um algoritmo de detecção de bordas, como um filtro Sobel, você fica com uma imagem monocromática (preto/branco). É útil para muitos algoritmos de detecção adicionais considerar a imagem como um conjunto de linhas. No entanto, uma imagem de pixels em preto e branco não é uma representação conveniente ou útil dessas linhas para algoritmos como detecção de objeto. A Transformada de Hough é uma transformação de pixels em um conjunto de "votos de linha". É comumente conhecido que uma linha pode ser representada em uma forma de interceptação de declive:

$$y = mx + b$$

Nesta forma, cada linha pode ser representada por duas constantes únicas, a inclinação (m) e a interceptação y (b). Portanto, cada par (m, b) representa uma reta única. No entanto, esta forma apresenta alguns problemas. Primeiro, uma vez que as linhas verticais têm uma inclinação indefinida, não pode representar linhas verticais. Em segundo lugar, é difícil aplicar técnicas de limiarização. Portanto, por razões computacionais em muitos algoritmos de detecção a forma normal de Hesse é usada. Esta forma possui a equação abaixo:

$$\rho = x\ cos\ \theta + y\ cos\ \theta$$

Nesta forma, cada linha única é representada por um par (ρ, θ). Esta forma não tem nenhum problema em representar linhas verticais, e você aprenderá como o limiar pode ser facilmente aplicado depois que a Transformada de Hough for aplicada. Na Figura 1.20 é mostrado o que os valores de ρ e θ representam na equação. Para cada linha que você deseja representar (veja a linha vermelha na imagem), haverá uma linha exclusiva que você pode desenhar da origem até ela com a distância mais curta (veja a linha cinza na imagem). Outra maneira de ver isso é a linha perpendicular à linha vermelha que

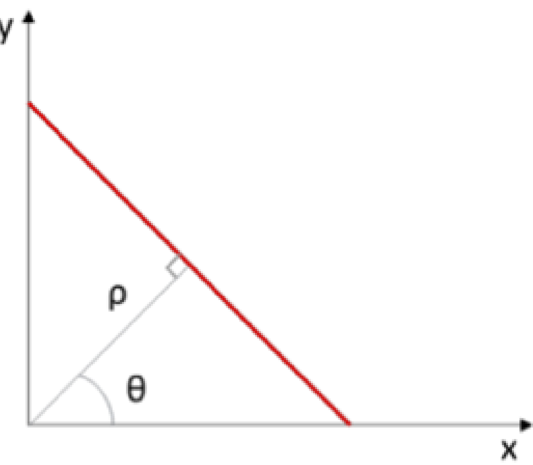
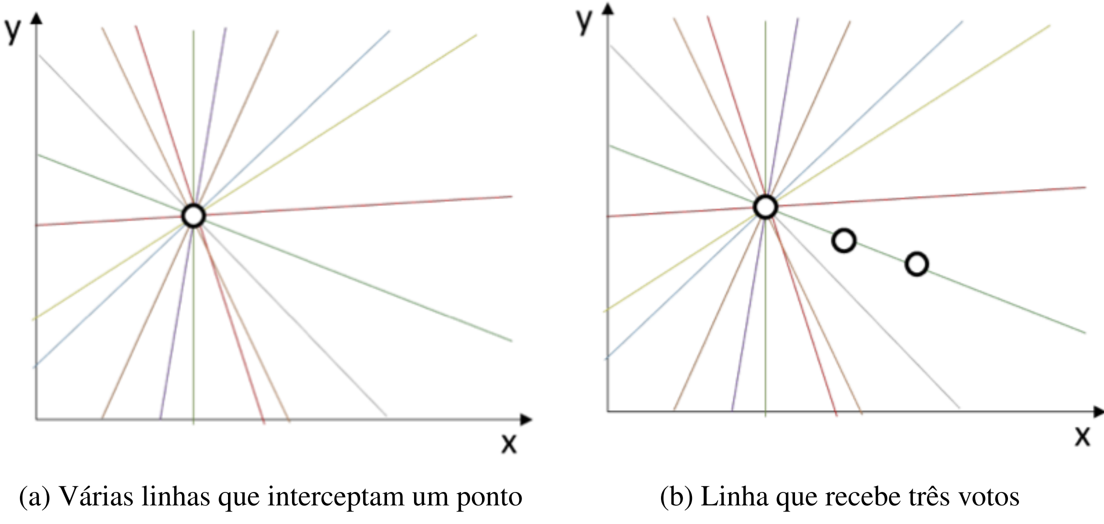


Figura 1.20: Representação dos valores de ρ e θ na equação

cruza a origem. ρ é a distância da linha mais curta que pode ser traçada da origem até a linha que você deseja representar. θ é o ângulo do eixo x para essa linha.

Ao trabalhar com uma imagem, um canto da imagem é tradicionalmente considerado como a origem (a origem não está no centro), então o maior valor que ρ pode ser é a medida da diagonal da imagem. Você pode escolher que os valores de ρ sejam todos positivos ou que possam ser positivos e negativos. Se você escolher que todos os valores de ρ sejam positivos, o intervalo de θ vai de 0 a 360 graus. Se você escolher que ρ possa ser positivo ou negativo, o intervalo de θ é de 0 a 180 graus. Esses intervalos são quantizados a fim de definir um espaço de solução finito.



(a) Várias linhas que interceptam um ponto

(b) Linha que recebe três votos

Figura 1.21: Demonstração da Transformada de Hough

Lembre-se de que a imagem antes de ser inserida na Transformada de Hough já passou por um algoritmo de detecção de bordas e, portanto, é monocromática (cada pixel é preto ou branco). As arestas detectadas são representadas por pixels brancos. A

Transformada de Hough transformará os pixels brancos em uma matriz de votos para linhas. Cada pixel branco na imagem é potencialmente um ponto em um conjunto de linhas. A Figura 1.21a representa as linhas das quais um pixel pode potencialmente fazer parte. (Nota: nem todas as linhas potenciais são desenhadas, isso serve apenas para fins ilustrativos.) Uma linha com cada inclinação potencial que passa por aquele pixel é potencialmente uma linha que aparece na imagem. Portanto, um voto será acumulado para cada uma dessas linhas.

O código percorrerá cada pixel da imagem, acumulando votos nas linhas conforme avança. Conforme uma linha acumula mais votos, a probabilidade de ser uma representação correta para uma linha na imagem aumenta. Assim, conforme visualizado na Figura 1.21b, a linha verde acumulará três votos, o que a tornará um candidato mais provável do que as demais linhas. Um limite pode ser facilmente aplicado, portanto, simplesmente definindo a quantidade de votos que é "suficiente"para definir se uma linha está presente ou não.

1.4.3. Bitonic sort

O Bitonic sort é um algoritmo de divisão e conquista que utiliza sequências bitônicas para realizar a ordenação. Esse algoritmo é baseado no merge sort. Para entender o algoritmo Bitonic Sort é necessário saber o que significa bitonic. O binotic deriva de sequências bitônicas, que por sua vez são sequências que possuem um período crescente e outro decrescente, por exemplo a sequência onde os números 1, 2 e 3 estão em ordem crescente e depois os números 3, 2 e 1 estão em ordem decrescente.

Melhor definindo, o Bitonic Sort é um algoritmo de ordenação baseado na comparação de elementos, no qual ordena um conjunto de dados ao converter uma lista de números em uma sequência bitônica. Um conjunto de dados é bitônico se existe um índice *i* para o qual todos os elementos menores ou iguais a *i* estão em ordem crescente e todos os elementos maiores ou iguais a *i* estão em ordem decrescente.

Figura 1.22: Exemplo de sequência bitônica.

Na Figura 1.22 o *i* da sequência bitônica, o valor que separa a parte crescente da parte decrescente, é o índice onde tem o valor 3. Sendo assim, os valores a esquerda do 3 estão em ordem crescente e os à direita do 3 estão em ordem decrescente.

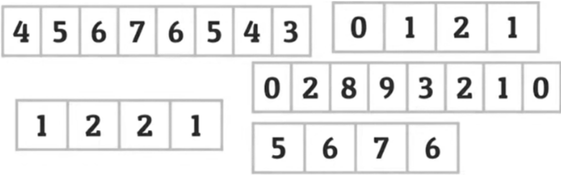


Figura 1.23: Outros exemplos de sequências bitônicas.

Na Figura 1.23 estão algumas sequências bitônicas, onde todas elas possuem um

valor i que separa a parte crescente da parte decrescente. Importante ressaltar que uma sequência que só decresce ou que só cresce também são consideradas sequências bitônicas. Uma sequência que só decresce a parte que cresce é considerada vazia, mas não nula, da mesma maneira para sequências que só crescem, a parte decrescente é vazia, mas não nula.

Criando sequências bitônicas

Para criar sequências bitônicas alguns passos devem ser seguidos. Primeiro é necessário repartir a sequência de valores desordenados em duplas de valores, como é representado na Figura 1.24.

Figura 1.24: Dividindo uma lista de valores em sequências bitônicas de 2 números cada.

Após isso, agrupe duas sequências em uma única sequência bitônica até restar apenas uma sequência no final. Importante observar que algumas operações devem ser feitas para obter a sequência ordenada no final.

No exemplo da Figura 1.25, os valores da esquerda (1, 3 7 e 5) se tornaram uma única sequência crescente, onde o 1 e 3 já estavam de maneira crescente mas o 7 e 5 precisavam ser trocados. Já na parte da direita (8, 6, 4 e 2) as duas sequências, crescente e decrescente se tornaram apenas uma, porém diferentemente da sequência à esquerda, essa sequência precisava estar em ordem decrescente após fazer a junção das duas anteriores. Sendo assim, os valores 2 e 4 estavam em ordem decrescente e precisavam ser trocados, enquanto os valores 8 e 6 já estavam em ordem decrescente, não necessitando de trocas.

1	3	5	7	8	6	4	2
Bitonic Sequence				Bitonic Sequence			

Figura 1.25: Agrupando duas sequências em uma.

Um processo semelhante ocorreu para obtermos o resultado da Figura 1.26. Como o intuito do algoritmo era resultar em uma sequência crescente, ambas as sequências anteriores, crescente e decrescente, necessitam ser trocadas de maneira a resultar na sequência final. O processo mais detalhado para obtenção da sequência da Figura 1.26 vai ser explicado mais adiante.

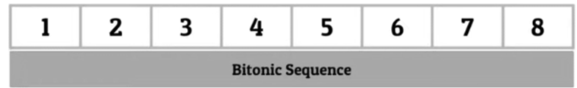


Figura 1.26: Sequência resultante do processo de BitonicSort.

Por ser um algoritmo que necessita de repartir uma sequência maior em sequências menores de dois elementos, o BinoticSort só pode ser aplicado em sequências de tamanho

de base dois. Ou seja, sequências de tamanhos: 2, 4, 8, 16, 32, 64, 128, 256. Existem variações do algoritmo que sanam essa limitação, mas não serão abordadas nesse trabalho.

Processo geral do Bitonic Sort

Para aplicar o BitonicSort em alguma sequência são necessárias duas etapas principais, a primeira foi demonstrada na Subseção 1.4.3 e consiste na quebra da sequência maior em sequências menores de tamanho dois, a segunda etapa é combinar as sequências bitônicas em sequências cada vez maiores, conforme as Figuras 1.27 e 1.28. Percebe-se que a estrutura desse algoritmo é leva à recursão.

Resultado da etapa 1:

Figura 1.27: Sequência maior repartida em 4 sequências menores.

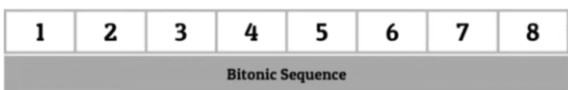


Figura 1.28: Sequências menores mescladas para atingir o resultado final.

Aplicando o Bitonic Sort

Agora vamos aplicar o Bitonic Sort para ordenar uma sequência de valores desordenada passo a passo para melhor compreensão. Primeiro passo é obter um vetor de valores desordenado, considere o exemplo da Figura 1.29. A sequência representa um estado inicial de um vetor desordenado de valores.

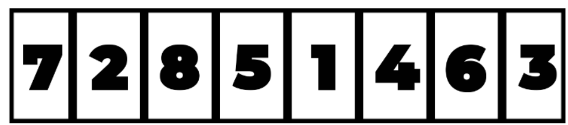


Figura 1.29: Sequências desordenada.

Segundo passo é dividir a grande sequência em sequências menores de tamanho 2 crescentes e decrescentes de maneira intercalada, obtendo a sequência da Figura 1.30.

As marcas coloridas sobre cada número significa a ordem que eles devem estar dentro de seus grupos, por exemplo, os valores 7 e 2 estão marcados com o verde, denotando que precisam estar em ordem crescente ao final dessa etapa, necessitando de uma ordenação interna. Resultando na sequência representada pela da Figura 1.31.

Como as outras duplas possuem marcações condizentes as suas ordens, elas não precisaram ser trocadas entre si. Por exemplo, a dupla 8 e 5 esta em ordem decrescente,

Figura 1.30: Sequências menores de tamanho 2 crescentes e decrescentes de forma intercalada.

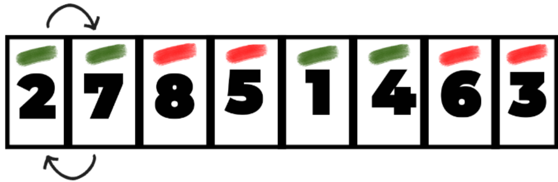


Figura 1.31: Sequências menores de tamanho 2 crescentes e decrescentes de forma intercalada.

assim, pela sua marcação não precisar ser trocada. Terceiro passo é juntar duas duplas e torná-las crescente e outras duas duplas e torná-las decrescente, é necessário fazer isso de maneira intercalada até não restar mais duplas de sequência, resultando na Figura 1.32.

Figura 1.32: Sequências menores de tamanho 2 crescentes e decrescentes de forma intercalada.

Após juntar as duplas resultando em sequências de 4 valores, ordenar conforme as marcações em cada número e unir duas sequências. Fazendo o movimento conforme a Figura 1.33.

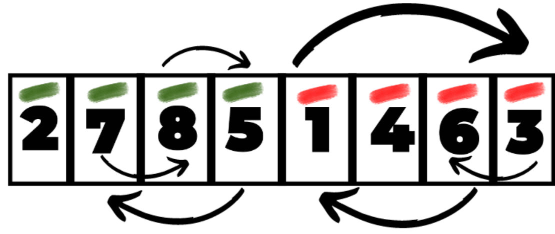


Figura 1.33: Sequências menores de tamanho 2 crescentes e decrescentes de forma intercalada.

O resultado do movimento anterior é a sequência na Figura 1.34.

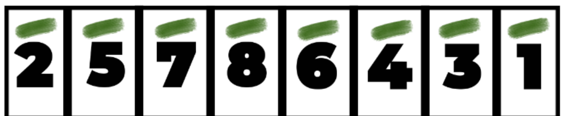


Figura 1.34: Sequências menores de tamanho 2 crescentes e decrescentes de forma intercalada.

Para finalizar a ordenação, é necessário fazer a intercalação final entre os elementos do vetor. Esta etapa é feita comparando cada elemento da primeira metade da sequência com cada elemento da segunda metade da sequência. Da seguinte maneira: compara-se o 2 com o 6, o 5 com o 4, o 7 com o 3 e 8 com o 1. Das 4 comparações, apenas 1 não foi necessário fazer a troca, que foi a primeira. Isso ocorreu, pois o resultado das trocas precisava estar em ordem crescente para funcionar.

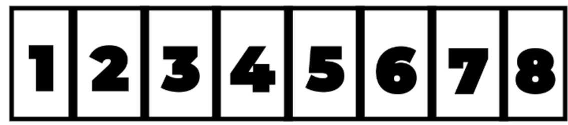


Figura 1.35: Sequência final após a execução do algoritmo.

Algoritmo paralelo do Bitonic

Na Figura 1.36 é apresentada uma parte do algoritmo paralelo do bitonic sort. Esse trecho de código é responsável por fazer a separação da sequência maior em menores sequências de tamanho dois. Na Figura 1.36 também é apresentado um trecho do algoritmo responsável por fazer as mesclas da sequência Bitônica dívida até resultar em uma sequência ordenada de tamanho igual ao inicial antes da divisão.

Por fim, também é possível realizar as mesclas entre as sequências menores nas linhas 20 à 25, onde checa se o sentido necessário (crescente ou decrescente) concorda com o sentido atual dos números. Na Figura 1.36 fica claro como essa comparação funciona e qual o intuito de realizá-la.

```
1  void ParallelBitonicSort(int data_gpu[], int n, queue &q) {
2    int size = pow(2, n);
3    int *a = data_gpu;
4
5    for (int step = 0; step < n; step++) {
6      for (int stage = step; stage >= 0; stage--) {
7        int seq_len = pow(2, stage + 1);
8        int two_power = 1 << (step - stage);
9
10       // Offload the work to kernel.
11       q.submit([&](auto &h) {
12         h.parallel_for(range<1>(size), [=](id<1> i) {
13           // Assign the bitonic sequence number.
14           int seq_num = i / seq_len;
15           int swapped_ele = -1;
16           int h_len = seq_len / 2;
17           if (i < (seq_len * seq_num) + h_len) swapped_ele = i +
    ↪   h_len;
18           int odd = seq_num / two_power;
19           bool increasing = ((odd % 2) == 0);
20           if (swapped_ele != -1) {
21             if (((a[i] > a[swapped_ele]) && increasing) ||
22                 ((a[i] < a[swapped_ele]) && !increasing)) {
23               int temp = a[i];
24               a[i] = a[swapped_ele];
25               a[swapped_ele] = temp;
26             }
27           }
28         });
29       });
30       q.wait();
31     }  // end stage
32   }    // end step
33 }
```

Figura 1.36: Parte do algoritmo Bitonic Sort em SYCL/DPC++.

Referências

Boyer, M., Meng, J., & Kumaran, K. (2013). Improving GPU Performance Prediction with Data Transfer Modeling. *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, 1097–1106. https://doi.org/10.1109/IPDPSW.2013.236

Ceissler, C., Nepomuceno, R., Pereira, M., & Araujo, G. (2018). Automatic Offloading of Cluster Accelerators. *IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 224–224.

Crockett, L. H., Elliot, R. A., Enderwitz, M. A., & Stewart, R. W. (2014). *The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc*. Strathclyde Academic Media.

Firmansyah, I., & Yamaguchi, Y. (2019). OpenCL Implementation of FPGA-Based Signal Generation and Measurement. *IEEE Access*, *7*, 48849–48859.

Gautier, Q., Althoff, A., Pingfan Meng & Kastner, R. (2016). Spector: An OpenCL FPGA benchmark suite. *2016 International Conference on Field-Programmable Technology (FPT)*, 141–148.

Hennessy, J. L., & Patterson, D. A. (2019). A New Golden Age for Computer Architecture. *Commun. ACM*, *62*(2), 48–60. https://doi.org/10.1145/3282307

Intel Corp. (2020a). *FPGA Add-On for oneAPI Base Toolkit(Beta)*. https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/fpga.html

Intel Corp. (2020b). *Intel oneAPI Toolkits(Beta)*. https://software.intel.com/content/www/us/en/develop/tools/oneapi.html

Intel Corp. (2020c). *Intel Stratix 10 FPGAs*. https://www.intel.com/content/www/us/en/products/programmable/fpga/stratix-10.html

Junking, S. (2015). *The Compute Architecture of Intel® Processor Graphics Gen9*. Recuperado setembro 2, 2022, de https://www.intel.com/content/dam/develop/external/us/en/documents/the-compute-architecture-of-intel-processor-graphics-gen9-v1d0.pdf

Keryell, R., & Yu, L.-Y. (2018). Early Experiments Using SYCL Single-Source Modern C++ on Xilinx FPGA: Extended Abstract of Technical Presentation. *Proceedings of the International Workshop on OpenCL*. https://doi.org/10.1145/3204919.3204937

Khronos Group. (2020a). *SYCL 1.2.1 API Reference Guide*. https://www.khronos.org/files/sycl/sycl-121-reference-guide.pdf

Khronos Group. (2020b). *SYCL Specification*. https://www.khronos.org/sycl/

Leiserson, C. E., Thompson, N. C., Emer, J. S., Kuszmaul, B. C., Lampson, B. W., Sanchez, D., & Schardl, T. B. (2020). There's plenty of room at the Top: What will drive computer performance after Moore's law? *Science*, *368*(6495), eaam9744. https://doi.org/10.1126/science.aam9744

Mbakoyiannis, D., Tomoutzoglou, O., & Kornaros, G. (2018). Energy-Performance Considerations for Data Offloading to FPGA-Based Accelerators Over PCIe. *ACM Trans. Archit. Code Optim.*, *15*(1). https://doi.org/10.1145/3180263

Menotti, R. (2010). *LALP: uma linguagem para exploração do paralelismo de loops em computação reconfigurável* [tese de dout., Universidade de São Paulo]. https://doi.org/10.11606/T.55.2010.tde-17082010-151100

NVIDIA. (2022). *CUDA C++ Programming Guide*. https://docs.nvidia.com/cuda/cuda-c-programming-guide/

Reinders, J., Ashbaugh, B., Brodman, J., Kinsner, M., Pennycook, J., & Tian, X. (2020). *Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL*. Apress.

Stock, F.-W. (2019). *Rapid Prototyping and Exploration Environment for Generating C-to-Hardware-Compilers* [tese de dout., Technische Universität]. Darmstadt.

Stroustrup, B. (2013). *The C++ programming language*. Pearson Education.

Sutter, H. (2011). Welcome to the Jungle. http://herbsutter.com/welcome-to-the-jungle

Xilinx Inc. (2020). *Virtex UltraScale+ VU19P FPGA*. https://xilinx.com/vu19p

Zahran, M. (2017). Heterogeneous Computing: Here to Stay. *Commun. ACM*, *60*(3), 42–45. https://doi.org/10.1145/3024918

Capítulo 2

Inteligência Artificial e Função como Serviço: Provisionando Aplicações com o AWS Lambda

Sayonara S. Araújo (UFC), Francisco R. P. da Ponte (UFC), Victória T. Oliveira (UFC), Wendley S. da Silva (UFC), Dario Vieira (EFREI), Miguel F. de Castro (UFC) e Emanuel B. Rodrigues (UFC)

Abstract

Data can be a valuable resource for companies, as it helps in decision making, performance analysis, and problem-solving. Estimates indicate that an average of 2.5 quintillion bytes of data are generated daily. The use of technology like Artificial Intelligence (AI) is indispensable for extracting useful information from this large amount of data. Companies that provide Cloud Computing services, such as Amazon with its Amazon Web Service (AWS) platform, allow the easy deployment of AI solutions capable of taking advantage of this immense volume of data. An example of these services is AWS Lambda, one of the main tools of what is known as serverless computing. Within this context, this mini-course will introduce the concepts of Function as a Service (FaaS), which encompass the use of AWS Lambda to provision AI applications.

Resumo

Os dados podem ser um recurso muito valioso para as empresas, visto que eles auxiliam na tomada de decisão, análise de desempenho e na resolução de problemas. Estima-se que em média 2,5 quintilhões de bytes são gerados diariamente. Para conseguirmos extrair informações úteis dessa grande quantidade de dados, o uso de uma tecnologia como a Inteligência Artificial (AI) é indispensável. Empresas que prestam serviços de Computação em Nuvem, como a Amazon com sua plataforma Amazon Web Service (AWS), permitem a implantação facilitada de soluções de AI capazes de tirar proveito desse imenso volume de dados. Para isso, pode-se utilizar, por exemplo, o AWS Lambda, uma das principais ferramentas do chamado serviço de computação sem servidor. Dentro desse contexto, o presente minicurso introduz os conceitos de Função como Serviço (FaaS), englobando o uso de AWS Lambda para provisionar aplicações de AI.

2.1. Introdução

Em uma época de destaque para o *Big Data*, negócios enfrentam desafios de lidar com uma grande quantidade de informações presentes no meio digital. Estima-se que 2,5 quintilhões de *bytes* são gerados por dia [1]. Para aproveitar esse volume de informações, é fundamental que haja meios eficientes de lidar com todo esse material.

Nesse contexto, a Inteligência Artificial – *Artificial Intelligence* (AI) – é uma tecnologia relevante para criar produtos que consigam analisar grande volume de dados. A AI se refere resumidamente a sistemas computacionais que tentam imitar a inteligência humana para realizar atividades com base nas informações disponíveis [2]. Devido ao seu potencial, nas últimas décadas, essa área vem impactando significativamente o mundo real [3].

Outra tecnologia que atualmente tem agregado nesse contexto de análise de dados é a Computação em Nuvem, um paradigma que surgiu para suportar o processamento e o armazenamento de grandes volumes de dados em conjuntos de computadores externos [4]. Assim, a junção de ambas as tecnologias permite potencializar o tratamento de informações.

O advento de plataformas de Computação em Nuvem, como a Amazon Web Service (AWS), permitiu a implantação facilitada de soluções capazes de tirar proveito desse imenso volume de dados [5]. Para isso, utiliza-se o chamado serviço de computação sem servidor, especialmente a Função como Serviço – *Function as a Service* (FaaS) –, que torna a implantação de arquiteturas de aplicativos corporativos para contêineres e microsserviços mais fácil e econômica [6].

O AWS Lambda foi a primeira ferramenta de mercado a possibilitar a execução de serviços como FaaS [7] e continua sendo uma das principais [8]. Dentro desse cenário, o presente minicurso introduz o conceito de FaaS, englobando o uso de *AWS Lambda* em soluções de AI. Ao longo deste trabalho, serão apresentados uma fundamentação teórica e alguns direcionamentos práticos para o provisionamento de aplicações de AI na FaaS da AWS. Para isso, o texto a seguir se divide nas seguintes seções: Inteligência Artificial (2.2), Contêiner (2.3), Função como Serviço (2.4), Prática (2.5) e Considerações Finais (2.6).

2.2. Inteligência Artificial

O termo Inteligência Artificial foi cunhado em 1956, em uma conferência no Dartmouth College[1], onde um grupo de matemáticos e cientistas se reuniu para discutir como aspectos de aprendizagem e inteligência podem, a princípio, serem descritos com tamanha precisão que uma máquina (i.e. um *software*) pode ser construída para simulá-los [9].

A disciplina de AI teve um desenvolvimento inicial lento, mas a partir da década de 90 e início do século XXI, com o desenvolvimento de computadores mais potentes e o aumento no volume de dados, as pesquisas nessa área aumentaram consideravelmente [10]. Esse crescimento pode ser visto na Figura 2.1, que mostra a quantidade de trabalhos publicados em AI ao longo dos últimos 30 anos, informação obtida através de uma

[1]Universidade estadunidense localizada no estado de New Hampshire: https://home.dartmouth.edu/.

consulta realizada no site Scopus[2] (acessado em 13 de agosto de 2022).

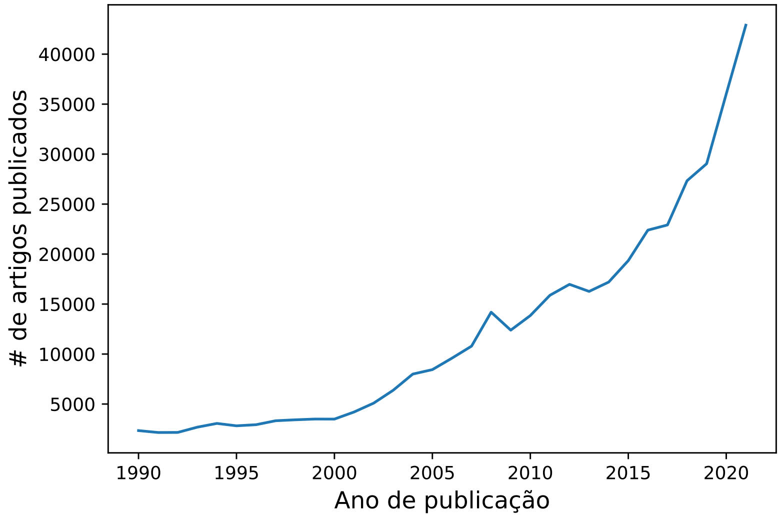


Figura 2.1: Número de artigos publicados em AI por ano desde 1990. Observa-se um crescimento acentuado de artigos publicados na área de AI a partir de 2000 [Autores].

2.2.1. Aprendizado de Máquina

A capacidade dos sistemas aprenderem e melhorarem seu funcionamento a partir de dados foi algo tão significativo para a área de AI, que um novo campo chamado de Aprendizado de Máquina – *Machine Learning* (ML) – surgiu. ML é um conjunto de métodos, ou algoritmos, que utilizam probabilidade e estatística para melhorar o desempenho de uma determinada tarefa [11]. Como podemos observar na Figura 2.2, que mostra um diagrama de Veen, ML é um subconjunto de AI.

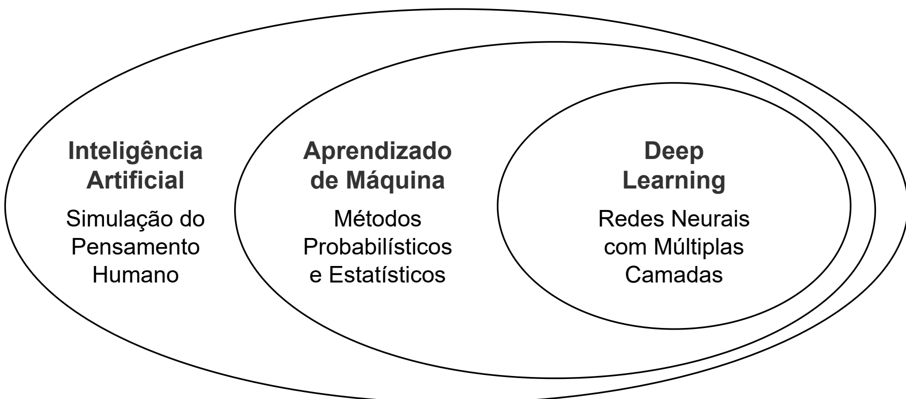


Figura 2.2: Diagrama de Veen que representa graficamente como os conjuntos de AI, ML e *Deep Learning* se correlacionam [Autores].

O processo de aprendizado é chamado de treinamento e, para que isso ocorra, um conjunto de dados deve ser fornecido como entrada para o algoritmo de ML. Padrões

[2] Agregador de diversas bases de dados de publicações acadêmicas: https://www.scopus.com/.

são extraídos desses dados e usados para construir um modelo preditivo capaz de fazer observações precisas em relação a dados nunca vistos [12]. As principais técnicas de aprendizado, as quais diferem entre si dada a presença ou ausência de rótulo nos dados, são: aprendizado supervisionado, semi-supervisionado e não supervisionado.

2.2.2. Redes Neurais Artificiais

Entre as diversas abordagens de ML, vale destacar as Redes Neurais Artificiais – *Artificial Neural Networks* (ANN), que são uma evolução dos métodos estatísticos tradicionais. As ANNs foram inspiradas em como sistemas biológicos processam informações, e são constituídas por nós chamados de neurônios e um conjunto de arestas que interliga esses nós formando uma rede [13]. Por sua vez, um grupo de neurônios forma uma camada e existem três tipos diferentes de camadas, que são: de entrada, oculta e de saída. A Figura 2.3 mostra um exemplo simples de uma rede neural com 7 neurônios: 2 na camada de entrada, 4 na camada oculta e 1 na camada de saída.

As ANNs são treinadas por meio de uma sequência de interações de *feedforward* e *backpropagation*. No processo de *feedforward*, os neurônios da camada oculta, utilizando uma função de ativação, ajustam seus pesos de modo que a rede neural tenha a melhor precisão na interpretação da saída, dada uma determinada entrada. Na camada de saída, calcula-se o erro, dado pela diferença entre a saída da rede neural e o valor desejado. O erro é propagado até a camada de entrada pelo processo de *backpropagation*. Essa sequência de passos é repetida diversas vezes até que o desempenho esperado para o modelo seja alcançado [14].

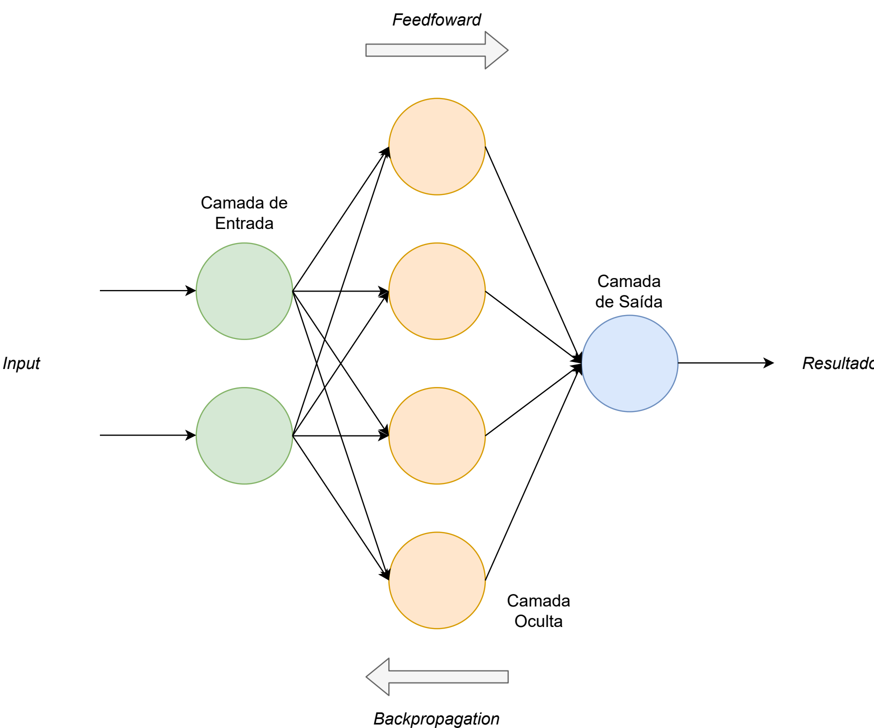


Figura 2.3: Exemplo de uma ANN simples, com neurônios dispostos em três camadas, sendo uma de entrada, uma oculta e uma de saída [Autores].

2.2.3. Aprendizado Profundo

Atualmente, grandes volumes de dados são gerados diariamente a uma velocidade sem precedentes e pelas mais variadas fontes (e.g. saúde, governo, financeiro, redes sociais, etc). Isso se deu graças a tecnologias emergentes, como, por exemplo, a Internet das Coisas – *Internet of Things* (IoT) – e Computação em Nuvem – *Clouding Computing* (CC) [15]. Diante desse cenário, desenvolveu-se a disciplina de *Big Data*, que lida com conjuntos de dados massivos, de difícil armazenamento e complexos de serem tratados e analisados por *softwares* tradicionais de processamento de dados [16].

Dado o crescimento no volume de dados gerados e o aumento no poder de processamento dos computadores modernos, os estudos dos algoritmos de Aprendizado Profundo – *Deep Learning* (DL) –, prosperaram. DL é um tipo de ANN e o termo "profundo" no nome desse método refere-se ao uso de várias camadas ocultas na rede neural, o que permite que o processamento de dados seja feito de uma forma não linear, que se traduz em um ganho de acurácia [17], como pode ser visto na Figura 2.4.

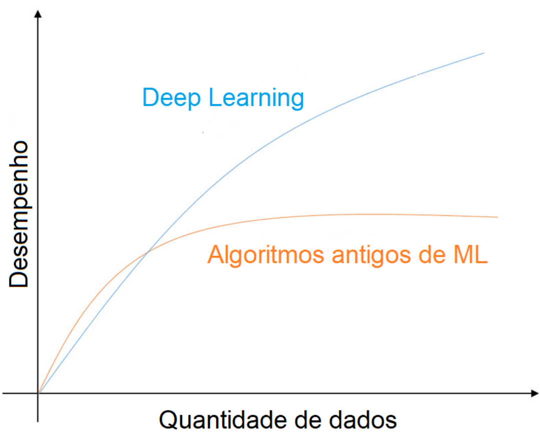


Figura 2.4: Diferença de desempenho dos algoritmos de DL em relação a algoritmos tradicionais de ML, com o aumento da quantidade de dados [Adaptado de Zahangir *et al.* [18]].

Nos nossos experimentos, utilizaremos uma Rede Neural Convolucional – *Convolutional Neural Network* (CNN) –, que é um tipo de algoritmo de DL utilizado no processamento de imagens. CNNs recebem como entrada um vetor de imagens, atribuem pesos às várias características observadas e classificam-na em diferentes classes. A arquitetura típica de uma CNN, como observada na Figura 2.5, é composta por três tipos básicos de camadas, que são: (*i*) convolucional, responsável por extrair recursos de baixo nível; (*ii*) *pooling*, usada para reduzir a complexidade do modelo; e (*iii*) totalmente conectado, responsável por produzir uma interpretação do que foi observado pelo modelo.

Em suma, DL pode alcançar maior precisão devido ao número de camadas ocultas adicionadas à rede e ao grande volume de dados utilizados no treinamento. No entanto, esse ganho de desempenho possui um *trade-off*: o alto poder computacional necessário para processar tamanho volume de informações [20]. Modelos de DL chegam a possuir milhões de parâmetros, como no caso do VGG16 [21], um modelo para classificação de

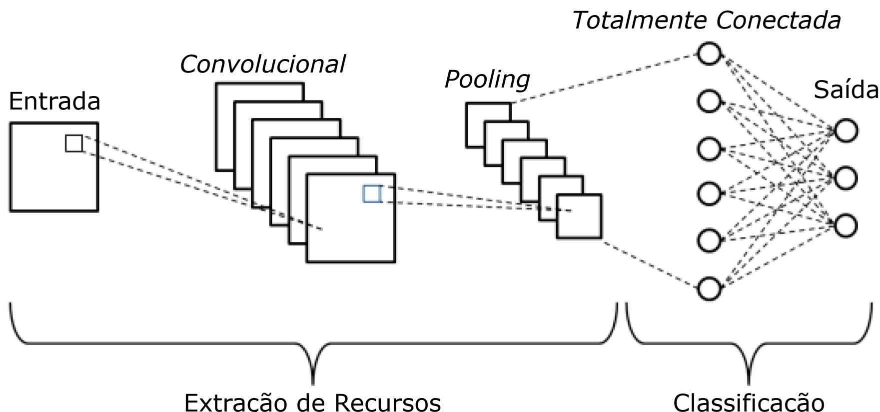


Figura 2.5: Exemplo simplificado da arquitetura de uma CNN com as camadas convolucionais, *pooling* e totalmente conectada [Adaptado de Phung *et al.* [19]].

imagem pré-treinado que analisa 138 milhões de parâmetros para chegar a um resultado.

2.2.4. *Frameworks* de Redes Neurais

Existem diversas ferramentas que podem ser utilizadas por cientistas de dados no desenvolvimento de modelos de aprendizado de máquina. Essas ferramentas são interessantes, pois permitem que os especialistas utilizem a Computação de Alto Desempenho – *High Performance Computing* (HPC) –, para desenvolver *softwares* de IA. Um exemplo disso são os Kits de Desenvolvimento de Software – *Software Development Kits* (SDKs) – da NVIDIA, o NVIDIA HPC SDK[3] e NVIDIA CUDA[4], que permitem que desenvolvedores utilizem todo o poder de processamento das placas gráficas da NVIDIA para melhorar o desempenho dos seus modelos de ML.

Esses SDKs podem ser utilizados através de *frameworks* que auxiliam os especialistas a desenvolverem soluções de forma facilitada. Alguns exemplos de *frameworks* existentes são: i) Tensorflow, biblioteca *open-source* que pode ser utilizada no desenvolvimento e treinamento de modelos de ML [22]; ii) PyTorch, uma biblioteca de aprendizado de máquina *open-source* desenvolvida pelo Facebook [23]; iii) Keras, API desenvolvida em *Python*, que permite o desenvolvimento de modelos de DL [24].

2.2.5. Operações de Aprendizado de Máquina (MLOps)

Como o conjunto de treinamento é finito e frequentemente novos dados (que modificam as propriedades estatísticas dos conjuntos) surgem, nem sempre é possível garantir um alto desempenho dos algoritmos. Essa perca de desempenho, que muitas vezes se traduz na perca de acurácia dos modelos, é um fenômeno chamado de desvio de conceito – *concept drift* [25].

Assim, é necessário que os modelos sejam re-treinados e implantados novamente

[3]Conjunto abrangente de compiladores, bibliotecas e ferramentas utilizadas no desenvolvimento de aplicações HPC: https://catalog.ngc.nvidia.com/orgs/nvidia/containers/nvhpc.

[4]API que permite que processamento de informações seja feito pelas Unidades de Processamento Gráfico – *Graphical Processing Units* (GPU): https://catalog.ngc.nvidia.com/orgs/nvidia/containers/cuda.

em produção de tempos em tempos, garantindo assim a sua eficiência. Isso é possível graças à utilização de Operações de Aprendizado de Máquina – *Machine Learning Operations* (MLOps) –, que é a união entre ML e a cultura DevOps [26]. MLOps fornece um conjunto de práticas que incentiva a colaboração entre cientistas de dados e engenheiros de DevOps, garantindo que a transição entre treinamento e implantação do modelo em produção seja feito de forma automatizada e contínua [27].

2.3. Contêiner

A tecnologia de contêineres foi introduzida em 1979 pela IBM [28]. Nomes diferentes são usados para se referir a contêineres na literatura, incluindo virtualização em nível de sistema operacional e virtualização leve. Um contêiner é uma unidade padrão de *software* que empacota o código e todas as suas dependências para que o aplicativo seja executado de forma rápida e confiável em diferentes ambientes de computação.

Os contêineres fornecem um ambiente isolado em um único *host*, em vez de usar sistema operacional (SO) dedicado como máquinas virtuais [29]. Este isolamento é possível devido ao mecanismo do *kernel Namespace* [30] e *Cgroup* [31]. As tecnologias de virtualização baseadas em *hipervisor* e multitarefas tradicionais têm sido usadas nas últimas duas décadas. Mais tarde, os contêineres entraram no centro das atenções. A tecnologia de contêiner é mais eficiente que a máquina virtual (VM) e, como resultado, muitas organizações estão mudando o ambiente virtualizado para contêineres Linux [32]. As diferenças entre VMs e Contêineres estão principalmente em arquiteturas e funcionalidades. A virtualização baseada em *hypervisor* e em contêiner é mostrada na Figura 2.6.

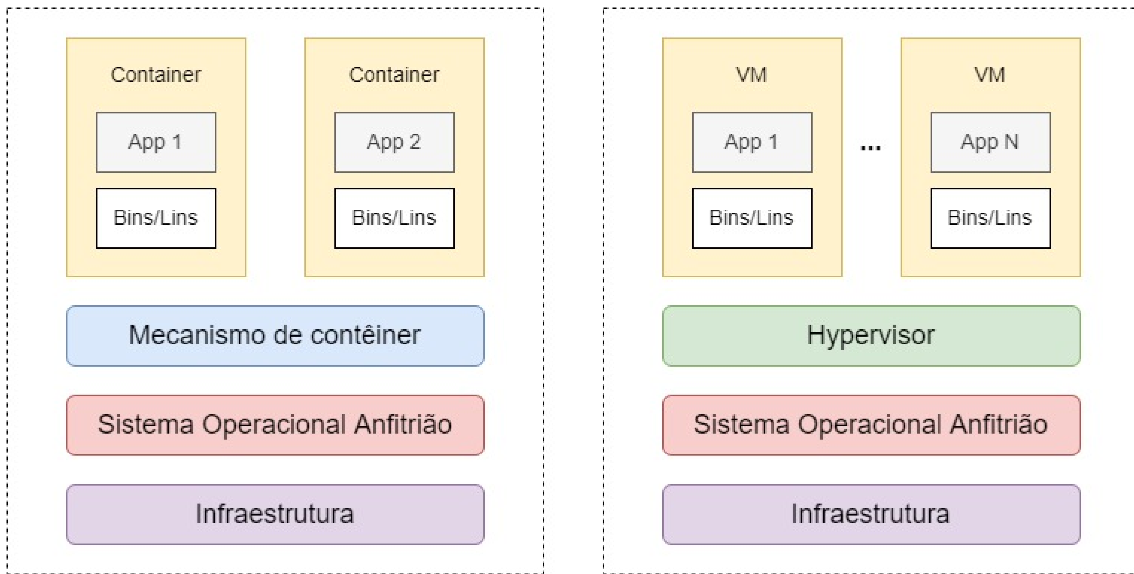


Figura 2.6: Virtualização baseada em *hipervisor* e em contêiner [Autores].

Cada VM contém o Sistema Operacional (SO) convidado, a cópia virtual de *hardware*, o aplicativo e todas as bibliotecas e dependências associadas. Por outro lado, os contêineres virtualizam o sistema operacional e contêm apenas o aplicativo, suas bibliotecas e os arquivos executáveis necessários. Os contêineres virtualizam a camada de

software acima do sistema operacional, onde as VMs virtualizam o *hardware* (memória, CPU, rede, armazenamento). Além disso, os contêineres são mais rápidos, portáteis, leves e eficientes. Um possível motivo para essas vantagens é que os contêineres não contêm SO convidado para todas as instâncias. Assim, os contêineres podem reduzir o consumo de recursos do sistema operacional do *host*. A tecnologia de contêiner e as VMs têm algumas características em comum, como portabilidade, escalabilidade e ciclo de vida de desenvolvimento de software aprimorado [33]. No entanto, o contêiner oferece benefícios significativos sobre as VMs, incluindo peso reduzido, maior eficiência de recursos, maior produtividade do desenvolvedor, melhor balanceamento de carga e tempo de início reduzido [34].

Os contêineres melhoram duas principais desvantagens das VMs [35]:

1. Os contêineres compartilham o mesmo *kernel* do sistema operacional e podem compartilhar recursos enquanto cada VM precisa de sua própria cópia;

2. Os contêineres podem ser iniciados e parados quase instantaneamente, enquanto as VMs precisam de um tempo considerável para iniciar [34].

Os contêineres também provaram ser mais eficientes do que as VMs para alguns aplicativos, tais como microsserviços, porque são leves e não exigem uma cópia completa do sistema operacional para cada imagem. No entanto, os contêineres ainda precisam de um *kernel* totalmente funcional que seja compartilhado entre diferentes contêineres. Além disso, o *design* de microsserviço ressalta a importância dos contêineres de estado efêmero, em que qualquer persistência de dados vai para outro armazenamento de dados ou serviço. Os contêineres efêmeros diferem de outros contêineres porque não possuem garantias de recursos ou execução e nunca serão reiniciados automaticamente. Os contêineres são considerados a maneira padrão de implantar microsserviços na Nuvem [36].

Um estudo comparativo de contêineres e virtualização é fornecido em [37], onde são investigados os conceitos de contêiner, vantagens, desvantagens e soluções. Os desafios da adoção, orquestração e segurança da tecnologia de contêiner são abordados em [38] [39] para tecnologias de contêiner na Nuvem. Além disso, a avaliação de desempenho entre tecnologias de virtualização e contêiner também são investigadas em [40] [41]. Outro trabalho de pesquisa analisa o desempenho de uma máquina virtual convencional e o contrasta com contêineres, e tira conclusões sobre qual é o ideal para usar para as necessidades desejadas [42].

Docker[5], LXC[6] e RKT[7] são exemplos de gerenciadores de contêiner. Muitos estudos se concentram no Docker porque é o ambiente de tempo de execução de contêiner predominante.

2.3.1. Docker

O Docker é a tecnologia de contêiner mais usada e popular devido aos contêineres de alto desempenho gerenciados diretamente pelo *kernel* do *host*. O Docker é uma plataforma de

[5]https://www.docker.com
[6]https://linuxcontainers.org/
[7]https://www.redhat.com/pt-br/topics/containers/what-is-rkt

conteinerização gratuita e de código aberto amplamente usada em empresas do setor e na Nuvem. É a base do Kubernetes, Google *Cloud* e é compatível com a Nuvem Amazon AWS. A plataforma Docker permite empacotar aplicativos em contêineres. Os contêineres contêm todo o código-fonte do aplicativo, bibliotecas e dependências necessárias. A plataforma Docker torna a criação, implantação, operação e atualização de contêineres mais fácil e mais segura contra quaisquer vulnerabilidades [43].

Um conceito interessante no qual o Docker se baseia é o que chamamos de *copy-on-write*. Muitos sistemas de arquivos, tais como Btrfs, *Device Mapper* e AuFS, suportam esse modelo *copy-on-write*, que é chamado de sistema de arquivos de união por desenvolvedores do *kernel*. Essencialmente, o que acontece é que você constrói camadas de sistemas de arquivos. Portanto, todo contêiner do Docker é construído sobre o que chamamos de imagem. A imagem do Docker é como um sistema de arquivos pré-configurado que contém uma camada muito fina de bibliotecas e binários necessários para fazer seu aplicativo funcionar, e talvez o código do aplicativo e alguns pacotes de suporte [44].

Os contêineres do Docker têm o Dockerfile, que contém instruções da interface de linha de comando (CLI), especifica as tarefas iniciais e cria as imagens do Docker [45]. A imagem é somente leitura e contém todo o código-fonte executável, bibliotecas e dependências para fornecer um ambiente conveniente para criar contêineres Docker. As imagens do Docker são feitas de camadas e novas camadas são criadas na parte superior se alguma alteração for feita usando um comando específico do Docker. As imagens podem ser desenvolvidas a partir do zero, bem como podem ser extraídas do Docker Hub[8]. O Docker Hub é um repositório público que contém imagens do Docker predefinidas e os contêineres do Docker são as instâncias ativas, em execução e executáveis de imagens do Docker. Usando o Docker-Compose, é possível executar vários contêineres Docker usando um arquivo YAML[9]. Este arquivo especifica quais serviços estão associados a quais contêineres e, portanto, permite gerenciar vários contêineres [45].

2.4. Função como Serviço

A Computação em Nuvem é uma tendência crescente no mercado tecnológico e proporciona um acesso onipresente, via rede e sob demanda a um conjunto de recursos computacionais configuráveis [46]. Os serviços de Computação em Nuvem são ofertados em diversos níveis de abstração, e alguns dessas principais abordagens de abstração são comparadas na Figura 2.7, onde as camadas em azul são abstraídas pelo usuário e de responsabilidade do provedor, já as camadas em amarelo são configuradas pelo usuário [47]. Nesta figura, temos:

- **Infraestrutura como Serviço**: É conhecida também como *Infrastructure as a Service* (IaaS). Esse modelo oferta recursos básicos como o armazenamento, a rede e os servidores;
- **Plataforma como Serviço**: O termo em inglês é *Platform as a Service* (PaaS). Ela oferece um conjunto de ferramentas para o desenvolvimento, operação e gerenciamento de aplicações;

[8]https://hub.docker.com/
[9]https://yaml.org/

- **Função como Serviço**: Essa abordagem permite executar códigos sem a necessidade de uma configuração complexa das ferramentas.

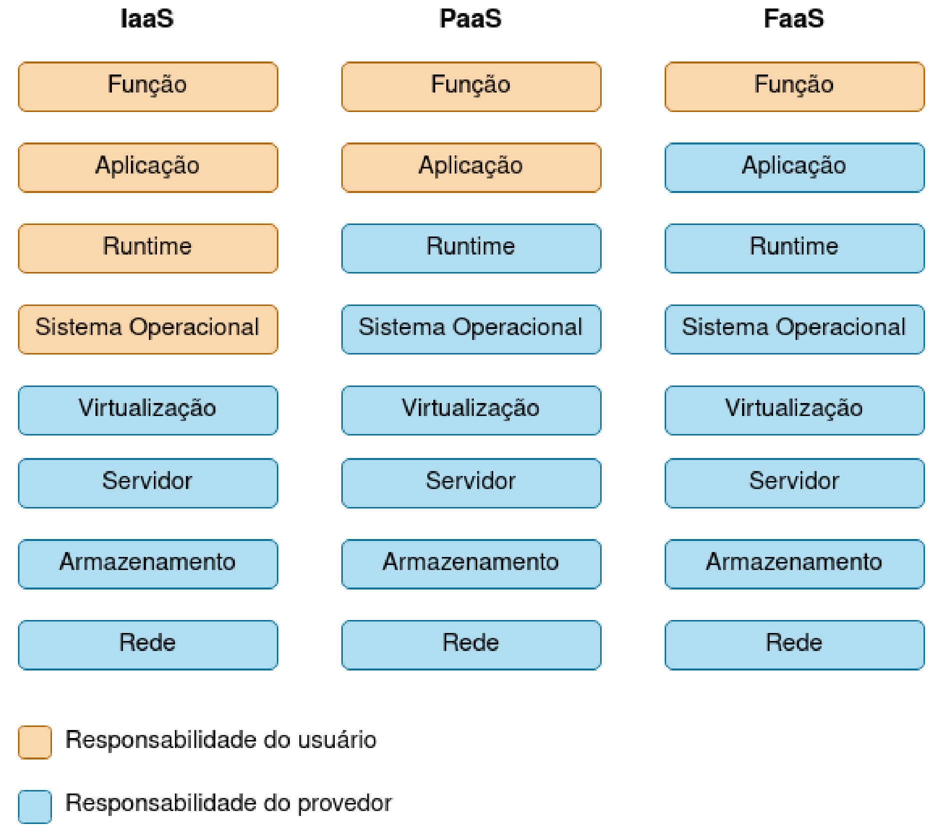


Figura 2.7: Comparação entre os serviços de Nuvem IaaS, PaaS e FaaS [Adaptada de S. K. R *et al.* [47]].

Ao longo do curso, iremos focar na FaaS, que é uma forma de computação sem servidor –– *serverless computing* —, no qual os desenvolvedores escrevem os códigos que serão executados em contêineres gerenciados por uma plataforma [48]. *Serverless* refere a diversos serviços (e.g. computação, banco de dados, mensageria) que têm a configuração, o gerenciamento e o faturamento invisíveis ao usuário final. Dentre esses serviços, a Função como Serviço se concentra em aplicações que não guardam estado e são orientadas a eventos, e os contêineres dessas aplicações só são executados quando solicitados por um evento [49].

2.4.1. Funcionamento da FaaS

Esse modelo de computação introduz uma forma diferente de cobrança por recurso em Nuvem, trazendo uma maior granularidade. Diferente das máquinas virtuais, na quais o usuário paga pelos recursos reservados mesmo quando o aplicativo não está em execução, com a FaaS o usuário só precisa pagar pelos recursos que foram utilizados durante a execução da função [50]. Então os provedores implementam o modelo iniciando uma instância da aplicação quando um certo evento ocorre e, ao final dessa execução, a instância é removida. O modelo de preço dessa abordagem depende da memória alocada às funções e do tempo de CPU para executá-las [51].

Essa natureza leve das funções, além de reduzir drasticamente o custo de implantação [52] e de manutenção, muda significativamente a atuação das equipes de desenvolvimento e operações. No desenvolvimento tradicional, essas equipes precisam gerenciar explicitamente suas máquinas virtuais. Com esse modelo *serveless*, os desenvolvedores podem se concentrar no código da aplicação, enquanto o provedor de Nuvem lida com o provisionamento, manutenção e dimensionamento da infraestrutura [53].

A FaaS pode ser útil para diversos tipos de aplicações (e.g. web, Internet das Coisas), até mesmo para produtos de Inteligência Artificial. No pipeline – fluxo de desenvolvimento e entrega – de aplicações de Aprendizado de Máquina, por exemplo, o treinamento do modelo e a inferência dos dados podem ser provisionados em funções [54]. Apesar dessa aplicabilidade, a abordagem apresenta vantagens e desvantagens que devem ser avaliadas antes de seu uso. A seguir listamos alguns desses aspectos.

2.4.2. Benefícios

Alguns dos principais pontos favoráveis ao uso de Função como Serviço são:

- **Gerenciamento simplificado**: Não há necessidade de gerenciar os servidores [48], mas apenas realizar configurações básicas como permissões, gatilhos e quantidade de CPU;
- **Maior foco no desenvolvimento**: Com níveis menor de infraestrutura para gerir, as equipes de desenvolvimento conseguem um maior tempo para se concentrarem na codificação das aplicações [50];
- **Economia nos custos de aplicações com tempo ocioso**: Em aplicações que possuem períodos de inatividade, é possível reduzir o custo de provisionamento em até 77,08% substituindo as máquinas virtuais por FaaS como mostram os autores M. Villamizar *et al.* [55];
- **Dimensionamento elástico**: As funções possuem a capacidade de dimensionar automaticamente, de forma independente e sem a necessidade de um ajuste complexo de regras, comuns em outros modelos de provisionamento [49].

2.4.3. Limitações

A Função como Serviço também possui suas desvantagens dependendo da necessidade dos usuários. Alguns desses pontos negativos são:

- **Aumento dos gastos de aplicações com carga de trabalho alta**: Em aplicações de uso frequente, sem tempo ocioso, o custo de uma FaaS pode ser três vezes maior que de uma máquina virtual [56];
- **Pouco controle**: Para regras de negócio que determinam um maior controle sobre a infraestrutura, a FaaS não é uma boa solução, pois oferece menos opções de gerenciamento de infraestrutura [53];
- **Dependência do provedor**: Cada provedor possui sua forma de configurar as funções, o que pode dificultar a migração das aplicações entre os provedores [53].

Como podemos observar, a FaaS possui muitos atrativos, mas não se aplica para todas as arquiteturas de *softwares*. Por exemplo, se uma aplicação deve estar 24 horas em execução todos os dias, esse modelo de Nuvem pode não ser aconselhável. Então antes de adotar a FaaS ou qualquer outro modelo, é interessante levantar os requisitos da aplicação, analisar a carga de trabalho, a necessidade de gerenciamento, a capacidade e o conhecimento da equipe de desenvolvimento, entre outros fatores.

2.4.4. AWS Lambda

Os principais provedores de Nuvem oferecem FaaS, tais como o AWS Lambda da Amazon[10], Cloud Functions da Google[11], Azure Functions da Microsoft [12] e IBM Cloud Functions da IBM[13]. O AWS Lamda foi o primeiro serviço de FaaS oferecido no mercado em 2014, teve uma grande adesão em 2016 [51] e continua sendo uma das principais ferramentas de FaaS.

Para realizar a parte prática deste curso, iremos utilizar a solução da Amazon. Essa ferramenta possui uma infraestrutura de alta disponibilidade e seu provedor administra os recursos computacionais, realiza a manutenção do servidor, gerencia o provisionamento e a escalabilidade automática, e monitora os *logs* (registro) das execuções das funções. O Lambda tem suporte para códigos em Java, Go, PowerShell, Node.js, C#, Python e Ruby, além possibilitar o acionamento das funções a partir de 200 serviços da própria AWS e de aplicações de Software como Serviço – *Software as a Service* (SaaS) [57].

2.4.4.1. Casos de uso

Comumente o fluxo de funcionamento do Lambda possui a estrutura apresentada na Figura 2.8. No fluxo, uma ação de um serviço (A) aciona a função, o código é executado e, ao final da execução, o resultado é enviado para outro serviço (B).

Figura 2.8: Fluxo comum de uso do Lambda [Autores].

O Lambda pode ser empregado em vários casos de uso, e o fluxo citado se encaixa em alguns casos como:

- **Processamento de dados em escala**: Pode-se criar integração com serviços da AWS, como o Amazon Simple Storage Service (Amazon S3)[14], o Amazon Dy-

[10]AWS Lambda: https://aws.amazon.com/pt/lambda/
[11]Cloud Functions: https://cloud.google.com/functions
[12]Azure Functions: https://docs.microsoft.com/en-us/azure/azure-functions/
[13]IBM Cloud Functions: https://cloud.ibm.com/functions/
[14]Serviço de armazenamento de objetos da AWS: https://aws.amazon.com/s3/

namoDB[15] e o Amazon Kinesis[16], para processamento de arquivo e de *streaming* (transmissão de dados);

- **Execução de *backend* (processo interno) de aplicações**: É possível provisionar *backend* de aplicações *web*, móveis, de IoT, entre outras.

2.4.4.2. Recursos

Para gerenciar seus serviços, a AWS proporciona o console *web*[17] e a Interface da Linha de Comando – *Command Line Interface* (CLI) – da AWS[18]. Além disso, as funções Lambda também podem ser configuradas e versionadas através de *softwares* de automatização como o Terraform[19] e o Serverless Framework[20], que possuem suporte ao provedor de Nuvem da Amazon.

Com essas ferramentas, é possível usufruir de uma série de recursos proporcionados pelo Lambda. Alguns dos principais recursos são:

- **Tolerância a falhas**: O Lambda estabelece sua computação em várias zonas de disponibilidades, que são *datacenters* (centro físico de processamento) com energia e rede em diversos locais do mundo. Isso possibilita alta disponibilidade das funções e elimina janelas de manutenção com tempo de inatividade do serviço;

- **Implantação como imagens de contêiner**: O Lambda é compatível com a implantação de funções utilizando imagens de contêiner, o que facilita a criação de aplicações com dependências não nativas do Lambda;

- **Escalabilidade automática**: Sendo um dos benefícios da FaaS, o Lambda dimensiona as funções de acordo com as solicitações recebidas. Quando a função é invocada enquanto uma instância sua está em execução, outra instância é criada para atender à solicitação. No fim dessas execuções, as instâncias inativas são congeladas ou interrompidas;

- **Controle de desempenho**: É possível habilitar a simultaneidade provisionada para manter funções já inicializadas, acarretando respostas ao evento em questão de milissegundos;

- **Orquestração de funções**: Com o AWS Step Functions[21], pode-se criar fluxo de trabalho para coordenar várias funções Lambda;

[15] Serviço de banco de dados NoSQL da AWS: https://aws.amazon.com/pt/dynamodb/

[16] Serviço de captura, processamento e armazenamento de transmissões de dados da AWS: https://aws.amazon.com/pt/kinesis/

[17] Console de gerenciamento da AWS: https://aws.amazon.com/console/

[18] AWS CLI: https://aws.amazon.com/cli/

[19] Terraform: https://www.terraform.io/

[20] Serverless Framework: https://www.serverless.com/framework

[21] Serviço de visualização de fluxos de trabalho da AWS: https://aws.amazon.com/step-functions

- **Controle de acesso**: Com o Identity and Access Management (IAM)[22], é possível controlar os usuários e as aplicações que têm acesso aos códigos e recursos das funções do Lambda;
- **URLs de função**: Pode-se atribuir um endereço *web* HTTP à função do Lambda. Com isso, é possível invocar a função por meio de navegadores ou outros clientes HTTP;
- **Modelo de funções**: Estão disponíveis no console da AWS vários esquemas de funções que apresentam um código de exemplo e configurações que demonstram a integração com outros serviços da AWS ou de terceiros;
- **Monitoramento automático**: o Lambda monitora automaticamente as funções enviando relatórios das métricas e as saídas das execuções para o Amazon CloudWatch[23].

2.4.4.3. Conceitos básicos

Para uma melhor compreensão da parte prática, apresentamos alguns termos e conceitos presentes no AWS Lambda:

- **Função**: A função é o recurso central, pois possui as configurações necessárias, dependências e o código que será invocado e executado;
- ***Trigger* (Gatilho)**: O *trigger* é o mecanismo que aciona a execução da função. Esse gatilho pode vir de diversos serviços da AWS como Amazon S3, o Amazon Simple Queue Service (Amazon SQS)[24] ou o Amazon EventBridge[25];
- **Evento**: O evento é um documento JSON que vem através do acionamento da função e é consumido pelo código. O conteúdo do evento pode apresentar informações personalizadas dependendo do serviço de origem do gatilho;
- **Arquiteturas de conjunto de instruções**: As arquiteturas determinam o tipo de processador que será utilizado para executar o código da função. A AWS oferece a arquitetura ARM de 64 bits para o processador AWS Graviton2 (`arm64`) e arquitetura x86 de 64 bits para processadores baseados em x86 (`x86_64`);
- ***Runtime* (Ambiente de execução)**: O *runtime* é um ambiente de execução isolado e específico para a linguagem do código do usuário;
- **Pacote de implantação**: É possível implantar uma aplicação na função Lambda utilizando um arquivo Zip ou uma imagem de contêiner que possua o código e suas dependências. No primeiro caso, o Lambda oferece o sistema operacional e o ambiente de execução. Já para o contêiner, esses itens devem estar presentes na imagem;

[22]Serviço de controle de acesso aos recursos da AWS: https://aws.amazon.com/iam/
[23]Serviço de monitoramento da AWS: https://aws.amazon.com/pt/cloudwatch/
[24]Serviço de mensageria da AWS: https://aws.amazon.com/pt/sqs/
[25]Barramento de eventos da AWS: https://aws.amazon.com/pt/eventbridge/

- ***Destination* (Destino)**: Ao terminar a execução da função, o Lambda pode acionar outros serviços que são configurados através do recurso de destino.

2.5. Prática

O objetivo principal desta prática é provisionar uma aplicação de AI na FaaS. Para isso, desenvolvemos usando Python um Classificador que indica se uma floresta está sofrendo incêndio ou não através de reconhecimento de imagens. O Classificador é provisionado em uma função no AWS Lambda. Para acioná-la e armazenar as imagens das florestas, utilizamos o Amazon S3. Também criamos imagens Docker que são guardadas no Amazon Elastic Container Registry (Amazon ECR)[26]. Além disso, o gerenciamento de acesso aos serviços citados acima é realizado através do IAM.

Na Figura 2.9, podemos ver o fluxo de funcionamento da aplicação. No fluxo, temos o Coletor de Imagens que envia a imagem da floresta para o *bucket* do S3. Esse envio do arquivo aciona a execução da função Lambda com o Classificador. Ao final da execução, o resultado da análise da imagem é enviado para o Serviço de Mensagens.

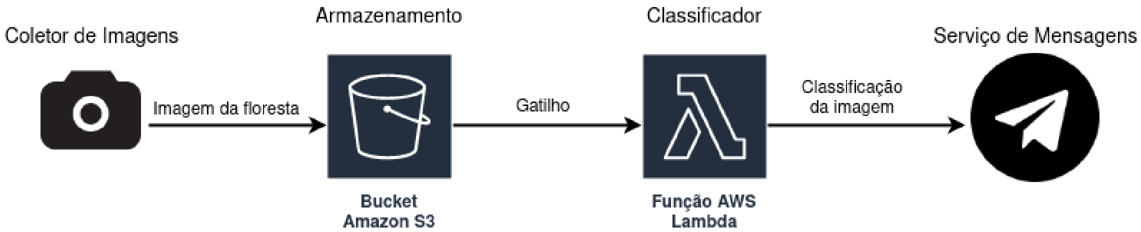


Figura 2.9: Arquitetura do serviço de classificação desta prática [Autores].

Com esse objetivo em mente, apresentamos a seguir algumas configurações básicas e trechos de códigos necessários para utilizar o AWS Lambda. Alguns trechos foram baseados na documentação da AWS[27], e todo o código desenvolvido para a prática está disponível na Internet[28].

2.5.1. Usando o AWS Lambda

Nesta seção, descrevemos o passo-a-passo para criar funções padrões em Python e com contêiner Docker no Lambda. Para isso, utilizamos o console *web* e o AWS CLI para a configurar os serviços de forma manual.

2.5.1.1. Criação de função padrão

Para criar uma função padrão no console *web*, basta ir à página funções – *Functions - Lambda*[29] – e clicar em “Create function“ (Criar função). Isso direciona o usuário para uma tela de configuração, onde deve-se digitar o nome, a linguagem e a arquitetura do

[26]Registro de contêiner da AWS: https://aws.amazon.com/ecr/
[27]AWS Lambda - Guia do desenvolvedor: https://docs.aws.amazon.com/lambda/latest/dg/welcome.html
[28]Repositório da prática online: https://github.com/sayonarasantos/wscad2022-ai-aws-lambda
[29]Functions - Lambda: https://console.aws.amazon.com/lambda/home/functions

processador da função. Ao clicar em "Create function" dessa segunda tela, é configurada uma função com código simples da linguagem escolhida e um *role* de execução que possui permissão de escrita dos resultados das execuções no Amazon CloudWatch. Na Figura 2.10, vemos a criação de uma função chamada "myFirstFunction" em Python 3.8 com arquitetura `x86_64`.

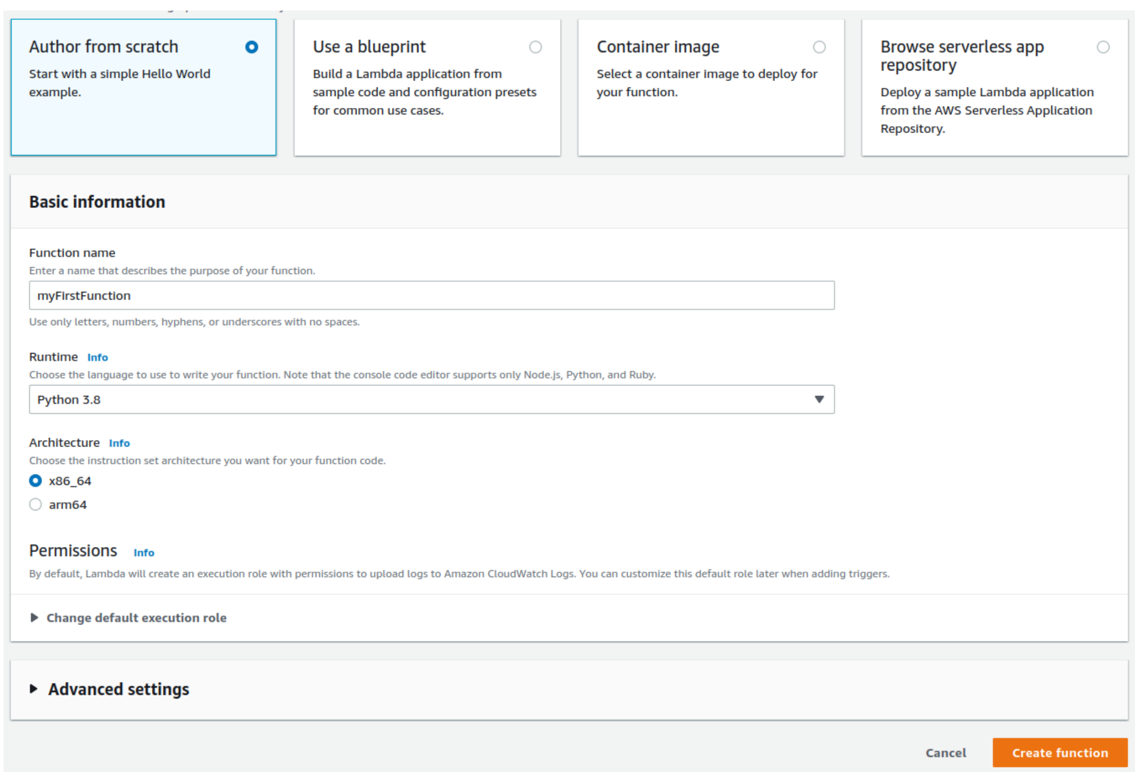


Figura 2.10: Tela de criação de uma função AWS Lambda padrão [Autores].

2.5.1.2. Teste de função

Depois de criar a função, pode-se testá-la utilizando os exemplos de eventos fornecidos pela AWS. Para isso, deve-se clicar no botão "Test" (Testar) na aba "Code" (Código) na página da função, que abrirá uma tela para configurar um evento de teste. Nessa tela, é necessário selecionar "New event" (Novo evento), digitar o nome, escolher o "Template" (Modelo) e salvar essa configuração clicando em "Save" (Salvar). A Figura 2.11 mostra a configuração de um evento de teste privado chamado "Hello" e baseado no modelo "hello-world".

Depois desse processo, basta clicar em "Test" novamente, que a função será executada recebendo o evento de teste. Após a execução, a saída será apresentada na aba "Execution result" (Resultado da execução) e no CloudWatch.

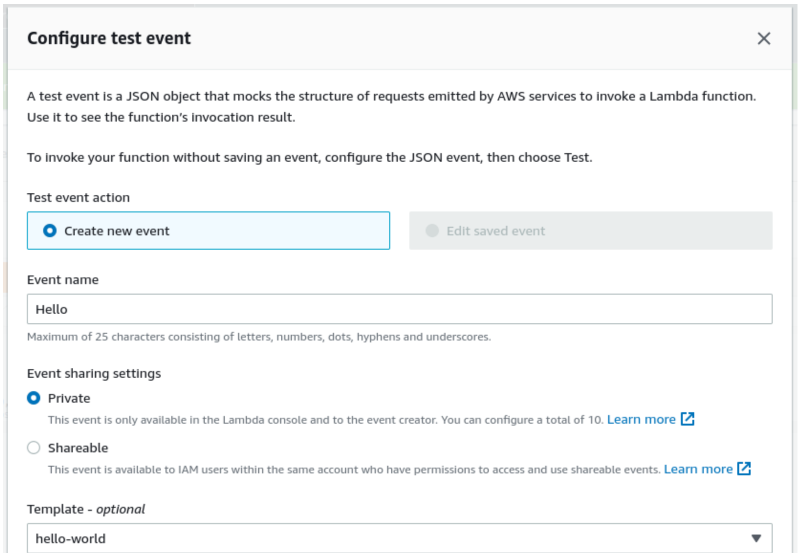


Figura 2.11: Tela de configuração do evento de teste [Autores].

2.5.1.3. Monitoramento de função

Como citamos na seção 2.4, o Lambda monitora as funções enviando métricas e *log* para o CloudWatch. Na aba "Monitor" (Monitorar) da página da função, é possível ver esses *logs* das execuções e gráficos dessas métricas (e.g. número de invocações, durações da execução, contagem de erro). Também é possível visualizar essa informações clicando no botão "View logs in CloudWatch", destacado em vermelho na Figura 2.12.

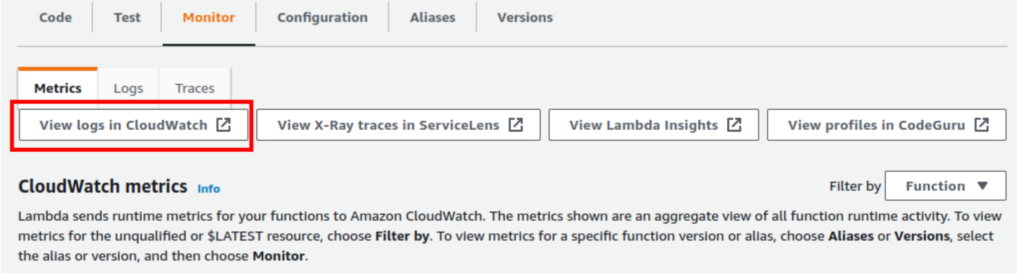


Figura 2.12: Tela de monitoramento com o botão de visualização no CloudWatch em destaque [Autores].

2.5.1.4. Gatilho S3

Um dos tipos de gatilhos disponíveis no Lambda é o acionamento da função através da movimentação dos objetos em um *bucket* do S3. Antes de adicionar esse gatilho, precisa-se criar um *bucket*. Para fazê-lo pelo console *web*, deve acessar a página inicial do S3[30] e clicar em "Create bucket" (Criar *bucket*), que direciona o usuário à tela de criação. Nessa tela, é necessário definir um nome exclusivo, escolher a região, o "Object Ownership" – a

[30] S3 *home*: https://console.aws.amazon.com/s3/

propriedade dos objetos que serão enviados – e os "Bucket settings for Block Public Access" (Configurações de *bucket* para o Bloqueio de Acesso Público), e clicar em "Create bucket" (Criar bucket).

Com o *bucket* criado, pode-se adicionar o gatilho na página da função, clicando em "Add trigger" (Adicionar gatilho) e selecionando o serviço S3 como a origem do acionamento. Na tela de configuração do gatilho S3, deve-se determinar o *bucket* e o tipo de evento que acionará a função. Também é possível definir um prefixo ou um sufixo dos objetos que participarão do acionamento. Na Figura 2.13, temos um exemplo de criação de um gatilho que invocará a função sempre que um objeto com o nome terminado em ".jpg" (*Suffix .jpg*) for adicionado (*Event type POST*) ao *bucket* "2022-lambda-course".

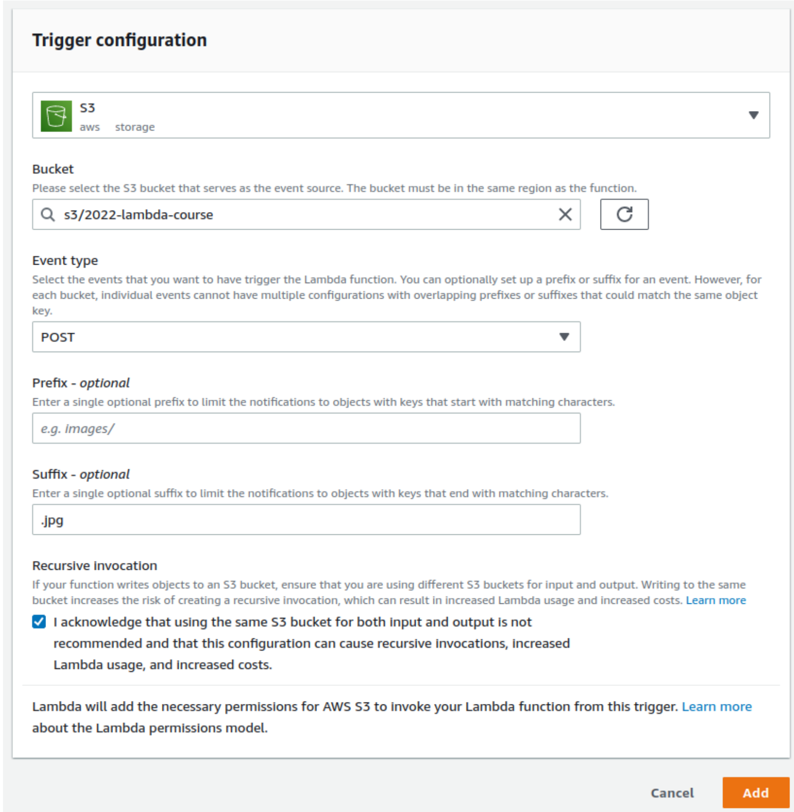


Figura 2.13: Tela de configuração de gatilho S3 [Autores].

Caso a função precise acessar os objetos do *bucket*, deve-se anexar uma política que conceda as permissões desejadas ao *role* da função. Na aba "Configuration" (Configuração) da página da função, o *role* está definido em "Permission" (Permissões) na seção "Execution role" (Função de execução). Para configurar suas permissões, basta clicar na identificação do *role*, que o usuário será redirecionado à página do *role* no IAM. Nessa tela, é possível anexar políticas feitas pela AWS ou customizadas pelo usuário.

2.5.1.5. Criação de função com imagem de contêiner Docker

Outra forma de criar uma função Lambda é utilizando imagem de contêiner. Esse tipo de abordagem é interessante para instalar dependências da aplicação, principalmente pacotes

que não são compatíveis com os formatos suportados pelo Lambda como pacotes Linux. Esse tipo de função carrega uma imagem de contêiner presente no Amazon ECR. A seguir, descrevemos um conjunto de etapas necessárias para configurar um Lambda com contêiner de forma manual.

- **Criação de repositório**: Para criar o repositório dessa imagem, é necessário navegar à página de repositório[31], clicar em "Create repository" (Criar repositório), selecionar a visibilidade em "Visibility settings" (Configurações de visibilidade), digitar o "Repository name" (Nome do repositório), determinar se as *tags* são imutáveis em "Tag immutability" (Imutabilidade de tag), escolher se haverá criptografia em "KMS encryption" (Criptografia KMS) e clicar "Create repository" (Criar repositório).
- **Criação do usuário do IAM**: Para criar a imagem Docker e enviá-la de forma manual, deve-se ter o Docker Engine e o AWS CLI instalados em seu computador e um usuário do IAM com permissões de escrita no ECR. Esse usuário pode ser criado na página inicial do IAM[32], clicando em "Users" e depois em "Add users". Na primeira tela de configuração, precisa-se digitar o nome de identificação do usuário "User name" (Nome do usuário), selecionar o tipo de credencial útil para usar com o AWS CLI – o "Access key - Programmatic access" (Chave de acesso - Acesso programático) – e clicar em "Next" (Próximo). A segunda tela é para anexar uma política que permita ao usuário escrever no ECR, como a política fornecida pela AWS "AmazonEC2ContainerRegistryFullAccess", que fornece acesso administrativo do ECR. Para finalizar, basta clicar em "Next" e "Create user" (Criar usuário) e baixar o arquivo CSV ou copiar o "Access key ID" (Código da chave de acesso) e o "Secret access key" (Chave de acesso secreta).
- **Instalação do Docker e do AWS CLI**: As instalações do Docker e AWS CLI irão depender do sistema operacional utilizado. Portanto, é recomendável seguir a documentação oficial de instalação do Docker Engine[33], instalação do AWS CLI[34] e guia de uso do ECR com AWS CLI[35]. No repositório *online* deste curso, descrevemos o passo-a-passo e os comandos de configuração que utilizamos no sistema operacional Ubuntu.
- **Configuração da imagem**: Com as configurações acima feitas, a próxima etapa é criar um Dockerfile. Nos Códigos 2.1 e 2.2, apresentamos um código Python simples e um Dockerfile que cumprem os requisitos necessários para executar no Lambda, como: (I) o código implementa a interface de execução do Lambda e (II) a imagem é baseada em um sistema Linux. Nesse exemplo, utilizamos uma imagem base Linux (a distribuição Debian bullseye) e implementamos a interface de execução do Lambda em Python (o pacote awslambdaric).

[31] Repositories - ECR: https://console.aws.amazon.com/ecr/repositories

[32] IAM dashboard: https://console.aws.amazon.com/iam/

[33] Instalação do Docker Engine: https://docs.docker.com/engine/install/

[34] Instalar ou atualizar o AWS CLI: https://docs.aws.amazon.com/cli/latest/userguide/getting-started-install.html

[35] Uso do Amazon ECR com o AWS CLI: https://docs.aws.amazon.com/AmazonECR/latest/userguide/getting-started-cli.html

Código 2.1: Código Python simples

```
1 import json
2
3
4 def lambda_handler(event, context):
5     print("Received event: " + json.dumps(event, indent=2))
```

Código 2.2: Dockerfile simples para código Python

```
1  FROM python:3.8-slim-bullseye
2
3  WORKDIR /project
4
5  RUN apt-get update && \
6      apt-get install -y \
7      g++ \
8      make \
9      cmake \
10     unzip \
11     libcurl4-openssl-dev
12
13 RUN pip install --no-cache-dir awslambdaric
14
15 COPY app.py .
16
17 CMD [ "python", "-m", "awslambdaric", "app.lambda_handler" ]
```

- **Envio da imagem**: Com os códigos escritos, pode-se construir a imagem e enviá-la para o ECR utilizando os comandos:

```
# Construir imagem
$ docker build -t simple-example:v1 .

# Login no Docker
$ aws ecr get-login-password --region REGION | docker login
    --username AWS --password-stdin ACCOUNT_ID.dkr.ecr.us-
    east-1.amazonaws.com

# Marcar imagem para corresponder ao nome no ECR
$ docker tag simple-example:v1 ACCOUNT_ID.dkr.ecr.REGION.
    amazonaws.com/simple-example:v1

# Enviar a imagem
$ docker push ACCOUNT_ID.dkr.ecr.us-east-1.amazonaws.com/
    simple-example:v1
```

- **Configuração da função**: A criação da função pelo console *web* deve ser feita na página das funções do Lambda, clicando em “Create function”. Na tela de configuração que será aberta, deve-se escolher o tipo de função “Container image”

(Imagem de contêiner), digitar o nome, escolher a arquitetura da imagem e clicar em "Create function". A Figura 2.14 apresenta a criação da função "containerSimpleExample" baseada em imagem de contêiner "simples-example" com arquitetura `x86_64`.

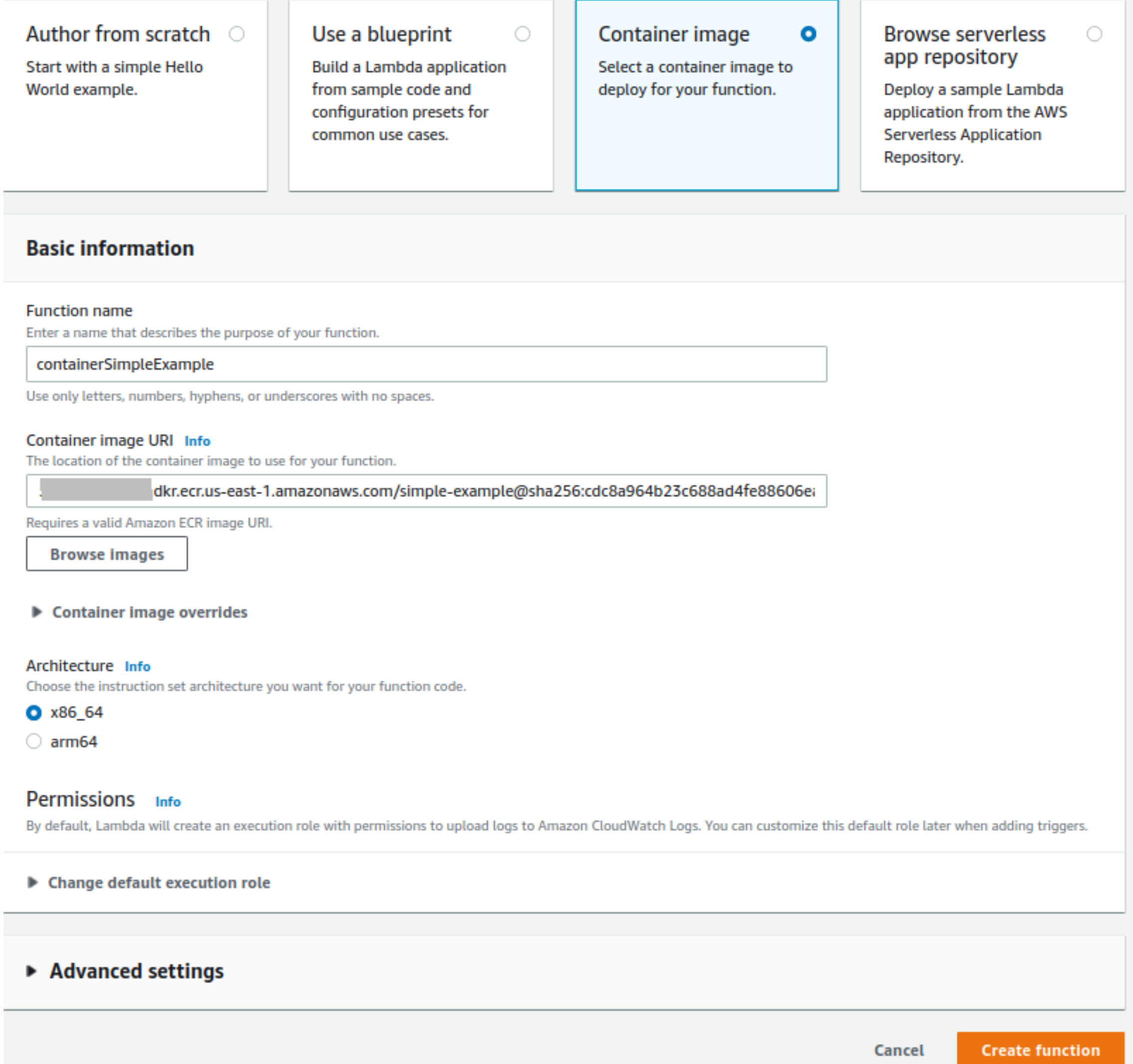


Figura 2.14: Tela de criação de uma função AWS Lambda baseado em imagem de contêiner [Autores].

2.5.2. Criação do código de implantação do modelo de ML

Como dito anteriormente, desenvolvemos uma solução de AI capaz de analisar uma imagem e identificar se ela mostra um incêndio florestal ou não. Para isso, utilizamos o *framework* Tensorflow (abordado na Subseção 2.2.4) e o modelo pré-treinado para classificação de imagens MobileNet [58], para criar uma solução de DL.

Para treinar o modelo utilizamos um conjunto de imagens para detecção de incêndios florestais [59]. Esse *dataset* contém 1900 imagens, sendo 950 imagens de florestas sem incêndio (que identificamos com o rótulo 0) e 950 imagens de florestas com incêndio

(cujo rótulo aplicado foi 1). A Figura 2.15 mostra um exemplo de imagens utilizadas no treinamento.

(a) Floresta normal.

(b) Floresta com incêndio.

Figura 2.15: Exemplo de imagens encontradas no *dataset* [Autores].

Salvamos o modelo, após ele ter sido treinado e validado, em um arquivo do tipo Formato de Dados Hierárquicos – *Hierarchical Data Format* (HDF)[36] – (formato de arquivo utilizado para armazenar grandes volumes de dados). Esse arquivo será utilizado para carregar o modelo treinado e realizar predições.

O Código 2.3 é uma função Lambda que, quando acionada, carrega o modelo de DL treinado, realiza a classificação de uma imagem e dá como saída sua classificação. Da linha 1 até a linha 6 importamos as bibliotecas utilizadas no programa que nos permitem manipular a imagem analisada, bem como interagir com o Tensorflow e a AWS. Na linha 8 inicializamos uma variável chamada `LABELS` utilizada para traduzir a saída do modelo, e na linha 13 instanciamos a variável `s3`, um objeto do tipo `bot3.client()`, utilizada para interagir com os recursos da AWS.

Código 2.3: Código Lambda utilizado na predição das imagens.

```
1  import boto3
2  import tensorflow as tf
3
4  from io import BytesIO
5  import numpy as np
6  from PIL import Image
7
8  LABELS = {
9      0: "With Fire",
10     1: 'Without Fire'
11 }
12
13 s3 = boto3.client("s3")
```

[36]Gravação de modelos no formato HDF5: https://www.tensorflow.org/guide/keras/save_and_serialize.

```
14
15
16 def lambda_handler(event, context):
17
18     model = tf.keras.models.load_model("./model.h5")
19
20     bucket_name = event["Records"][0]["s3"]["bucket"]["name"]
21     key = event["Records"][0]["s3"]["object"]["key"]
22
23     file_byte_string =\
24         s3.get_object(
25             Bucket=bucket_name, Key=key)["Body"].read()
26
27     image = Image.open(BytesIO(file_byte_string))
28     image = image.resize((224, 224))
29
30     image = image - np.mean(image)
31
32     input_tensor =\
33         np.array(
34             np.expand_dims(image, axis=0), dtype=np.float32)
35
36     output_data = model.predict(x=input_tensor, verbose=0)
37
38     output_data = np.squeeze(output_data)
39     result = np.argmax(output_data, axis=-1)
40
41     print(f"Prediction: {LABELS[result]}")
42     print(f"Probability: {output_data[result]:.2f}")
```

A função Lambda, definida na linha 16, recebe como parâmetro as variáveis `event` (que contém informações úteis para a função processar) e `context` (que traz informações sobre o contexto onde a função está sendo executada). Na linha 18 carregamos o modelo treinado utilizando a função do Tensorflow `load_module()` que recebe como parâmetro o endereço para o arquivo do modelo salvo. Nas linhas 20 e 21 obtemos o nome do *bucket* onde a imagem analisada foi salva e a chave de segurança utilizada para recuperar essa imagem, respectivamente.

Na linha 23 utilizamos a função `get_object()` do objeto instanciado `s3` para recuperar a imagem que será analisada. Para isso, precisamos passar o nome do *bucket* onde a imagem foi salva e a chave de acesso. Ambas informações foram recuperadas nas linhas 20 e 21.

Da linha 27 a 30 realizamos um pré-processamento na imagem. Isso é necessário, pois o modelo espera receber uma imagem que possua um tamanho de 224×224 *pixels* e cujo valor médio dos *pixels* tenha sido subtraído de cada canal da imagem. A Figura 2.16 mostra um exemplo de como as imagens ficam após o pré-processamento.

Na linha 32 inicializamos a variável `input_tensor` com a imagem pré-processada.

(a) Floresta normal.

(b) Floresta com incêndio.

Figura 2.16: Exemplo de imagens pré-processadas [Autores].

Essa variável será passada ao modelo, na linha 36, para que a predição seja. Nas linhas 38 e 39 utilizamos as funções `squeeze()` (que reduz a dimensionalidade de um *array*) e `argmax()` (que retorna o índice do maior valor no *array*) da biblioteca *numpy* para preparar o resultado dado pelo modelo. E por fim, nas linhas 41 e 42 imprimimos na tela o rótulo predito pelo modelo para a imagem e a probabilidade de certeza em relação ao resultado, respectivamente.

2.5.3. Provisionamento da aplicação de ML no AWS Lambda

Para provisionar a aplicação ML de classificação descrita na seção anterior, escrevemos o Dockerfile presente no Código 2.4. A imagem base "python:3.8-slim-bullseye" vem do repositório Docker Hub, e seu sistema é um Debian bullseye. Da linha 5 a 11 é feita a instalação das dependências Linux necessárias para compilar a interface de execução do AWS Lambda. Nas linhas 13 e 15 utiliza-se o arquivo "requiments.txt" para instalar os pacotes Python necessários. Nas linhas 17 e 19 copia-se o código Python e o modelo de ML. Por fim, na última linha declara-se o comando que manterá a execução do contêiner da função Lambda.

Código 2.4: Dockerfile para o Classificador

```
1  FROM python:3.8-slim-bullseye
2
3  WORKDIR /project
4
5  RUN apt-get update && \
6    apt-get install -y \
7    g++ \
8    make \
9    cmake \
10   unzip \
11   libcurl4-openssl-dev
12
```

```
13 COPY requirements.txt .
14
15 RUN pip install --no-cache-dir -r requirements.txt
16
17 COPY model.tflite .
18
19 COPY app.py .
20
21 CMD [ "python", "-m", "awslambdaric", "app.lambda_handler" ]
```

2.6. Considerações finais

A proposta deste minicurso foi apresentar uma introdução sobre o provisionamento de aplicações de Inteligência Artificial na Nuvem através de Funções como Serviços, que é uma abordagem atual que introduz a capacidade de se executar tarefas sem um servidor. Para esse tipo de tarefa, utilizamos como caso de uso a implantação de uma aplicação de Aprendizado de Máquina no AWS Lambda.

Após uma breve introdução sobre o tema, abordamos AI junto com Redes Profundas e Redes Convolucionais. Em seguida, falamos sobre contêineres, dando um foco na ferramenta Docker, utilizada para empacotar a aplicação executada no Lambda. Depois descrevemos os conceitos da FaaS, além de apresentar o AWS Lambda. Por fim, demonstramos a base técnica para realização da prática do minicurso.

A prática envolve a implantação de um modelo de Aprendizado Profundo capaz de identificar se está ocorrendo ou não um incêndio florestal, através da classificação de uma imagem na nuvem usando o AWS Lambda. Nessa prática guiamos os alunos por todo o processo de configuração e implantação do modelo de IA. Importante destacar que nossa metodologia é genérica e pode ser aplicada em outras aplicações.

Diante do conteúdo exposto, podemos concluir que a FaaS pode ser empregada no *pipeline* de soluções de AI. Contudo, devido às limitações, esse modelo de serviço de Nuvem não será sempre a melhor alternativa para o provisionamento de aplicações. Para determinar se o modelo é útil para aplicação, é necessário levar em consideração alguns pontos como a carga de trabalho e o nível de controle exigido pela empresa contratante.

Referências

[1] K. K. Mohbey and S. Kumar, "The impact of big data in predictive analytics towards technological development in cloud computing," *International Journal of Engineering Systems Modelling and Simulation*, vol. 13, no. 1, pp. 61–75, 2022.

[2] Oracle, "O que é inteligência artificial (ia)? saiba mais sobre inteligência artificial." https://www.oracle.com/br/artificial-intelligence/what-is-ai/, 2022. Online; acessado em Ago 2022.

[3] P. Linardatos, V. Papastefanopoulos, and S. Kotsiantis, "Explainable ai: A review of machine learning interpretability methods," *Entropy*, vol. 23, no. 1, 2021.

[4] B. P. Rimal, E. Choi, and I. Lumb, "A taxonomy and survey of cloud computing systems," in *2009 Fifth International Joint Conference on INC, IMS and IDC*, pp. 44–51, Ieee, 2009.

[5] V. Ishakian, V. Muthusamy, and A. Slominski, "Serving deep learning models in a serverless platform," in *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pp. 257–262, 2018.

[6] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, *et al.*, "Serverless computing: Current trends and open problems," in *Research advances in cloud computing*, pp. 1–20, Springer, 2017.

[7] T. Elgamal, "Costless: Optimizing cost of serverless computing through function fusion and placement," in *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pp. 300–312, 2018.

[8] R. Bala, B. Gill, D. Smith, K. Ji, and D. Wright, "Quadrante mágico para infraestrutura em nuvem e serviços de plataforma." https://www.gartner.com/technology/media-products/reprints/AWS/1-271W1OT3-PTB.html, 2021. Online; acessado em Jun 2022.

[9] V. Rajaraman, "Johnmccarthy—father of artificial intelligence," *Resonance*, vol. 19, no. 3, pp. 198–207, 2014.

[10] D. Crevier, *AI: the tumultuous history of the search for artificial intelligence*. Basic Books, Inc., 1993.

[11] R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, *Machine learning: An artificial intelligence approach*. Springer Science & Business Media, 2013.

[12] A. Shrikumar, P. Greenside, and A. Kundaje, "Learning important features through propagating activation differences," in *International conference on machine learning*, pp. 3145–3153, PMLR, 2017.

[13] A. Krenker, J. Bešter, and A. Kos, "Introduction to the artificial neural networks," *Artificial Neural Networks: Methodological Advances and Biomedical Applications. InTech*, pp. 1–18, 2011.

[14] P. Joshi, *Artificial intelligence with python*. Packt Publishing Ltd, 2017.

[15] A. Botta, W. De Donato, V. Persico, and A. Pescapé, "Integration of cloud computing and internet of things: a survey," *Future generation computer systems*, vol. 56, pp. 684–700, 2016.

[16] S. Sagiroglu and D. Sinanc, "Big data: A review," in *2013 international conference on collaboration technologies and systems (CTS)*, pp. 42–47, IEEE, 2013.

[17] R. Alshamrani and X. Ma, *Deep Learning*, pp. 1–5. Cham: Springer International Publishing, 2019.

[18] M. Z. Alom, T. Taha, C. Yakopcic, S. Westberg, P. Sidike, M. Nasrin, M. Hasan, B. Essen, A. Awwal, and V. Asari, “A state-of-the-art survey on deep learning theory and architectures,” *Electronics*, vol. 8, p. 292, 03 2019.

[19] V. H. Phung and E. J. Rhee, “A high-accuracy model average ensemble of convolutional neural networks for classification of cloud image patches on small datasets,” *Applied Sciences*, vol. 9, no. 21, 2019.

[20] R. Alshamrani and X. Ma, “Deep learning,” *por Laurie A. Schintler y Connie L. McNeely. Cham: Springer International Publishing*, pp. 1–5, 2019.

[21] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” *arXiv preprint arXiv:1409.1556*, 2014.

[22] TensorFlow. URL https://www.tensorflow.org/?hl=pt-br. Acessado: 14/08/2022.

[23] PyTorch. URL https://pytorch.org/. Acessado: 14/08/2022.

[24] Keras. URL https://keras.io/. Acessado: 14/08/2022.

[25] J. Zenisek, F. Holzinger, and M. Affenzeller, “Machine learning based concept drift detection for predictive maintenance,” *Computers & Industrial Engineering*, vol. 137, p. 106031, 2019.

[26] S. Alla and S. K. Adari, “What is mlops?,” in *Beginning MLOps with MLFlow*, pp. 79–124, Springer, 2021.

[27] S. Mäkinen, H. Skogström, E. Laaksonen, and T. Mikkonen, “Who needs mlops: What data scientists seek to accomplish and how can mlops help?,” in *2021 IEEE/ACM 1st Workshop on AI Engineering - Software Engineering for AI (WAIN)*, pp. 109–112, 2021.

[28] Q. Zhang, L. Liu, C. Pu, Q. Dou, L. Wu, and W. Zhou, “A comparative study of containers and virtual machines in big data environment,” in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pp. 178–185, IEEE, 2018.

[29] Docker, “What is a container.” https://www.docker.com/resources/what-container/, 2021. Online; acessado em Ago 2022.

[30] Namespaces-Overview, “Namespaces-overview of linux namespaces oct..” https://man7:org/linux/man-pages/man7/namespaces:7:html, 2021. Online; acessado em Ago 2022.

[31] C. Namespaces-Overview, “Cgroup namespaces-overview of linux cgroup namespaces oct..” https://www:man7:org/linux/manpages/man7/cgroup_namespaces:7:html, 2021. Online; acessado em Ago 2022.

[32] A. Lingayat, R. R. Badre, and A. K. Gupta, “Integration of linux containers in openstack: An introspection,” *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 12, no. 3, pp. 1094–1105, 2018.

[33] L. Benedicic, F. A. Cruz, A. Madonna, and K. Mariotti, “Sarus: Highly scalable docker containers for hpc systems,” in *International Conference on High Performance Computing*, pp. 46–60, Springer, 2019.

[34] K. Kaur, T. Dhand, N. Kumar, and S. Zeadally, “Container-as-a-service at the edge: Trade-off between energy efficiency and service availability at fog nano data centers,” *IEEE wireless communications*, vol. 24, no. 3, pp. 48–56, 2017.

[35] B. M. Abbott, *A security evaluation methodology for container images*. Brigham Young University, 2017.

[36] S. Vaucher, R. Pires, P. Felber, M. Pasin, V. Schiavoni, and C. Fetzer, “Sgx-aware container orchestration for heterogeneous clusters,” in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pp. 730–741, IEEE, 2018.

[37] R. Dua, A. R. Raja, and D. Kakadia, “Virtualization vs containerization to support paas,” in *2014 IEEE International Conference on Cloud Engineering*, pp. 610–614, IEEE, 2014.

[38] B. Bermejo, C. Juiz, and C. Guerrero, “Virtualization and consolidation: a systematic review of the past 10 years of research on energy and performance,” *The Journal of Supercomputing*, vol. 75, no. 2, pp. 808–836, 2019.

[39] Z. Li, M. Kihl, Q. Lu, and J. A. Andersson, “Performance overhead comparison between hypervisor and container based virtualization,” in *2017 IEEE 31st International Conference on advanced information networking and applications (AINA)*, pp. 955–962, IEEE, 2017.

[40] C. Pahl, “Containerization and the paas cloud,” *IEEE Cloud Computing*, vol. 2, no. 3, pp. 24–31, 2015.

[41] S. A. Babu, M. Hareesh, J. P. Martin, S. Cherian, and Y. Sastri, “System performance evaluation of para virtualization, container virtualization, and full virtualization using xen, openvz, and xenserver,” in *2014 fourth international conference on advances in computing and communications*, pp. 247–250, IEEE, 2014.

[42] P. R. Desai, “A survey of performance comparison between virtual machines and containers,” *Int. J. Comput. Sci. Eng*, vol. 4, no. 7, pp. 55–59, 2016.

[43] P. Liu, S. Ji, L. Fu, K. Lu, X. Zhang, W.-H. Lee, T. Lu, W. Chen, and R. Beyah, “Understanding the security risks of docker hub,” in *European Symposium on Research in Computer Security*, pp. 257–276, Springer, 2020.

[44] C. Anderson, “Docker [software engineering],” *IEEE Software*, vol. 32, no. 3, pp. 102–c3, 2015.

[45] Docker, “Dockerfile reference.” https://docs.docker.com/engine/reference/builder/, 2021. Online; acessado em Ago 2022.

[46] P. Mell, T. Grance, *et al.*, “The nist definition of cloud computing,” 2011.

[47] S. K. R and J. Lakshmi, "Qos aware faas platform," in *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pp. 812–819, 2021.

[48] I. Red Hat, "Cloud computing." https://www.redhat.com/pt-br/topics/cloud, 2018. Online; acessado em Ago 2022.

[49] I. C. Education, "Faas (function-as-a-service)." https://www.ibm.com/cloud/learn/faas, 2022. Online; acessado em Ago 2022.

[50] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu, "Serverless computing: One step forward, two steps back," 2018.

[51] T. Elgamal, A. Sandur, K. Nahrstedt, and G. Agha, "Costless: Optimizing cost of serverless computing through function fusion and placement," in *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pp. 300–312, 2018.

[52] T. Lynn, P. Rosati, A. Lejeune, and V. Emeakaroha, "A preliminary review of enterprise serverless cloud computing (function-as-a-service) platforms," in *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 162–169, 2017.

[53] B. King, "What is faas? function as a service explained." https://www.digitalocean.com/blog/what-is-faas-function-as-a-service-explained, 2022. Online; acessado em Ago 2022.

[54] M. Chadha, A. Jindal, and M. Gerndt, "Towards federated learning using faas fabric," in *Proceedings of the 2020 Sixth International Workshop on Serverless Computing*, WoSC'20, (New York, NY, USA), p. 49–54, Association for Computing Machinery, 2020.

[55] M. Villamizar, O. Garcés, L. Ochoa, H. Castro, L. Salamanca, M. Verano Merino, R. Casallas, S. Gil, C. Valencia, A. Zambrano, and M. Lang, "Cost comparison of running web applications in the cloud using monolithic, microservice, and aws lambda architectures," *Service Oriented Computing and Applications*, vol. 11, 06 2017.

[56] A. Eivy and J. Weinman, "Be wary of the economics of "serverless"cloud computing," *IEEE Cloud Computing*, vol. 4, no. 2, pp. 6–12, 2017.

[57] Amazon, "Aws lambda: Execute código sem se preocupar com servidores ou clusters." https://aws.amazon.com/lambda/, 2022. Online; acessado em Ago 2022.

[58] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[59] A. Khan and B. Hassan, "Dataset for forest fire detection," Mendeley Data, V1, 2020. doi: 10.17632/gjmr63rz2r.1.

Capítulo 3

Coisas para Fazer Antes de Paralelizar [1]

Alfredo Goldman (USP), Vitor Tessari Terra (USP) e Sarita Mazzini Bruschi (USP)

Abstract

In this short course, we will look at some practical things that should be done to improve code performance before thinking about parallelization. For example, on a 32-core machine, the potential gain of speeding up a sequential program with threads can be close to 32 times. On the other hand, by using a proper language, having good data structures and using the cache in the right way, the gains can be much greater. To this end, this short course reviews essential concepts for program evaluation and comparison, including examples and live demonstrations of how to apply them. Divided into two parts, the first part covers general aspects while the second part focuses on the individual analysis of a program. Part of the course was inspired by the Computer Language Benchmarks Game.

Resumo

Nesse minicurso, veremos algumas coisas práticas que devem ser feitas para melhorar o desempenho de código antes de se pensar em paralelização. Por exemplo, em uma máquina com 32 núcleos, o ganho potencial de aceleração de um programa sequencial com threads pode ser próximo a 32 vezes. Por outro lado, ao usar uma linguagem adequada, ter boas estruturas de dados e usar o cache da forma correta, os ganhos podem ser bem maiores. Para isso, este minicurso revisa conceitos essenciais para a avaliação e comparação de programas, incluindo exemplos e demonstrações ao vivo de como aplicá-los. Dividido em duas partes, a primeira parte aborda os aspectos gerais enquanto a segunda foca na análise individual de um programa. Parte do curso foi inspirada no Computer Language Benchmarks Game.

[1] Esse minicurso teve apoio do processo nº 2019/26702-8, Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP)

3.1. Introdução

Uma das formas de se melhorar o desempenho de aplicações é por meio da paralelização, que é muito usada em computação de alto desempenho (do inglês, *HPC - High Performance Computing*). No entanto, antes de se pensar em como fazer com que partes de um programa possam ser executadas de forma paralela, é essencial saber avaliar o seu desempenho.

Várias técnicas estão disponíveis para paralelizar um código, tais como paralelização manual, paralelização automática na compilação, paralelização de laços, uso de bibliotecas com funções já paralelizadas, uso de programação baseada em diretivas, etc. Dependendo da escolha, o esforço necessário para paralelizar um código é alto. Partindo do pressuposto que já existe um código sequencial, temos como objetivo entender melhor como avaliar e comparar diferentes variações de um mesmo pseudo-código antes de decidir paralelizar o código.

O texto desse minicurso é uma versão atualizada do minicurso "Coisas para saber antes de fazer o seu próprio *Benchmarks Game*", apresentado na Escola Regional de Alto Desempenho - Região Sul, 2022 [Goldman et al. 2022, Capítulo 4]. Inspirado no *Computer Language Benchmarks Game* [2], este minicurso revisa conceitos essenciais para a avaliação e comparação de programas, incluindo exemplos e demonstrações ao vivo de como aplicá-los. Dividido em duas partes, a primeira parte aborda os aspectos gerais, enquanto a segunda foca na análise individual de um programa.

Na primeira parte utilizaremos um conjunto de programas que implementam um mesmo algoritmo (pseudo-código) em linguagens diferentes. Em seguida, será construído um `Makefile` para comparar o desempenho dos programas. Vamos:

- apresentar algumas métricas para avaliação de um programa (tempo, espaço, impacto energético, memória, ...);
- detalhar do que é composto o tempo de execução (tempo real, em espaço de usuário e espaço do sistema operacional);
- mostrar formas de tabular os dados coletados, meios de avaliar, interpretar e visualizar as medições;
- e por fim discutir fatores básicos que afetam o desempenho com base nas medições (linguagem de programação, sistema operacional, hardware e estado de carga do sistema).

Na segunda parte vamos trabalhar nos programas de forma individual para explicar tópicos de análise de desempenho usando o `Makefile` desenvolvido na parte anterior. Vamos abordar:

- os diferentes tipos de relógios que um sistema pode oferecer (incluindo relógios disponibilizados pela arquitetura) usando C;

[2] `https://benchmarksgame-team.pages.debian.net/benchmarksgame/`

- o impacto de *flags* no tempo de execução e compilação usando o GCC;
- uso de compiladores/interpretadores alternativos e considerações sobre compilação JIT usando o PyPy;
- *profiling* usando a versão em Go;
- verificação de saída e soma de verificação usando Go.

Os códigos apresentados no texto estão disponíveis em `https://github.com/saritabruschi/Minicurso-WSCAD2022`.

3.2. O caso de uma maratona de programação paralela

Um caso que ilustra a necessidade de analisar outros fatores que alterem o desempenho antes de se paralelizar ocorreu na Maratona de Programação Paralela da ERAD-SP [3] 2022. A competição consistia em dois problemas, descritos de forma simplificada a seguir:

- Problema A: a entrada consiste em um texto codificado em ASCII (palavras separadas por espaços e texto terminado por `"\n"`) e um conjunto de palavras a serem removidas; a saída deve ser o texto sem as palavras removidas. Por exemplo:

  Texto: `"0650660650326606508203206706809 0010"`

  Texto decodificado: `"ABA BAR DEZ\n"`

  Palavras a remover: `"BOA", "BAR"`

  Saída: `"ABA DEZ\n"`

- Problema B: P números primos são divididos em duas partes cada – por exemplo, o número 504155039 pode ser dividido de diversas formas, tais como: 5041 e 55039, ou 504155 e 039. A entrada consiste em uma lista de $2P$ "partes" de números primos, sendo as P primeiras correspondentes à "parte esquerda" de um número primo e as P demais correspondentes à "parte direita"; a saída deve ser a lista dos possíveis números primos formados a partir dessas "partes", em ordem crescente. Por exemplo:

  Entrada: `"8 10 11 03 27 889"`

  Saída: `"827 1103 10889"`

Junto com o enunciado dos problemas, foram fornecidas soluções sequenciais como referência. O time vencedor seria aquele que obtivesse o maior *speedup*, ou seja, a maior redução de tempo de execução em relação à versão sequencial.

Para o Problema A, uma possível estratégia de paralelização seria dividir o texto em partes de tamanho próximo (sem quebrar palavras pela metade), de modo que cada *thread* seja responsável por remover as palavras de uma parte do texto. Já para o Problema B, há P^2 candidatos a números primos a serem formados – cada *thread* pode ser

[3] Escola Regional de Alto Desempenho de São Paulo

responsável por testar uma parte desses candidatos e devolver aqueles que forem, de fato, primos.

Surpreendentemente, as melhores soluções para os problemas A e B não aplicaram nenhuma estratégia de paralelização. Dispondo apenas de melhorias para as versões sequenciais, o time vencedor chegou a obter um *speedup* superior a 3000 para o Problema B! As otimizações adotadas foram as seguintes:

- Problema A: as palavras a serem removidas foram armazenadas em uma estrutura de dados mais eficiente para consultas (*trie* [4]). Além disso, a solução foi compilada com uma *flag* de otimização mais agressiva do que a versão inicial: `-Ofast` no lugar de `-O3`. Nesse caso, algumas garantias de corretude para a execução do programa podem ser perdidas [5], embora não tenha afetado a solução deste problema em particular.
- Problema B: a principal melhoria foi em relação ao teste de primalidade utilizado. Na versão sequencial fornecida, para testar se um número N é primo, testa-se a divisibilidade por todos os fatores possíveis até $\sqrt{N}$, de modo que o tempo consumido é $O(\sqrt{N})$, o que é exponencial no número de dígitos de N. Já a solução submetida usou o teste AKS de primalidade, cujo tempo consumido é poli-logarítmico no número de dígitos de do número a ser verificado N [6].

Em outras palavras, muitas vezes é possível obter melhorias significativas de desempenho sem necessariamente paralelizar o código, utilizando estruturas de dados, algoritmos e opções de compilação adequadas, por exemplo.

3.3. Conceitos importantes para avaliar o desempenho

Esta seção apresenta alguns conceitos importantes para que seja possível avaliar o desempenho de programas.

Primeiramente, um ponto importante é definir o que é desempenho. Em termos computacionais, o desempenho pode ser considerado como a quantidade de trabalho útil realizado por um computador. O desempenho deve ser mensurável e, para isso, algumas métricas são utilizadas.

O desempenho pode também ser absoluto ou relativo. Em termos absolutos, temos como exemplo as métricas: tempo de execução, latência, vazão (*throughput*), consumo energético e operações por segundo. Outras métricas podem ser calculadas por meio de operações aritméticas considerando várias métricas absolutas. Nesse caso, essas métricas fazem mais sentido quando são utilizadas para comparação entre sistemas, pois isoladamente elas não possuem significado. Por exemplo, podemos combinar métricas de tempo de execução e de gasto de energia simultaneamente - isso é feito em uma famosa lista de HPC verde [7].

[4] `https://pt.wikipedia.org/wiki/Trie`
[5] `https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`
[6] `https://mathworld.wolfram.com/AKSPrimalityTest.html`
[7] `https://www.top500.org/lists/green500/`

Quando se deseja comparar o desempenho, é necessário que, de todos os fatores que possam influenciar no desempenho, somente um ou poucos fatores sofram variação. Com isso, é possível avaliar a influência da alteração daquele fator (através de simples comparação dos valores obtidos) ou dos poucos fatores que foram alterados (utlizando técnicas de análise de variância ou outras técnicas estatísticas).

Em computação, um *benchmark* é um programa (ou um conjunto de programas), definido especificamente com o objetivo de avaliar o desempenho, podendo ser executado em diferentes configurações. As métricas coletadas nas execuções podem então ser comparadas, sendo possível afirmar qual configuração tem o melhor desempenho sob algum aspecto. Um dos *benchmarks* mais conhecido é o NAS [8], que fornece diversos tipos de código, com diferentes características.

Vários são os fatores que influenciam o desempenho dos programas, incluindo fatores em nível de hardware (por exemplo, hierarquia de memória e níveis de cache, velocidade e conjunto de instruções do processador) e software (por exemplo, estrutura do código, compilador/interpretador e sistema operacional).

Como a análise de desempenho está intimamente ligada a medidas, é essencial também se levar em conta a precisão. Variações em medições de tempo de execução são usuais em computação, e existem várias formas de se procurar encontrar relevância estatística. A regra mais comum, que infelizmente não é usada de forma adequada, é a de se repetir a medição 30 vezes, e apresentar a média e o desvio padrão encontrado. Mas, em muitos casos isso não é suficiente, pois quando o desvio padrão é alto, a média pode deixar de ser significativa.

O tema de relevância estatística foge um pouco do escopo do texto, mas vamos mostrar um exemplo de como esse processo pode ser bem feito; ou seja, em uma medição, vamos mostrar a distribuição dos valores. Com isso podemos verificar se a distribuição obedece um padrão (se espera uma curva em forma de sino), e nesse caso, podemos falar em média e desvio padrão com segurança. Para o leitor interessado sugerimos as leituras de [Jain 1991, Kalibera and Jones 2013].

Vejamos os tempos de execução do programa a seguir. Ele aloca um vetor de tamanho `SIZE`, o preenche com números aleatórios e depois calcula a média dos números:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4  #define SIZE  2000000
5
6  int media() {
7      int v[SIZE], sum = 0;
8      for (int i = 0; i < SIZE; i++)
9          v[i] = rand() % 100; // Random number between 0 and 99
10     for (int i = 0; i < SIZE; i++)
11         sum = sum + v[i];
12     return sum/SIZE;
13 }
14
15 void main() {
```

[8] https://www.nas.nasa.gov/software/npb.html

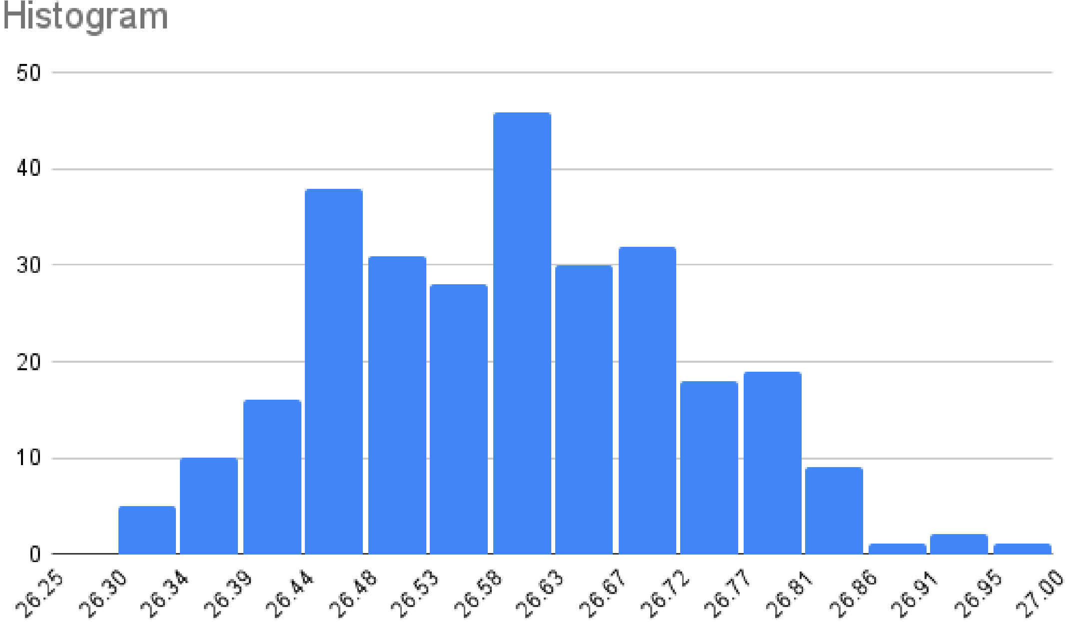


Figura 3.1. Histograma com os valores do tempo de execução da média dos elementos do vetor em milissegundos

```
16     clock_t start, end;
17     double cpu_time_used;
18     int val;
19     for (int i = 0; i < 300; i++){
20         start = clock();
21         val = media();
22         end = clock();
23         cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
24         printf("%g \n",  cpu_time_used);
25     }
26 }
```

Ao fazer um histograma com os valores obtidos em um processador i7 de nona geração (desconsiderando *outliers*, que são valores muito altos - sim, uma das execuções levou mais de 59 milissegundos), podemos observar na Figura 3.1 que a média sim, faz sentido. Dois indícios são importantes nessa distribuição: a forma de sino e o desvio padrão pequeno. Nesse caso, podemos dizer que a média foi de 26.6 milissegundos com desvio padrão 0.3 milissegundos.

No entanto, nem sempre a média representa de forma adequada o tempo de execução de um programa. Considere o programa a seguir, que calcula o produto de duas matrizes quadradas de ordem 2048 contendo valores aleatórios do tipo *double*. A variação dos tempos de execução é devida apenas às mudanças com relação aos tamanhos do blocos usados.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
```

```
4 #include <time.h>
5
6
7 void matrix_fill_rand(int n, double *restrict _A)
8 {
9     #define A(i, j) _A[n*(i) + (j)]
10   int i, j;
11
12   for (i = 0; i < n; ++i)
13         for (j = 0; j < n; ++j)
14             A(i, j) = 10*(double) rand() / (double) RAND_MAX;
15
16     #undef A
17 }
18
19 void mmul(int n, double *restrict _C, double *restrict _A, double *
    restrict _B)
20 {
21     // Macros para acesso aos elementos das matrizes
22     #define A(i, j) _A[n*(i) + (j)]
23     #define B(i, j) _B[n*(i) + (j)]
24     #define C(i, j) _C[n*(i) + (j)]
25
26     int min_BLC = 32;
27     int max_BLC = 1024;
28
29     // Tamanho do bloco aleatorio entre min_BLC e max_BLC
30     int BLC = min_BLC + (rand()/(double) RAND_MAX)*(max_BLC - min_BLC);
31
32     int ib, jb, kb, i, j, k;
33     int num_blocks = n/BLC + 1;
34
35     for (ib = 0; ib < num_blocks; ++ib)
36         for (kb = 0; kb < num_blocks; ++kb)
37             for (jb = 0; jb < num_blocks; ++jb)
38                 for (i = ib*BLC; i < (ib+1)*BLC && i < n; ++i)
39                     for (k = kb*BLC; k < (kb+1)*BLC && k < n; ++k)
40                         for (j = jb*BLC; j < (jb+1)*BLC && j < n; ++j)
41                             C(i, j) += A(i, k)*B(k, j);
42
43     #undef A
44     #undef B
45     #undef C
46 }
47
48 int main()
49 {
50     double *restrict A;
51     double *restrict B;
52     double *restrict C;
53
54     int n = 2048;
55
56     // Aloque memoria com alinhamento de 8 bytes
57     A = aligned_alloc(8, n*n*sizeof(*A));
```

```
58      B = aligned_alloc(8, n*n*sizeof(*B));
59      C = aligned_alloc(8, n*n*sizeof(*C));
60
61      // Usar tempo atual como seed para geracao de numeros (pseudo-)
       aleatorios
62      srand(time(0));
63
64      // Matrizes A e B preenchidas com valores aleatorios
65      matrix_fill_rand(n, A);
66      matrix_fill_rand(n, B);
67
68      // Matriz C inicialmente preenchida com zeros
69      memset(C, 0x00, n*n*sizeof(*A));
70
71      clock_t start, end;
72      double cpu_time_used;
73
74      start = clock();
75      // Calcula o produto das matrizes A e B e armazena o
76      // resultado na matriz C
77      mmul(n, C, A, B);
78      end = clock();
79
80      cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
81
82      printf("%lf\n", cpu_time_used);
83
84      free(A);
85      free(B);
86      free(C);
87
88      return 0;
89  }
```

A Figura 3.2 mostra a distribuição dos tempos de 100 execuções do programa em um processador Intel i7-8550U 1.80GHz. Nesse caso, o tempo médio de execução foi de 5.2 segundos e o desvio-padrão de 0.8s, mais significativo do que no exemplo anterior. Apesar do desvio padrão ser relativamente pequeno, ao valor da média é pouco significativo.

3.3.1. Ferramentas utilizadas para medir o desempenho

Esta seção apresenta alguns programas utilizados para medir o desempenho de programas e as métricas que podem ser obtidas a partir deles. Além de programas, também é possível utilizar dispositivos físicos, como medidores de consumo de energia ou processadores auxiliares que obtêm informação fora do dispositivo.

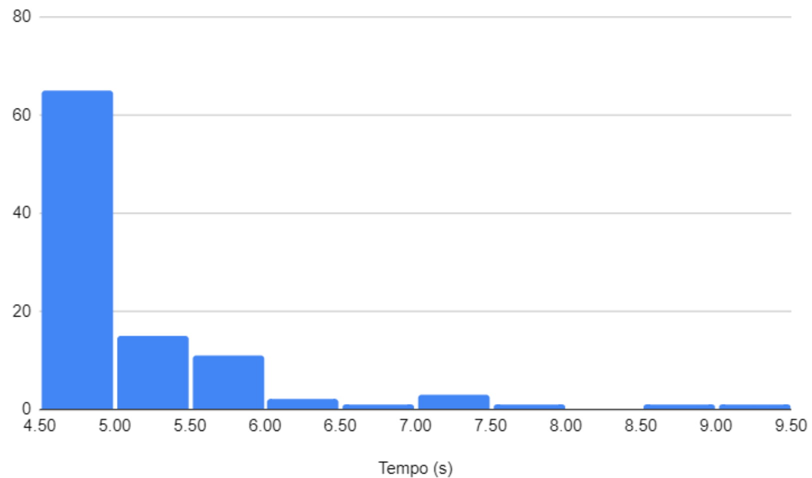


Figura 3.2. Histograma com os valores do tempo de execução da multiplicação de matrizes em segundos

3.3.1.1. Time

O utilitário `time`[9], especificado na ISO/IEC 9945-2:1993, é utilizado para medir a duração da execução de um programa qualquer em sistemas POSIX. Ele pode ser disponibilizado como um programa individual ou como um comando embutido em *shell*s. Sua saída padrão é dependente da implementação, assim, o argumento `-p` pode ser utilizado para emitir o resultado em um formato padronizado.

Na chamada padrão são mostrados três valores: o **real**, que corresponde ao tempo total (ou de relógio) entre o início e o final do programa, o tempo **user**, que é o tempo gasto pelas instruções do programa, e finalmente o **sys**, que o tempo usado em chamadas de sistema (*kernel*) do programa. Mas, há diversas *flgas* que podem ser usadas, por exemplo para obter também informações sobre a memória. A vantagem de se usar o `time` é que ele não é invasivo, mas só mede o programa como um todo.

3.3.1.2. Perf

O `perf`[10] é um comando que utiliza o subsistema de contadores de desempenho do *kernel Linux* para gerar métricas detalhadas do sistema. O `perf` oferece uma quantidade de eventos que podem ser obtidos a partir de diferente fontes. Algumas métricas são do próprio kernel do Linux (troca de contexto, *page-fault*) e outras são obtidas a partir dos contadores disponíveis na *Performance Monitoring Unit* - (PMU), tais como ciclos de processador utilizados e referências à memória cache.

Apesar de ser específico para sistema com o *kernel Linux*, outros sistemas possuem ferramentas com funcionalidades similares, como o *Instruments*[11] para *MacOS*.

[9] `https://pubs.opengroup.org/onlinepubs/009604499/utilities/time.html`
[10] `https://perf.wiki.kernel.org/index.php/Main_Page`
[11] `https://developer.apple.com/xcode/features/`

3.3.1.3. Valgrind

O *Valgrind*[12] é um arcabouço para análise de programas em tempo de execução. Apesar de ser conhecido como ferramenta para a detecção de vazamento de memória, ele também pode ser utilizado para a geração de árvores de execução e análise de uso do *cache* do processador. Devido à utilização de uma técnica de virtualização para a análise dos programas, ele aumenta consideravelmente a duração da execução do programa.

3.3.1.4. Vtune™ Profiler

O *Intel® Vtune™ Profiler*[13] é um ambiente integrado para análises de programas exclusivo para plataformas da *Intel*. Ele permite a análise de diversos tipos de programas, incluindo a análise de programas que rodam em *clusters MPI*, gerando um conjunto diverso de métricas, como contadores de desempenho de hardware, utilização do *cache* e consumo de energia.

3.3.2. Tabulação e análise dos dados

Esta seção apresenta informações sobre a tabulação dos dados coletados e algumas ferramentas para análise deles.

3.3.2.1. Resultados, parâmetros e ambiente

Além do registro dos resultados, é importante registrar informação sobre os parâmetros utilizados e o ambiente de execução. Os parâmetros compreendem as informações passadas que alteram o funcionamento do programa, como variáveis de ambiente que determinam número de *threads* a serem utilizadas ou os argumentos de execução. O ambiente de execução compreende dados sobre o *hardware* e o *software* em que o *benchmark* está sendo executado.

3.3.2.2. Tabulação

Para experimentos onde poucos dados são gerados, o registro manual dos dados é uma opção viável e pode ser feita utilizando uma planilha eletrônica. Para quantidades maiores de dados, o registro automático através de *script*s passam a ser um caminho mais vantajoso pela economia de tempo e menor chance de erros no processo de registro.

Ao automatizar o processo é necessário definir o formato de registro utilizado, como arquivos CSV ou bancos SQLite. Além da facilidade de registro, deve ser considerado o suporte de leitura do formato escolhido pelas ferramentas de análise. No caso de campos não estruturados, representações como JSON podem apresentar a flexibilidade necessária.

[12] `https://valgrind.org`

[13] `https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html`

3.3.2.3. Análise

Após o registro dos dados, a análise pode ser realizada por uma vasta gama de programas. Além de planilhas eletrônicas, pode ser empregado o uso de programas como *Jupyter notebooks*[14] com bibliotecas como a *Matplotlib*[15], ou programas como o *Jamovi*[16].

3.4. Parte 1

Nosso objetivo nesta primeira parte do minicurso é apresentar como obter algumas métricas e utilizá-las para comparação de um mesmo algoritmo implementado em várias linguagens.

Para isso, utilizaremos o algoritmo *binary-trees* como *benchmark*, o qual é uma adaptação do benchmark de Hans Boehm para analisar *Garbage Collection* (GC) [17]. Esse *benchmark* aloca milhões de *short-lived trees* e percorre-as.

As linguagens que iremos comparar são: C, Python e Go. Os três códigos estão disponíveis no *Benchmarks Game* e são as versões que possuem o melhor desempenho em suas respectivas linguagens.

3.4.1. Compilação e Execução

Arquivos `makefile` são utilizados para definir regras de modo que o processo de compilação seja mais automático. Neste minicurso serão definidos dois arquivos de `makefile` para a compilação dos programas em C e Go.

O código em C utilizará o compilador GCC[18], o código em Go utilizará o compilador Go[19] e o código em Python utilizará o interpretador Python[20].

O `Makefile` para o programa em C define o compilador, as *flags* de compilação, os diretórios para os arquivos de *header* e das bibliotecas, o caminho onde é encontrada a biblioteca que será linkada ao arquivo executável, e o nome dos arquivos fonte, objeto e executável:

```
1  # the compiler: gcc for C program
2  CC=gcc
3
4  # compiler flags:
5  CFLAGS=-pipe -Wall -O3 -fomit-frame-pointer -march=ivybridge -fopenmp
6
7  # directory containing hearder files
8  INCLUDES=-I/usr/local/apr/include/apr-1
9
10 # library paths:
11 LFLAGS=-L/usr/local/apr/lib -fopenmp
12
```

[14] `https://jupyter.org`
[15] `https://matplotlib.org/`
[16] `https://www.jamovi.org`
[17] `https://hboehm.info/gc/gc_bench/`
[18] `https://gcc.gnu.org/`
[19] `https://go.dev/`
[20] `https://www.python.org/`

```
13 # library to link into executable
14 LIBS=-lapr-1
15
16 # C source file
17 SRCS=binary-trees.c
18
19 # C object file
20 # chance .c in SRSC by .o
21 OBJS=$(SRCS:.c=.o)
22
23 # executable file
24 MAIN=binary-trees
25
26 .PHONY: depend clean
27
28 all:  $(MAIN)
29   @echo File compiled
30
31 $(MAIN): $(OBJS)
32   $(CC) $(OBJS) -o $(MAIN)  $(LFLAGS) $(LIBS)
33
34 $(OBJS): $(SRCS)
35   $(CC) $(CFLAGS) $(INCLUDES) -c $(SRCS)
36
37 clean:
38   $(RM) *.o *~ $(MAIN)
```

Nesse exemplo do *benchmark binary-trees*, é necessário instalar o *Apache Portable Runtime*[21], o qual irá fazer o gerenciamento de alocação de memória do programa.

Para compilação do código em Go, basta ter instalado o compilador e executar o seguinte `Makefile`:

```
1 build:
2   go build -o binary-trees.go_run binarytrees.go-2.go
```

No caso da linguagem Python, como se trata de uma linguagem interpretada, não é necessário compilar o código, sendo que o comando será executado diretamente na execução.

O esforço de deixar claros todos os parâmetros usados, como ambiente, compilador e *flags* é essencial para permitir a reprodutibilidade dos experimentos.

Para facilitar a repetição da execução dos programas, como descrito em 3.3, foi elaborado um *script shell* que repete a execução dos três programas que serão comparados, coletando o tempo de execução de cada repetição e armazenando um arquivo texto:

```
1 # ===================
2 # ||    CLEANUP    ||
3 # ===================
4
5 rm results/*.out
6
7 # ===================
```

[21]`https://apr.apache.org/`

```
# ||   EXECUTION   ||
# ===================

SOURCE="binary-trees"
PARAM="15"


for i in $(seq 1 30)
do
  (time C/$SOURCE $PARAM) 2>> results/C.out
  (time python3 -OO Python/$SOURCE.py $PARAM) 2>> results/Python.out
  (time Go/$SOURCE.go_run $PARAM) 2>> results/Go.out
done
```

3.4.2. Métricas

Em termos de métricas a serem analisadas, as ferramentas descritas em 3.3.1 podem fornecer muitas métricas, tais como: tempo de execução, quantidade de troca de contexto, *page-fault*, quantidade de ciclos de máquina utilizados, quantidade de instruções, *branchs* (desvios) e *branch-misses* (previsão errada no desvio). A saída abaixo é resultado da execução do `perf` com o seguinte comando:

```
sudo perf stat -o ../results/perf.out ./binary-trees 15
       ../results/saida.out
```

```
# started on Fri Sep 30 18:24:23 2022


 Performance counter stats for './binary-trees 15 saida.out':

             43,72 msec task-clock           #    2,824 CPUs utilized
                85      context-switches     #    1,944 K/sec
                 3      cpu-migrations       #   68,625 /sec
               784      page-faults          #   17,934 K/sec
       139.404.625      cycles               #    3,189 GHz
       331.406.108      instructions         #    2,38  insn per cycle
        58.226.381      branches             #    1,332 G/sec
            98.349      branch-misses        #    0,17% of all branches

       0,015479133 seconds time elapsed

       0,044337000 seconds user
       0,000000000 seconds sys
```

Se o teste estiver sendo executado em uma máquina virtual ou em um *container*, alguns resultados podem aparecer como `<not supported>` pois o `perf` não consegue coletar algumas métricas de hardware.

Para essa parte do minicurso, utilizaremos a ferramenta `time`, como já observado na Seção 3.4.1 e é com base nesses valores que faremos a comparação das linguagens.

3.4.3. Tabulando os dados

Para a tabulação dos dados, foi elaborado um programa em Python que lê os arquivos de saída gerados (C.out, Python.out e Go.out), trata os valores, insere-os em um *dataframe*,

salva o *dataframe* em um arquivo .csv e gera os gráficos para cada linguagem e cada métrica analisada (tempos real, user e sys).

```
import os, math
import subprocess
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

NUM_EXPERIMENTS = 30


def proccess_file(df, lang):
    data  = { 'language':lang }

    f     = open('results/{}.out'.format(lang))

    line = f.readline()
    line = f.readline()

    for i in range (NUM_EXPERIMENTS):
      if 'real' in line:
        # Trata a linha com a informacao do tempo real
        line = line.strip().split()
        time = line[1].split("m")
        seg = time[1].split("s")
        seg = seg[0].replace(',', '.')
        time2 = float(time[0])*60 + float(seg)
        data['real'] = time2
        # Trata a linha com a informacao do tempo user
        line = f.readline()
        line = line.strip().split()
        time = line[1].split("m")
        seg = time[1].split("s")
        seg = seg[0].replace(',', '.')
        time2 = float(time[0])*60 + float(seg)
        data['user'] = time2
        # Trata a linha com a informacao do tempo sys
        line = f.readline()
        line = line.strip().split()
        time = line[1].split("m")
        seg = time[1].split("s")
        seg = seg[0].replace(',', '.')
        time2 = float(time[0])*60 + float(seg)
        data['sys'] = time2
        line = f.readline()
        line = f.readline()
        df = df.append(data, ignore_index=True)

    f.close()

    return df

def proccess_data():
    languages = ['C', 'Python', 'Go']

```

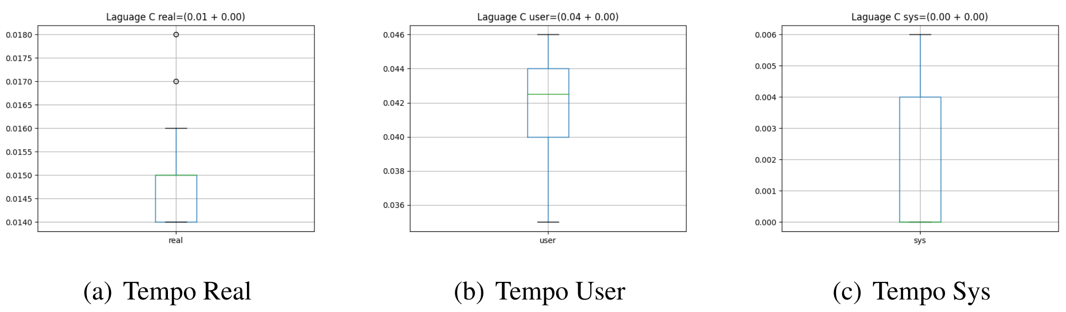
```
54      # Criando o DataFrame a ser populado
55      column_names = ['real', 'user', 'sys']
56      df = pd.DataFrame(columns = column_names)
57
58      # Insere as informacoes de cada arquivo de resultado
59      for lang in languages:
60        df = proccess_file(df,lang)
61
62      return df
63
64
65  def processs_results(df):
66
67      languages = ['C', 'Python', 'Go']
68      metrics = ['real', 'user', 'sys']
69
70      # Gerando Box plot dos graficos de maneira simples e calculando o
        intervalo de confianca
71      for lang in languages:
72          if lang == 'C':
73            str_option = 'C'
74          elif lang == 'Python':
75            str_option = 'Python'
76          elif lang == 'Go':
77            str_option = 'Go'
78
79          print("Gerando Imagens para Linguagem {}".format(str_option))
80
81          temp_df = df[df['language'] == lang]
82          print(temp_df)
83          for c in metrics:
84              # Calculando o intervalo de confianca a 95% com t-student
85              conf_interval = (2.5096 * temp_df[c].std()) / math.sqrt(
        float(NUM_EXPERIMENTS))
86
87              plt.title("Laguage {} {}=({:.2f} + {:.2f})".format(
        str_option, c, temp_df[c].mean(), conf_interval ))
88              temp_df.boxplot(column=c)
89              plt.savefig('figures/fig_{}_{}'.format(lang, c))
90              plt.close('all')
91
92
93  if __name__ == '__main__':
94
95      print("Processamento os arquivos de resutlados")
96      df = proccess_data()
97      print("Salvando resultados em disco (results/results.csv)...")
98      df.to_csv('results/results.csv', index=False, header=True)
99      print("Resultados salvos em um Dataframe {}.".format(df.shape))
100
101     print("Processando a Analise dos dados")
102     processs_results(df)
103     print("Analise finalizada")
```

Listagem 3.1. Código para ler os resultados e gerar os gráficos

3.4.4. Analisando os resultados

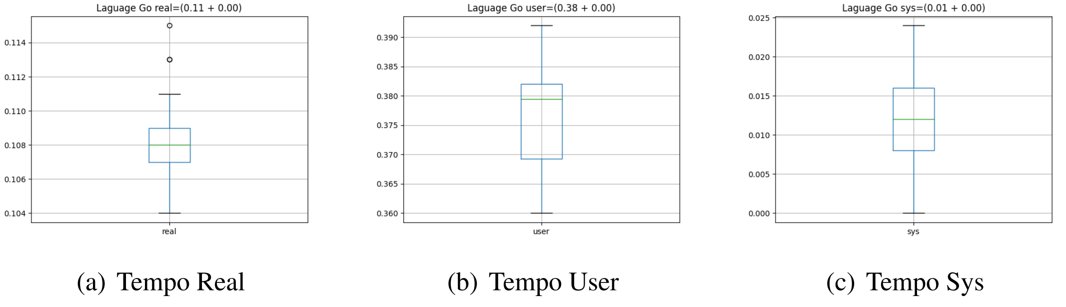
Os gráficos gerados pelo código da Listagem 3.1 podem ser observados nas Figuras 3.3, 3.5 e 3.4, as quais apresentam boxplots com 30 resultados para cada código para cada métrica. Além disso, são apresentados também os valores referentes à média e a metade do valor do intervalo de confiança com nível de confiança de 95%.

A linguagem que obteve o melhor desempenho (menor tempo de execução) é a linguagem C, seguida pela linguagem Go. Python obteve o pior desempenho. Esses resultados confirmam os resultados apresentados na página com os *benchmarks*.



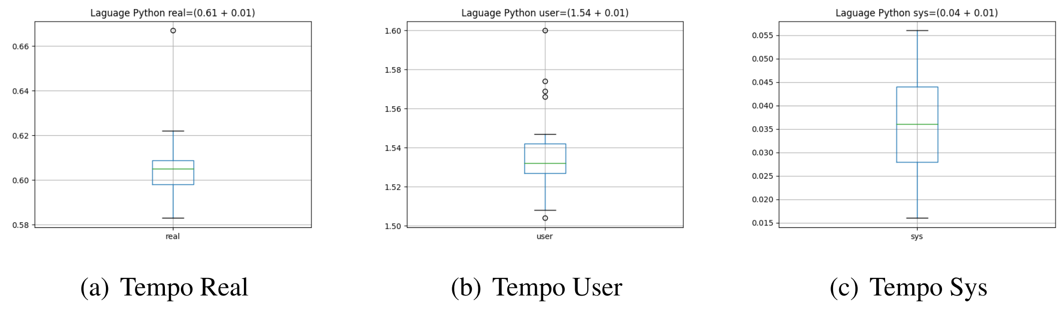
(a) Tempo Real (b) Tempo User (c) Tempo Sys

Figura 3.3. Tempos para linguagem C



(a) Tempo Real (b) Tempo User (c) Tempo Sys

Figura 3.4. Tempos para linguagem Go



(a) Tempo Real (b) Tempo User (c) Tempo Sys

Figura 3.5. Tempos para linguagem Python

É interessante também ressaltar que uma outra métrica que pode ser usada é o valor mínimo, ou seja, entre todas a medidas a de menor valor. Dessa forma, temos que o

tempo real em C é 0.014 segundos, em Go, 0.1 segundos. O menor tempo em Python foi de 0.58 segundos.

Pode-se observar também que o tempo **real** é menor do que o tempo **user**, quando o que se esperava era que fosse o contrário. Isso significa que o código é CPU Bound e que aproveitou o paralelismo de execução em vários núcleos.

3.5. Parte 2

Nesta parte são explorados tópicos de análise de desempenho de programas. Utilizaremos o cálculo do conjunto Mandelbrot[22] para realizar as análises. Primeiro será apresentada a definição geral do programa, com implementações disponíveis para as linguagens C[23], Python[24] e Go[25]. Posteriormente são abordados os diferentes tipos de relógios que um sistema pode oferecer utilizando a versão em C; impacto de *flags* no tempo de execução e compilação usando o GCC; uso de compiladores/interpretadores alternativos usando o PyPy e considerações sobre compilação JIT; *profiling* usando a versão em Go; e verificação de saída e soma de verificação usando a versão em Go.

3.5.1. Conjunto Mandelbrot

O conjunto Mandelbrot consiste nos números complexos c tais que a função $f_c(z) = z^2 + c$ não diverge quando aplicada de forma recursiva: $f_c(f_c(...f_c(0)))$. Para fins computacionais, consideramos que um número complexo diverge quando, em algum momento da aplicação recursiva da função, a norma euclidiana do valor intermediário é maior que 2, e consideramos que um número complexo converge quando, após um número pré-estabelecido de iterações, nenhum de seus valores intermediários apresenta norma euclidiana maior que 2. O programa gera uma imagem 2D onde o eixo das abscissas representa a parte real e o eixo das ordenadas a parte imaginária.

O programa recebe como argumentos de entrada a largura (w) e altura (h) da imagem a ser gerada e os valores em ponto flutuante x_r, x_i, y_r e y_i que serão utilizados para definir o conjunto de pontos analisados. Para cada pixel da imagem (x,y), sendo $(0,0)$ o ponto superior esquerdo e $(w-1,h-1)$ o ponto inferior esquerdo, é atribuído o número complexo $(x/w*(y_r - x_r) + x_r) + (y/h*(x_i - y_i) + y_i)i$. Caso o número atribuído faça parte do conjunto Mandelbrot ele é pintado de branco, caso contrário, de preto. Por fim, o programa deve escrever na saída padrão a imagem utilizando o formato Netpbm P1[26].

3.5.2. Relógios do sistema

Sistemas operacionais modernos possuem diferentes fontes de tempo disponíveis para os processos. Essas fontes avançam com base em critérios diferentes e permitem obter diferentes métricas para análise.

[22] `https://en.wikipedia.org/wiki/Mandelbrot_set`

[23] `https://github.com/elisauhura/ERAD-benchmarks/blob/main/Parte2/mandelbrot.c`

[24] `https://github.com/elisauhura/ERAD-benchmarks/blob/main/Parte2/mandelbrot.py`

[25] `https://github.com/elisauhura/ERAD-benchmarks/blob/main/Parte2/mandelbrot.go`

[26] `https://en.wikipedia.org/wiki/Netpbm`

Fonte	Segundos	Pontos Convergentes
thread 1	0.022489s	5406
thread 2	0.022460s	5525
thread 3	0.112023s	219960
thread 4	0.112285s	220777
mono	0.116688s	-
proc	0.271308s	-

Tabela 3.1. Dados de tempo e pontos convergentes para o programa Mandelbrot em C com os argumentos: largura = 1600; altura = 1600; xr = -1.5; xi = -1.5; yr = 1.5; yi = 1.5; máximo de iterações = 50; workers = 4

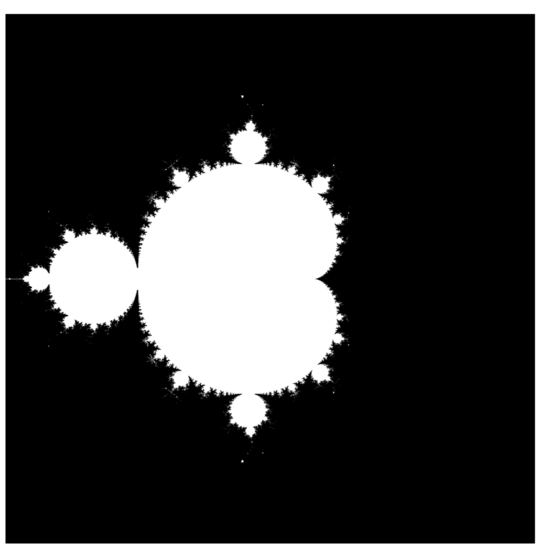


Figura 3.6. Exemplo de imagem gerada pelos programas Mandelbrot

Um modo portátil para ter acesso a um conjunto útil de fontes é por meio da função `clock_gettime`[27], disponível em sistemas POSIX.

Sistemas GNU/Linux costumam disponibilizar 3 fontes de interesse para análise de programas: CLOCK_MONOTONIC, CLOCK_PROCESS_CPUTIME_ID, CLOCK_-THREAD_CPUTIME_ID. Elas representam, respectivamente, uma fonte de tempo que incrementa de forma consistente, uma fonte de tempo que incrementa de forma consistente sempre que o processo está em execução e uma fonte de tempo que incrementa de forma consistente sempre que a *thread* que invocou a função está em execução.

A versão em C do programa faz uso das três fontes de tempo diferentes e escreve na saída padrão de erro a duração de execução para cada *thread*, para o processo como um todo (proc) e o tempo decorrido no mundo real (mono). No caso das *threads*, também é escrito a quantidade de pontos dentre os processados por cada *thread* que convergiram.

A Tabela 3.1 apresenta um exemplo dos dados gerados para uma execução utilizando 4 *threads* em um computador 4-core comum para gerar a Figura 3.6. É possível

[27] `https://linux.die.net/man/3/clock_gettime`

Otimização	Média (compile)	Média (runtime)
-O0	0.04s	0.14s
-O1	0.07s	0.064s
-O2	0,07s	0,060s
-O3	0,07s	0,060s

Tabela 3.2. Tempos de compilação e execução para a versão em C do programa

observar que o tempo de execução do programa (mono) é dominado em boa parte por duas das *threads*, enquanto as outras completam suas partes da imagem antes, indicando uma má distribuição da carga de trabalho. Como cada *thread* processa uma quantidade similar de pontos, a diferença de carga está na quantidade de iterações para determinar se um ponto diverge ou não.

3.5.3. Impacto de flags no *runtime* e *compile time*

Um forma de otimizar a execução de um programa é através da alteração das *flags* de otimização passadas ao compilador. Geralmente, ao ativar uma *flag* acontece uma troca, onde tenta se reduzir o tempo de execução de um programa por meio de um maior tempo de análise e transformações durante a compilação.

Todavia, a tentativa de reduzir o tempo de execução nem sempre é alcançada, por isso é importante analisar o impacto do uso de *flags* tanto no tempo de execução quanto no de compilação por ser possível que o tempo extra gasto na compilação não seja vantajoso em determinadas partes do estudo.

A Tabela 3.2 apresenta o tempo de compilação e execução médio com 10 amostras para as flags -O0, -O1, -O2 e -O3 e com os mesmos argumentos de execução da Tabela 3.1 em um computador 4-core comum utilizando o compilador *Clang* (a variância foi omitida pois todas foram inferiores a 10^{-4}). É possível observar que o melhor tempo de execução está com as otimizações -O2 e -O3.

É importante ressaltar que o espaço de busca por *flags* adequadas vai muito além das otimizações padrão, pois certas combinações de *flags* fora do padrão -O"podem ser benéficas, ou mesmo levar a erros na execução. Quando se pensa em desempenho uma busca pelas melhores *flags* não pode ser descartada, mas só lembrando o espaço de busca pode ser exponencial. Para o leitor interessado sugerirmos um artigo clássico [Hoos 2012] e um mais recente que mostra o potencial de uso de *flags* para GPUs [Bruel et al. 2017].

3.5.4. Compiladores e interpretadores alternativos

Uma outra alternativa para otimizar o programa é através do uso de compiladores ou interpretadores alternativos. Eles podem explorar otimizações não implementadas no projeto principal ou utilizarem como alvo aceleradores disponíveis no sistema, como GPUs ou TPUs.

Um caso em que é comum observar uma grande diferença nos números observados é em linguagens interpretadas como no caso do Python. Utilizando a versão em Python do programa e os mesmos argumentos da Tabela 3.1 (exceto pela omissão dos

workers, já que esta é uma versão sequencial), com o interpretador padrão da linguagem (Python 3.10.14), para 10 amostras, foi observado um tempo de execução médio de 7.43 segundos com variância de 0,04 e com o interpretador alternativo (Pypy 7.3.9), para 10 amostras, foi observado um tempo de execução médio de 0,53 segundos com variância de 0.00021. Esse salto acontece pelo uso da técnica de compilação *just-in-time* utilizada pelo Pypy.

3.5.5. Considerações sobre interpretadores *just-in-time*

Uma forma de se aumentar o desempenho de linguagens interpretadas como Python, ou Julia é o uso da compilação *just-in-time* (JIT). Usualmente, o código é interpretado para ser executado e o tempo gasto com a interpretação pode ser considerável. Algo semelhante acontece em Java, onde após a pré-compilação em bytecode (javac), o código é interpretado (java), dessa forma os processos de compilação e execução ficam explícitos.

Em casos que a perca de tempo interpretando é grande o bastante, a técnica do JIT se torna vantajosa já que o tempo gasto na compilação em tempo de execução é compensado pelo ganho de desempenho na execução do código em si. Porém é sempre bom lembrar as questões ligadas ao *warm-up*, ou aquecimento, pois ao menos a primeira execução será mais lenta. Logo, assim como em experimentos podemos cortar os valores extremos (*outliers*), os valores iniciais podem ser ignorados.

Vale lembrar que também pode ocorrer um efeito contrário, de uma aceleração artificial de programas após as primeiras execuções. Isso geralmente ocorre pois os níveis de *cache* já podem conter dados relevantes para às execuções seguintes. Nesses casos, para uma análise de desempenho justa é importante limpar as memórias *cache*. Isso pode ser feito facilmente alternando diferentes programas a serem medidos.

3.5.6. *Profiling*

Profiling é uma forma de análise dinâmica que visa medir diversos aspectos de um programa. Essa análise pode ser realizada com pouco impacto no desempenho do programa ao utilizar funcionalidades disponíveis em processadores. Para análises mais completas ou em sistemas que não possuem suporte de hardware, é possível utilizar técnicas de virtualização que, apesar de reduzirem o desempenho do programa durante o processo, permitem uma análise mais detalhada do comportamento do mesmo.

A linguagem Go provê um ambiente prático por incluir diversas ferramentas para analisar o comportamento do programa, como a *go tool pprof*. A versão `mandelbrot_prof.go`[28] possui as alterações necessárias no código da versão em Go do programa para realizar a coleta dos dados para a ferramenta. Com a alteração, apesar de se comportar como o programa anterior, a versão modificada gera um arquivo `cpu.prof` que contem os detalhes da execução do programa. Como a ferramenta é baseada na coleta de amostras em curtos intervalos do programa, execuções curtas (menos de 5 segundos) podem não trazer todo o detalhe desejado e com isso, uma possibilidade é aumentar a duração da tarefa a ser realizada.

[28] `https://github.com/elisauhura/ERAD-benchmarks/blob/main/Parte2/mandelbrot_prof.go`

flat	flat%	sum%	cum	cum%	node
58.11s	96.67%	96.67%	59.04s	98.22%	main.work
0.93s	1.55%	98.22%	0.93s	1.55%	runtime.asyncPreempt
0.73s	1.21%	99.43%	0.73s	1.21%	runtime.memmove
0	0%	99.43%	0.38s	0.63%	fmt.(*buffer).writeString
0	0%	99.43%	0.38s	0.63%	fmt.(*fmt).fmtS
0	0%	99.43%	0.38s	0.63%	fmt.(*fmt).padString
0	0%	99.43%	0.38s	0.63%	fmt.(*pp).doPrintln
0	0%	99.43%	0.38s	0.63%	fmt.(*pp).fmtString
0	0%	99.43%	0.38s	0.63%	fmt.(*pp).printArg
0	0%	99.43%	0.39s	0.65%	fmt.Fprintln

Tabela 3.3. `top` gerado pelo pprof para o programa Mandelbrot em Go adaptado com os argumentos: largura = 28000; altura = 28000; xr = -1.5; xi = -1.5; yr = 1.5; yi = 1.5; máximo de iterações = 100; workers = 12

A Tabela 3.3 foi gerada ao utilizar o comando `top` do *go tool pprof*. Ela apresenta os 10 nós que representam o maior tempo de execução do programa. A coluna *flat* representa o tempo executando instruções dentro da função (nó) sem contar o tempo executando invocações internas de outras funções e a coluna *cum* conta o tempo dessas invocações internas. Como podemos observar, boa parte do tempo do programa é gasto determinando se os pontos divergem ou não (main.work) e o tempo gasto para escrever a imagem na saída padrão é ínfimo (fmt.(*buffer).writeString). Assim, o programa possui apenas uma área cuja otimizações e melhorias impactariam de forma significativa o desempenho do programa.

Devido a simplicidade do programa utilizado, o *profiling* acaba sendo uma técnica que não traz muito mais valor do que uma simples análise do código, porém em projetos maiores, as métricas geradas podem auxiliar no processo de encontrar as partes mais importantes a serem otimizadas.

3.5.7. Soma de verificação

Um ponto crucial que não pode ser esquecido é o fato que desempenho deixa de ser relevante quando o resultado obtido não é o esperado. Ou seja, antes de se falar em melhorar o desempenho é essencial sempre garantir (de preferência por meio de testes automatizados) a corretude do programa. Talvez o exemplo mais conhecido foi o erro de divisão de ponto flutuante dos primeiros processadores Pentium [29].

Entretanto, para saídas muito grandes, não é viável sempre comparar uma saída completa. Para evitar isso, uma forma é usar algo como uma função de *hash*, permitindo que seja necessário guardar e comparar apenas pequenas strings no lugar de de toda a saída do programa. Além de simplificar o processo de validar a saída de um programa, ela pode reduzir o tempo de escrita, pois os algoritmos de *hash* podem funcionar como dispositivos de escrita que não exigem chamadas de sistema. A versão `mandelbrot_-`

[29] `https://en.wikipedia.org/wiki/Pentium_FDIV_bug`

Versão	Tempo médio	Linhas de Código
Go	0.064s	80
C	0.068s	145
Python (Pypy)	0.562s	44
Python (CPython)	7.421s	44

Tabela 3.4. Tempo médio de 10 execuções dos programas com os argumentos: largura = 1600; altura = 1600; xr = -1.5; xi = -1.5; yr = 1.5; yi = 1.5; máximo de iterações = 50; workers = 4

`sha.go`[30] possui as alterações necessárias para utilizar a função SHA-256[31] como função dispositivo de saída.

Outro ponto importante é que guardar a saída do programa pode ajudar a detectar erros de natureza não-determinística, como condições de corrida.

3.5.8. Diferença de desempenho entre linguagens

Como um assunto extra, a Tabela 3.4 apresenta o tempo médio de execução dos diferentes programas para uma mesma entrada (utilizando 4 *workers* nas versões em C e Go). É possível observar uma separação considerável entre as versões de linguagens compiladas e a versão em Python. Mesmo utilizando um interpretador JIT, a versão em Python continua uma ordem de grandeza mais lenta que as versões em C e em Go.

Ao comparar C e Python é possível argumentar que, apesar do pior desempenho, o programa possui um menor número de linhas de código. Porém, a versão em Go conseguiu apresentar o mesmo desempenho que a versão em C e quase metade das linhas de código, ao mesmo tempo que provê funcionalidades como Coletor de Lixo e CSP[32] na linguagem. Uma possível razão para Go ter apresentado um desempenho um pouco melhor que C se dá pela linguagem fazer uso de *goroutines*[33], que apresentam um overhead menor que as *threads* do *pthreads*.

Os programas executados na Parte 2 rodaram em MacBook Pro (15", 2019) com um processador Intel I7-9750H 6-core, 16 GB 2400 MHz DDR4 e rodando o MacOS 12.2.1 com o compilador C Apple clang version 13.0.0, compilador Go go1.17.8 e interpretadores Python 3.9.10 e PyPy 7.3.8.

3.6. Conclusão

Nesse texto apresentamos diversas técnicas de medir desempenho e de mostrar os resultados com relevância estatística. O nosso objetivo foi mostrar que é possível sistematizar a análise de desempenho. Além disso, o desempenho depende de diversos fatores, entre eles a linguagem de programação e/ou o compilador usado. Esperamos que esse texto

[30] `https://github.com/elisauhura/ERAD-benchmarks/blob/main/Parte2/mandelbrot_sha.go`

[31] `https://pkg.go.dev/crypto/sha256`

[32] https://en.wikipedia.org/wiki/Communicating_sequential_processes

[33] https://go.dev/doc/effective_go#goroutines

sirva como um começo de uma jornada!

Referências

[Bruel et al. 2017] Bruel, P., Amaris Gonzalez, M., and Goldman, A. (2017). Autotuning cuda compiler parameters for heterogeneous applications using the opentuner framework. *Concurrency and Computation Practice and Experience*, 29.

[Goldman et al. 2022] Goldman, A., Uhura, E., and Bruschi, S. (2022). Coisas para saber antes de fazer o seu próprio *Benchmarks Game*. In Lorenzon, A., Castro, M., and Pillon, M., editors, *Minicursos da XXII Escola Regional de Alto Desempenho da Região Sul*.

[Hoos 2012] Hoos, H. H. (2012). Programming by optimization. *Commun. ACM*, 55(2):70–80.

[Jain 1991] Jain, R. (1991). *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling*. Wiley New York.

[Kalibera and Jones 2013] Kalibera, T. and Jones, R. (2013). Rigorous benchmarking in reasonable time. In *Proceedings of the 2013 International Symposium on Memory Management*, ISMM '13, page 63–74, New York, NY, USA. Association for Computing Machinery.

Capítulo 4

Fundamentos de Computação Acelerada com CUDA C/C++

Arthur Francisco Lorenzon (UFRGS)

Abstract

This chapter highlights the fundamental tools and techniques for accelerating applications written in C/C++ languages to run on massively parallel architectures with CUDA. With CUDA, developers are able to dramatically accelerate the computation of applications on GPU (graphic processing unit) architectures. The Chapter has the following learning objectives: (I) writing parallel code to run on the GPU; (II) Expose and express data and instruction level parallelism in C/C++ applications using CUDA; (III) Use CUDA-managed memory and optimize memory migration using asynchronous prefetching; and (IV) Use concurrent streams for instruction-level parallelism.

Resumo

Este capítulo destaca as ferramentas e técnicas fundamentais para acelerar aplicações escritas em linguagens C/C++ para execução em arquiteturas massivamente paralelas com CUDA. Com CUDA, desenvolvedores são capazes de acelerar dramaticamente a computação de aplicações em arquiteturas GPU (graphic processing unit). O Capítulo tem os seguintes objetivos de aprendizagem: (I) escrever código paralelo para ser executado na GPU; (II) Expor e expressar paralelismo no nível de dados e instruções em aplicações C/C++ usando CUDA; (III) Utilizar o CUDA-managed memory e otimizar a migração de memória usando prefetching assíncrono; e (IV) Usar streams (fluxo de instruções) concorrentes para paralelismo no nível de instruções.

4.1. Introdução

Sistemas computacionais de alto desempenho são amplamente utilizados para executar aplicações de diversos domínios, que envolvem computação financeira, aplicações médicas, processamento de vídeo e imagem, entre outros. Esses sistemas geralmente são baseados em arquiteturas *multi-core* e *many-core* com memória e dispositivos de

computação altamente heterogêneos. Isso geralmente implica a existência de hierarquias de memória complexas e tecnologias que evoluem a cada ano. Assim, para continuar explorando o potencial dos sistemas computacionais de alto desempenho, as bibliotecas e interfaces de programação paralela se equipam com diversas políticas de gerenciamento de recursos [Navarro et al. 2020].

Neste cenário, a exploração do paralelismo em arquiteturas heterogêneas surge como um desafio para os desenvolvedores de *software* uma vez que estas arquiteturas são compostas de diferentes dispositivos computacionais. Um exemplo deste ambiente corresponde a combinação de CPUs de propósito geral (ou de alto desempenho) com unidades de processamento gráfico (*graphical processing units* - GPUs). As GPUs, também chamadas de aceleradores, fornecem um *throughput* de instruções e largura de banda de memória muito maiores que a CPU com similar consumo de energia. Assim, diversas aplicações aproveitam esses recursos para executar mais rapidamente na GPU do que na CPU. Adicionalmente, outros dispositivos de computação, como FPGAs (*field-programmable gate array*), também são muito eficientes em termos de energia, mas oferecem menor flexibilidade de programação do que GPUs [Knorst et al. 2021].

A diferença de recursos entre GPU e CPU existe porque elas são projetadas com objetivos diferentes em mente, conforme ilustrado na Figura 4.1. Enquanto a CPU é projetada para se destacar na execução de uma sequência de operações, chamada de *threads*, o mais rápido possível e pode executar algumas dezenas dessas *threads* em paralelo, a GPU é projetada para se destacar na execução de milhares de *threads* em paralelo, amortizando o desempenho mais lento fornecido por uma única por thread para obter maior *throughput*.

O interesse do uso de GPUs na computação de alto desempenho surgiu quando o potencial destes componentes foi combinado com uma linguagem de programação que tornou as GPUs mais fáceis de programar. Atualmente, o programador conta com diferentes interfaces de programação paralela que expõem o paralelismo em GPUs: CUDA (*Compute Unified Device Architecture*) [Sanders and Kandrot 2010] – criada pela NVI-

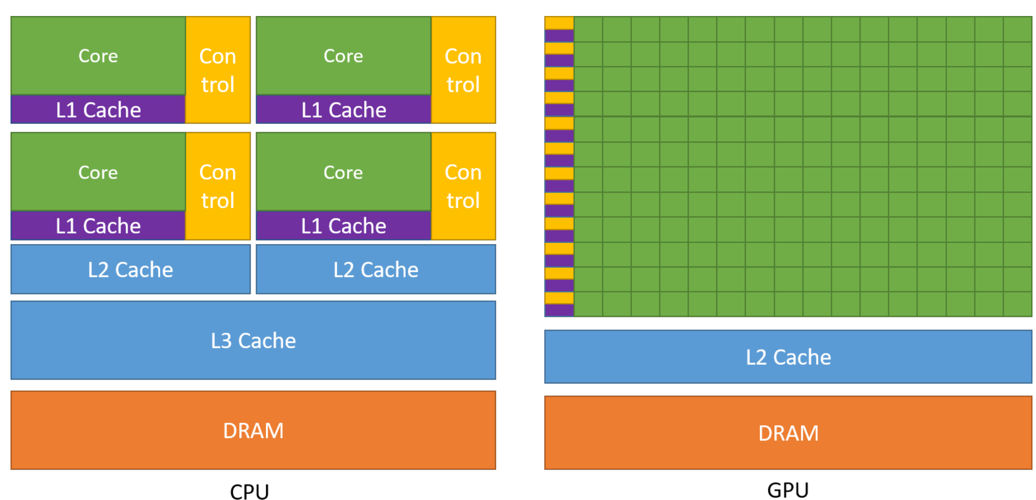


Figura 4.1. Comparação entre uma CPU e uma GPU - [Zone 2019]

DIA – e OpenCL (*Open Computing Language*) [Stone et al. 2010] fornecem maior flexibilidade no gerenciamento do paralelismo enquanto que OpenACC [Farber 2016] e, recentemente, OpenMP (*Open Multi-Processing*) [Chandra et al. 2001], são orientadas a diretivas de compilação permitindo o rápido desenvolvimento. Portanto, programadores de aplicações científicas e multimídia estão ponderando se devem usar CPUs ou GPUs.

Considerando a crescente demanda de desenvolvedores de software no aprendizado de tais interfaces de programação paralela para ambientes heterogêneos, este capítulo aborda os fundamentos de computação acelerada com CUDA C/C++. O material apresentado no decorrer do capítulo é baseado no *workshop* disponibilizado pelo instituto de *deep learning* (DLI) da NVIDIA: *Fundamentals of Accelerated Computing with CUDA C/C++*. Assim, uma breve fundamentação teórica sobre o modelo de programação e arquitetura de GPUs NVIDIA é destacada na Seção 4.2. A Seção 4.3 descreve fundamentos básicos de programação com CUDA, enquanto que as Seções 4.4 e 4.5 discutem técnicas de otimização que podem ser empregadas para acelerar a comunicação entre CPU-GPU e aumentar o nível de paralelismo. Por fim, a Seção 4.6 conclui este capítulo.

4.2. Fundamentação Teórica

GPUs são SIMD (*single instruction, multiple data*) *engines underneath*, onde existe um *pipeline* de instruções que opera como um pipeline SIMD (e.g., um processador vetorial). No entanto, a grande diferença comparado a processadores vetoriais é que o programador não precisa escrever código com instruções vetoriais, mas sim, programar as GPUs usando *threads*, o que é uma grande vantagem na perspectiva de programação. Assim, é possível programar de maneira mais eficiente, pois o programador só precisa se preocupar com a execução de múltiplas *threads* individuais ao invés de se preocupar com detalhes de baixo nível, e.g., *packing* de dados, instruções vetoriais e despachar as instruções no *hardware*.

Existem dois conceitos principais que são relacionados, mas que devem ser distinguidos: *modelo de programação* (*software*), que refere-se a como o programador escreve o código, podendo ser de maneira sequencial (e.g., *von Neuman*), explorando o paralelismo de dados (SIMD), *dataflow* e *multi-threaded*, por exemplo; E, *modelo de execução* (*hardware*), que refere-se em como o *hardware* executa o código, podendo ser de diferentes maneiras, como por exemplo, *out-of-order*, processador vetorial, processador *dataflow*, multiprocessador e processador *multithreaded*. Para melhor entende-los, as seguintes subseções descrevem brevemente as principais características do modelo de programação CUDA e da arquitetura de GPUs NVIDIA.

4.2.1. Modelo de Programação

O paralelismo explorado na GPU pode ser classificado como SPMD (*single program, multiple data*), onde múltiplas *threads* são criadas para executar partes do mesmo código de maneira concorrente, porém, operando sob diferentes dados. Cada *thread* tem seu próprio contexto, podendo ser tratada, reiniciada, executada de maneira independente das outras. Assim, com esta maneira de exploração do paralelismo, é dito que a GPU é chamada de máquina SIMT: *Single instruction, Multiple Threads*. A grande diferença para uma máquina SIMD é que em uma máquina SIMT, o programador não desenvolve o código

usando instruções vetoriais (SIMD) e sim, usando *threads* no modelo de programação SPMD. De uma maneira geral, podemos dizer que GPUs são máquinas SIMD onde o paralelismo no nível de operações vetoriais não é exposto ao programador. Existem duas grandes vantagens do modelo de execução SIMT empregado pelas GPUs: (*i*) Cada *thread* é tratada de maneira separada, o que é visto no processamento MIMD; e (*ii*) múltiplas *threads* podem ser agrupadas em threads de instruções SIMD, isto é, grupos de *threads* que supostamente irão executar a mesma instrução são dinamicamente agrupadas para obter o máximo dos benefícios oferecidos pelo processamento SIMD.

Em um modelo de execução SIMD, um único fluxo sequencial de instruções SIMD são executadas, onde cada instrução específica múltiplas entradas de dados. Por outro lado, no modelo de execução SIMT, múltiplos fluxos de instruções escalares (*threads*) são agrupadas dinamicamente em *threads* de instruções SIMD (*warps*, termo oficial CUDA/NVIDIA). As *threads* de instruções SIMD correspondem a um conjunto de *threads* que estão executando a mesma instrução (i.e., no mesmo PC). Tais *threads* de instruções SIMD são criadas de maneira transparente para o usuário, no sentido de que o programador não precisa se preocupar com esta definição no momento que está explorando o paralelismo no código. Nas arquiteturas atuais da NVIDIA, o tamanho do *warp* é de 32 *threads*, enquanto que em arquiteturas AMD, este número é de 64 *threads*. No entanto, as GPUs não possuem hardware disponível para executar simultaneamente todas as *threads* de instruções SIMD criadas em uma aplicação. Portanto, a GPU aplica *fine-grained multithread*, onde estas *threads* são intercaladas no mesmo *pipeline*.

Existem diferentes interfaces de programação e bibliotecas disponíveis para auxiliar o programador no desenvolvimento de código paralelo para executar na GPU, como por exemplo, CUDA e OpenACC. CUDA fornece um paradigma de codificação que estende linguagens como C, C++, Python e FORTRAN, possibilitando a execução de código massivamente paralelo nas GPUs NVIDIA. CUDA acelera as aplicações com pouco esforço, possuindo um ecossistema de bibliotecas altamente otimizadas para *deep neural networks*, BLAS, análise de grafos, FFT e muito mais, além de fornecer ferramentas de *profiling*. A Seção 4.3 destaca os fundamentos da programação paralela com CUDA, enquanto que as Seções 4.4 e 4.5 discutem características avançadas da interface.

4.2.2. Arquitetura e Organização da GPU

Uma arquitetura GPU é normalmente composta de dois componentes principais: (*i*) Memória Global, que é análoga a RAM em um servidor CPU, sendo acessível por ambos GPU e CPU; e (*ii*) processadores SIMD *multithreaded*, onde a computação é realizada. Estes processadores são chamados de *streaming multiprocessors* (SMs) nas arquiteturas NVIDIA e cada SM tem sua própria unidade de controle, registradores, *pipelines* de execução, *caches*, entre outros componentes de *hardware*. Considerando que a nomenclatura que envolve a discussão de GPUs pode variar de acordo com a bibliografia e empresa (e.g., NVIDIA e AMD utilizam nomenclaturas diferentes para se referir a mesma informação), o resto deste texto considera a nomenclatura utilizada pela NVIDIA.

SMs são processadores de propósito geral porém com uma frequência de operação mais baixa. Um SM é composto basicamente dos seguintes componentes (no entanto, é importante destacar que, a cada nova versão da arquitetura da NVIDIA, diferenças signi-

Figura 4.2. SM da arquitetura Ampere da NVIDIA - [Zone 2019]

ficativas são realizadas nos componentes internos a um SM). Estas diferenças são discutidas abaixo e podem ser acompanhadas na Figura 4.2, que apresenta um SM da arquitetura Ampere da NVIDIA.

- **Núcleos de processamento**: podendo ser sub-divididos em *Cuda cores, Ray-Tracing cores* e *Tensor cores*. Cada GPU NVIDIA contém centenas ou dezenas de *CUDA cores*, onde um *CUDA core* não pode buscar ou decodificar instruções. Ele apenas faz requisições, computa sobre os dados e responde com o resultado. Assim, *CUDA cores* contém unidades de ponto flutuante de precisão simples, dupla, unidades funcionais especiais, unidades lógicas, unidades de *branch*, entre outros componentes. Por outro lado, *Tensor Cores* apresentam computação de multi-precisão para inferência eficiente em inteligência artificial e aprendizado de máquina. Por fim, *Ray-Tracing Cores* são unidades aceleradoras dedicadas para realizar operações de *ray-tracing* e são exclusivos para placas gráficas NVIDIA RTX.

- Milhares de registradores de 32 bits;

- ***Warp schedulers***: unidades de escalonamento e despacho para conjuntos de threads;

- ***Caches***: *Cache L0 de instrução*, encontrada em arquiteturas mais atuais, que é privada a cada bloco de processamento; *L1 data cache*, que é privada a cada SM,

com baixa latência de acesso e alta largura de banda, podendo ser reconfigurada; *Constant cache*, para comunicar as leituras de uma memória somente leitura;

Adicionalmente, externo ao SM, podem ser encontradas as seguintes memórias: *L2 data cache*, que é compartilhada entre todos os SMs da GPU, usada para compartilhar dados, constantes e instruções; e *RAM*, consistindo da memória global.

O número de *warps* que podem ser executadas em paralelo é dependente da quantidade de *CUDA cores* disponíveis na arquitetura. Por exemplo, se existirem 8 *CUDA cores* em cada SM, então as 32 *threads* do *warp* levarão 4 ciclos para concluir a execução (8 *threads* ativas por ciclo). Como esta é uma definição utilizada pelo *hardware*, o número de *warps* não pode ser mudado pelo programador. Assim, para fornecer maior flexibilidade ao programador, a sequencia de operações SIMD (*CUDA threads*) são agrupadas em blocos de *threads* (e no nível de hardware elas são dinamicamente distribuídas em *warps*). Portanto, os programadores podem definir o número de *CUDA threads* por bloco assim como o número de blocos que são criados na chamada da função que irá executar na GPU.

Um bloco de *threads* consiste de uma parte do laço vetorizado que será executado em SMs, composto de um ou mais *warps* onde a comunicação acontece via memória local. Um conjunto de blocos de threads é denominado *Grid* (*loop* vetorizado). Assim, para oferecer mais oportunidades para o *hardware* da GPU gerenciar a execução e explorar o paralelismo disponível, um *grid* é decomposto em múltiplos blocos de *threads*. Um bloco de *thread* é então atribuído pelo escalonador de *warps* ao SM, que executa este código. O programador informa ao escalonador do *warp*, que é implementado em hardware, quantos blocos de *threads* devem ser executados.

Uma vez que SMs são processadores completos com PCs separados, uma GPU pode ter de uma a várias dezenas de SMs. Por exemplo, o sistema Pascal P100 tem 56, enquanto que chips menores podem ter apenas um ou dois. Portanto, para fornecer escalabilidade transparente entre modelos de GPUs com um número diferente de SMs, o escalonador de blocos de *threads* é responsável por atribuir os blocos de *threads* aos SM. Uma vez que o bloco de *threads* foi escalonado para um SM, o escalonador do *warp* sabe quais *warps* estão prontos para executar e os envia para uma unidade de despacho. Assim, a GPU tem dois níveis de escalonadores de *hardware*: (*i*) o *escalonador de blocos de threads* que atribui blocos de threads a SMs e (*ii*) o *escalonador de warps* interno ao SM, que escalona os *warps* quando estiverem prontos para execução.

Como por definição os *warps* são independentes um dos outros, o escalonador de *warps* pode escolher qualquer um que esteja pronto para execução. Para tanto, ele inclui um algoritmo de *scoreboard* para manter informação de até 64 *warps* e decidir quais estão prontos para execução. Uma vez que a latência da memória de instruções é variável devido aos *hits* e *misses* nas *caches* e TLB (*translation lookaside buffer*), uma *scoreboard* é usada para determinar quando as instruções estão completas. A suposição dos projetistas de GPU é que as aplicações que rodam na GPU têm tantos *warps* que o *multithreading* pode ocultar a latência de acesso a DRAM e aumentar a utilização de processadores SIMD *multithreaded*.

4.3. Acelerando Aplicações com CUDA C/C++

4.3.1. Escrevendo Códigos para a GPU

CUDA fornece extensões para as principais linguagens de programação, o que é o caso deste mini-curso, C/C++. Estas extensões da linguagem permitem que os desenvolvedores modifiquem o código fonte para executar em uma GPU. O trecho de código mostrado no Algoritmo 1 contém duas funções além da `main`: `CPUFunction`, que irá executar na CPU; e `GPUFunction`, que irá executar na GPU. Normalmente, o código executado na CPU é referenciado como código do *host*, enquanto que o código que executará na GPU é denominado como código do *dispositivo*. A partir do Algoritmo 1, pode-se destacar algumas características da programação paralela com CUDA:

- A palavra chave `__global__` (linha 6) indica que a respectiva função irá executar na GPU e pode ser invocada globalmente, isto é, pela CPU ou pela GPU. Note que as funções definidas com esta palavra chave devem retornar tipo `void`.
- Ao executar `GPUFunction`$<<<$ *NUM_BLOCOS*, *NUM_THREADS* $>>>$();"(linha 13), dizemos que esta função é um *kernel* que será despachado para execução na GPU. Em conjunto com a chamada do *kernel*, o programador deve fornecer uma configuração de execução, que é feita usando a sintaxe $<<<$ ... $>>>$. Esta configuração permite a especificação da hierarquia de *threads* para a execução do respectivo *kernel*: o primeiro parâmetro informa o número de blocos que serão criados; o segundo parâmetro representa o número de *threads* que serão criadas por bloco. Parâmetros adicionais e opcionais podem ser informados, os quais serão discutidos na Seção 4.5.
- Diferentemente de muitos códigos paralelos escritos em C/C++, o despacho de *kernels* é assíncrono, isto é, o código da CPU continuará executando sem aguardar o término da execução do *kernel* na GPU. Portanto, a função `cudaDeviceSynchronize()` (linha 14), fornecida pelo *runtime* do CUDA, é usada para que o código da CPU aguarde até que o *kernel* finalize sua execução na GPU para então continuar a execução.

Algorithm 1 Primeiro programa em CUDA

```
1:
2:  function VOID CPUFUNCTION(...)
3:      printf("Esta função está definida para executar na CPU.");
4:  end function
5:
6:  function __GLOBAL__ VOID GPUFUNCTION(...)
7:      printf("Esta função está definida para executar na GPU.");
8:      printf("Thread %d do bloco %d.", threadIdx.x, blockIdx.x);
9:  end function
10:
11: function INT MAIN(...)
12:     CPUFunction();
13:     GPUFunction<<<NUM_BLOCOS,NUM_THREADS>>>();
14:     cudaDeviceSynchronize();
15: end function
```

A plataforma CUDA é fornecida com o compilador *nvcc*, que é utilizado para compilar aplicações aceleradas com CUDA através do comando `nvcc -arch=sm_70 -o out app-cuda.cu`. Onde, a *flag* `arch` indica para qual arquitetura o arquivo deve ser compilado de maneira otimizada.

4.3.2. Hierarquia de *Threads* em CUDA

A configuração de execução permite a especificação de detalhes referentes a execução do *kernel* na GPU. Mais precisamente, ela permite ao programador definir quantos grupos de *threads* (i.e., blocos) e quantas *threads* serão criadas em cada bloco. A sintaxe para definição da configuração é a seguinte: `<<<NUM_BLOCOS, NUM_THREADS>>>`. O código do respectivo *kernel* será executado por todas as *threads* criadas no despacho do *kernel*. Assim, o total de *threads* que serão criadas corresponde ao produto do número de blocos e *threads* por bloco.

Cada *thread* criada recebe um índice interno ao seu bloco, iniciando pelo identificador 0. Adicionalmente, cada bloco recebe um índice, iniciando em 0. Da mesma maneira que as *threads* são agrupadas em um bloco, os blocos também são agrupados em uma *grid*, que é a mais alta entidade na hierarquia de *threads* em CUDA. De uma maneira resumida, os *kernels* de um programa CUDA são executados em uma *grid* de um ou mais blocos, cada um deles contendo o mesmo número de uma ou mais *threads*. A Figura 4.3 exemplifica a hierarquia de *Threads* em CUDA. Uma CUDA *Thread* é executada em um *CUDA Core*. Um bloco de CUDA *threads* é atribuído para execução por um SM. E, um *kernel* é encaminhado para execução em uma GPU. Durante a execução de um *kernel*, as *threads* têm acesso a variáveis especiais para identificar os seus índices interno ao bloco (`threadIdx.x`) e o índice do bloco na *grid* (`blockIdx.x`), conforme descrito no Algoritmo 1.

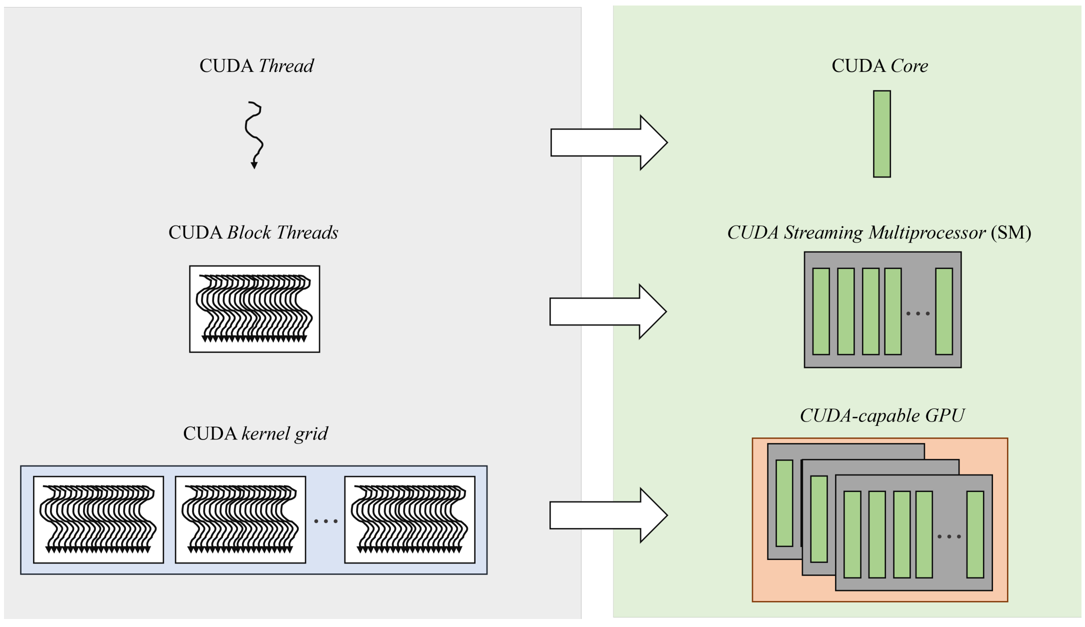


Figura 4.3. Execução de um Kernel na GPU - [Zone 2019]

Ao programar com CUDA, existe um limite de 1024 *threads* que podem ser criadas em um bloco. Assim, para aumentar o grau de exploração de paralelismo, há a necessidade de coordenar a execução de múltiplos blocos de *threads*. Para tanto, as *threads* executando um *kernel* escrito em CUDA têm acesso a variável especial `blockDim.x`, que fornece o número total de *threads* em um bloco. Esta variável pode ser usada em conjunto com as descritas anteriormente para aumentar o grau de paralelismo via a organização da execução paralela entre múltiplos blocos de múltiplas *threads* com a expressão: `threadIdx.x + blockIdx.x * blockDim.x`.

4.3.3. Alocando Memória Unificada para ser Acessada na GPU e CPU

As versões mais recentes de CUDA (a partir da 6.0, inclusive) facilitam a alocação de memória disponível tanto para a CPU quanto para a GPU através da memória unificada (*unified memory* - UM) [Chien et al. 2019]. Embora existam diversas técnicas intermediárias e avançadas para o gerenciamento de memória que podem fornecer desempenho ótimo, a técnica básica de gerenciamento de memória em CUDA é capaz de fornecer desempenho satisfatório em muitas aplicações com quase nenhuma sobrecarga de programação ao desenvolvedor. Assim, para alocar e liberar memória, além de obter um ponteiro que pode ser referenciado tanto no código do *host* quanto no dispositivo, as funções `cudaMallocManaged` e `cudaFree` são utilizadas, respectivamente. Um exemplo da aplicabilidade destas duas funções pode ser observado no Algoritmo 2 na utilização dos vetores `a` e `b`. A partir da alocação de memória, nas linhas 13 e 14, as duas variáveis podem ser acessadas tanto na CPU quanto na GPU, sem a necessidade do uso de funções específicas para transferência de dados entre a CPU e GPU (e.g., `cudaMemCpy`). Por fim, a liberação de memória ocorre nas linhas 19 e 20.

Em palavras simples, a UM fornece ao usuário a visão de um único espaço de memória que é acessível por todas as GPUs e CPUs no sistema, conforme ilustrado na Figura 4.4. Quando a memória unificada é alocada, os dados ainda não residem no dispositivo nem no *host*. Quando um deles tentar acessar tais dados, ocorrerá um *page fault*, momento em que o *host* ou dispositivo irá migrar os dados necessários em lotes. De modo similar, a qualquer momento em que a CPU ou GPU tentar acessar uma posição de memória que ainda não reside nela, *page faults* irão ocorrer, acionando a migração dos dados.

A capacidade de migrar os dados da memória sob demanda é extremamente útil para facilitar o desenvolvimento de aplicações. Além disso, ao trabalhar com dados que exibem padrões de acesso esparsos, por exemplo, quando não é possível saber quais dados devem ser trabalhados até que a aplicação seja executada e para cenários em que os dados

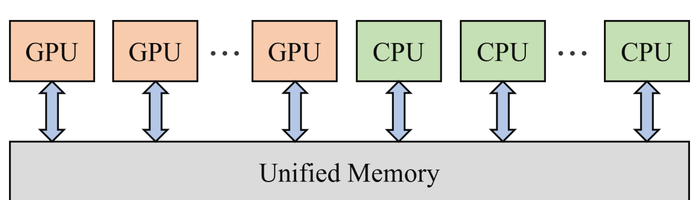


Figura 4.4. Diagrama de Memória Unificada - [Han and Sharma 2019]

podem ser acessados por várias GPUs em um sistema multi-GPU, a migração de memória sob demanda é extremamente benéfica. Por outro lado, em aplicações onde os dados necessários são conhecidos a priori e onde grandes blocos contíguos de memória são necessários, o sobrecusto do *page-fault* e migração de dados sob demanda incorre em um alto custo no desempenho que é melhor evitar. Este assunto é discutido com maior profundidade na Seção 4.4.

4.3.4. Acelerando Estruturas de Repetição

Estruturas de repetição representadas através de laços `for` em aplicações otimizadas para a CPU normalmente estão prontas para serem aceleradas na GPU: em vez de executar cada iteração do laço de maneira serial, uma após a outra, cada iteração do laço pode ser executada em paralelo por sua própria *thread*. Assim, para paralelizar um laço de repetição com CUDA, duas etapas devem ser seguidas: (*i*) re-escrever o *kernel* para fazer o trabalho de uma única iteração do laço; e (*ii*) como o *kernel* será independente de outros *kernels* em execução, a configuração de execução deve garantir que o *kernel* execute o número correto de vezes, por exemplo, o número de vezes que o laço de repetição deveria iterar.

Dependendo da aplicação que está sendo paralelizada, a configuração de execução pode não ser capaz de criar o número exato de *threads* necessários para a paralelização do laço. Considere o seguinte cenário como exemplo: a paralelização de um laço de repetição com 1000 iterações. Conforme mencionado anteriormente, blocos que possuem um número múltiplo de 32 *threads* tendem a apresentar melhor desempenho devido ao tamanho do *warp* e da maneira como estes são escalonados nos SMs. Assumindo que queremos despachar blocos com 256 *threads* (múltiplo de 32), não existirá um número de

Algorithm 2 Acelerando um laço de repetição em CUDA

```
 1:
 2: function __GLOBAL__ VOID MULTIPLICA(int valor, int *a, int *b, int N)
 3:    int id = threadIdx.x + blockIdx.x * blockDim.x;
 4:    int gridStride = gridDim.x * blockDim.x;
 5:    for (int i = id; i < N; i += gridStride) do
 6:       a[i] = valor * b[i];
 7:    end for
 8: end function
 9:
10: function INT MAIN(...)
11:    int N = 10000, *a, *b, valor = 5;
12:    size_t size = N * sizeof(int);
13:    cudaMallocManaged(&a, size);
14:    cudaMallocManaged(&b, size);
15:    size_t num_threads = 256;
16:    size_t num_blocks = 32;
17:    multiplica<<<num_blocos, num_threads>>>(valor, a, b, N);
18:    cudaDeviceSynchronize();
19:    cudaFree(a);
20:    cudaFree(b);
21:    return 0;
22: end function
```

blocos que produzirá o total exato de 1000 *threads* na *grid* (e.g., 4 blocos resultarão no total de 1024 *threads*).

Este cenário pode ser tratado da seguinte maneira:

1. Escrever uma configuração de execução que cria mais *threads* do que o necessário para computar o *kernel*.

2. Passar um valor `N` como um argumento para o *kernel*, que representa o tamanho total dos dados que serão processados, ou o total de *threads* que são necessárias para finalizar o trabalho.

3. Após, calcular o índice de cada *thread* na *grid* (usando `threadIdx.x + blockIdx.x * blockDim.x`), e apenas realizar o trabalho se o índice não exceder `N`.

De maneira similar, há situações em que o número total de *threads* criadas para executar um *kernel* é menor do que o tamanho do conjunto de dados. Como um simples exemplo, considere um vetor com 1000 elementos e uma *grid* com 250 *threads*. Neste cenário, cada *thread* na *grid* deverá ser usada quatro vezes para garantir a execução correta do *kernel*. Para tratar tal cenário, o método de *grid-stride loop* é utilizado dentro do *kernel*.

Em um *grid-stride loop*, cada *thread* irá: (*i*) calcular com seu identificador único dentro da *grid*; (*ii*) realizar a operação no elemento representado pelo índice da *thread*; e (*iii*) adicionar a este índice o número total de *threads* que existem na *grid* e repetir até que o índice esteja fora do intervalo do vetor. O Algoritmo 2 destaca este cenário na função `multiplica`: na linha 3, calcula-se o índice de cada *thread* na *grid*; na linha 4, obtém-se o deslocamento do *grid-stride loop*, o qual será utilizado como incremento no laço de repetição da linha 5.

4.3.5. Tratamento de Erros

Como em qualquer aplicação, o tratamento de erros em códigos implementados com CUDA é essencial. A grande maioria das funções (e.g., funções de gerenciamento de

Algorithm 3 Tratamento de Erros

```
 1: ...
 2: cudaError_t err1, err2;
 3: err1 = cudaMallocManaged(&a, size);
 4: if (err1 != cudaSuccess) then
 5:    printf("Error:  %s", cudaGetErrorString(err1));
 6: end if
 7:
 8: GPUFunction<<<1, 1>>>();
 9: err2 = cudaGetLastError();
10: if (err2 != cudaSuccess) then
11:    printf("Error:  %s", cudaGetErrorString(err2));
12: end if
13: cudaDeviceSynchronize();
14: ...
```

memória) retorna um valor do tipo `cudaError_t`, que pode ser utilizado para verificar a ocorrência de erro durante a execução da função. Um exemplo deste cenário é demonstrado no Algoritmo 3 para a execução da função `cudaMallocManaged`, na linha 3. O retorno da função é então armazenado na variável `err1` para então ser verificada na linha 4. Caso a função não tenha sido executada corretamente, a função `cudaGetErrorString` é utilizada para obter informações do erro.

Um outro cenário ocorre quando a função a ser executada é do tipo `void`, a qual não retorna um valor do tipo `cudaGetLastError`. Este cenário, descrito nas linha 8-12 do Algoritmo 3, acontece no despacho dos *kernels* que irão executar na GPU. Assim, para verificar se aconteceu algum erro durante o despacho do *kernel*, por exemplo, CUDA fornece a função `cudaGetLastError`, que é utilizada para obter informações do último erro que ocorreu durante a execução.

4.3.6. Obtendo Informações da GPU

Conforme destacado na Seção 1.2, as GPUs na qual as aplicações CUDA executam possuem unidades de processamento chamadas de *streaming multiprocessors*, ou SMs. Durante a execução do *kernel*, os blocos de *threads* são alocados nos SMs para execução. Assim, para dar suporte à capacidade da GPU de executar o maior número possível de operações em paralelo, os maiores ganhos de desempenho normalmente podem ser obtidos escolhendo um tamanho de *grid* que tenha um número de blocos múltiplo do número de SMs. Além disso, os SMs criam, gerenciam, escalonam e, executam grupos de 32 *threads* dentro de um bloco (*warp*). Assim, é importante destacar que escolher um tamanho de bloco que seja múltiplo de 32 tende a fornecer um desempenho melhor para a aplicação.

Neste sentido, uma vez que o número de SMs pode variar de acordo com a GPU que está sendo usada e para fornecer portabilidade às aplicações, o número de SMs e outras informações úteis podem ser obtidas através de uma *struct* em C. O trecho de código no Algoritmo 4 destaca a obtenção de algumas propriedades da GPU: as linhas 2 e 3 são responsáveis por obter o identificador da GPU no sistema; nas linhas 5 e 6, a *struct* `props` é declarada e utilizada para receber as propriedades da respectiva GPU. Por fim, nas linhas 8 a 11 são destacadas algumas propriedades, como por exemplo a capacidade computacional da GPU, o número de SMs e o tamanho do *warp*. A lista completa de

Algorithm 4 Obtendo informações da GPU

```
 1: ...
 2: int deviceId;
 3: cudaGetDevice(&deviceId);
 4:
 5: cudaDeviceProp props;
 6: cudaGetDeviceProperties(&props, deviceId)
 7:
 8: int computeCapabilityMajor = props.major;
 9: int computeCapabilityMinor = props.minor;
10: int numberSMs = props.multiProcessoCount;
11: int warpSize = props.warpSize;
12: ...
```

propriedades pode ser obtida diretamente na documentação do CUDA [1].

4.4. Otimizando a Troca de Dados entre CPU e GPU

Quando a memória unificada é utilizada, a GPU pode acessar qualquer página da memória do sistema e, ao mesmo tempo, migrar os dados sob demanda para sua própria memória para acesso de alta largura de banda. No entanto, para obter o melhor desempenho possível com a memória unificada, é importante entender como funciona a migração de página sob demanda. Para tanto, considere o pseudocódigo descrito no Algoritmo 2. Nele, os dados dos vetores `a` e `b` são inicializados na CPU (função omitida do algoritmo por simplicidade). Então, os dados são migrados da memória da CPU para a GPU e acessados na GPU durante a computação do *kernel*.

No entanto, trabalhos da literatura têm mostrado que a memória unificada introduz um complexo mecanismo para tratamento de *page-fault* [Landaverde et al. 2014, Chien et al. 2019, Knap and Czarnul 2019]. Além disso, o *trashing* de memória que move as mesmas páginas entre as memórias se torna um gargalo no desempenho. A natureza da GPU em explorar o paralelismo massivo de dados exacerba ainda mais a sobrecarga de *page-fault*, uma vez que os processos param quando um *page fault* está sendo resolvido e múltiplas *threads* em diferentes *warps* acessando a mesma página podem causar várias falhas duplicadas.

Desta maneira, a interface CUDA introduziu um mecanismo de pré-busca assíncrona (*assynchronous prefetching*) de página. Esta técnica poderosa pode ser utilizada para reduzir o *overhead* das migrações de memória sob demanda entre CPU-GPU e GPU-CPU quando ocorrer um *page-fault*. Ao utilizar este mecanismo (i.e., `cudaMemPrefetch-Async()`, a migração de dados ocorre em segundo plano através de um *stream* CUDA para evitar a interrupção das *threads* que estão computando. Ao fazer isso, o desempenho dos *kernels* da GPU e funções da CPU pode ser melhorado devido à redução de *page fault* e consequente *overhead* da migração de dados sob demanda.

Um exemplo da aplicabilidade da pré-busca assíncrona de página é destacado no Algoritmo 5 para a computação dos vetores `a` e `out` na GPU. Inicialmente, os dados devem ser corretamente alocados através da função `cudaMallocManaged` (linhas 10 e 11). Então, para cada um dos vetores, deve-se adicionar uma chamada à função `cudaMemPrefetchAsync` para habilitar a pré-busca assíncrona de páginas. Esta função é chamada com os seguintes parâmetros nas linhas 12 e 13 para os vetores `a` e `out`, respectivamente: o primeiro parâmetro é a variável alvo da migração; o segundo parâmetro consiste no tamanho da estrutura que será migrada; por fim, o identificador do dispositivo GPU ativo é indicado.

Com os dados sendo migrados em segundo plano para a memória da GPU, no momento que o *kernel* `GPUFunction` for despachado para execução, os dados necessários para execução já estarão presentes na GPU. O respectivo *kernel* (omitido por simplicidade) irá produzir como saída o vetor `out`, que será então utilizado pela CPU. Assim, para otimizar a cópia dos dados do vetor `out` para a memória da CPU através da pré-busca assíncrona, a função `cudaMemPrefetchAsync` é inserida na linha 21. Cabe notar

[1] https://docs.nvidia.com/cuda/cuda-runtime-api/structcudaDeviceProp.htmlstructcudaDeviceProp

que o último parâmetro contém a variável disponível na interface do CUDA que define o dispositivo CPU como sendo o destino da migração dos dados (`cudaCpuDeviceId`).

4.5. Sobrepondo Computação com Comunicação através de CUDA *Streams*

Em CUDA, um *stream* é definido como uma série de comandos que serão executados em ordem. Diversas operações, como por exemplo, execução de *kernel* e algumas transferências de dados ocorrem em CUDA *streams*. Embora até este momento o texto não tenha entrado no mérito de *streams*, os códigos descritos anteriormente executam seus *kernels* interno ao *default stream*.

Um exemplo da execução de vários *kernels* é ilustrado na Figura 4.5. Quando o programador não define explicitamente um *stream*, o *default* será utilizado. Nele, nenhuma operação irá iniciar até que todas as operações despachadas anteriormente no dispositivo sejam concluídas. Além disso, qualquer operação no *default stream* deve ser concluída antes de qualquer outra operação começar. Assim, este modo não permite a sobreposição de computação com comunicação, limitando os potenciais ganhos de desempenho.

Neste sentido, para aumentar o nível de concorrência durante a execução de uma aplicação CUDA na GPU, os programadores CUDA podem criar e utilizar *non-default CUDA streams* em adição ao *default stream*. Ao fazer isto, múltiplas operações podem

Algorithm 5 Exemplo da pré-busca assíncrona de memória

```
 1: function INT MAIN(...)
 2:    int N = 1000, *a, *out, deviceId, numberSMs;
 3:    size_t size = N * sizeof(int);
 4:
 5:    cudaGetDevice(&deviceId);
 6:    cudaDeviceProp props;
 7:    cudaGetDeviceProperties(&props, deviceId);
 8:    numberSMs = props.multiProcessorCount;
 9:
10:    cudaMallocManaged(&a, size);
11:    cudaMallocManaged(&out, size);
12:    cudaMemPrefetchAsync(a, size, deviceId);
13:    cudaMemPrefetchAsync(out, size, deviceId);
14:
15:    size_t num_threads = 256;
16:    size_t num_blocos = 32 * numberSMs;
17:
18:    GPUfunction<<<num_blocos, num_threads>>>(a, out, ...)
19:
20:    cudaDeviceSynchronize();
21:    cudaMemPrefetchAsync(out, size, cudaCpuDeviceId);
22:
23:    checkElements(c,N);
24:
25:    cudaFree(a);
26:    cudaFree(out);
27: end function
```

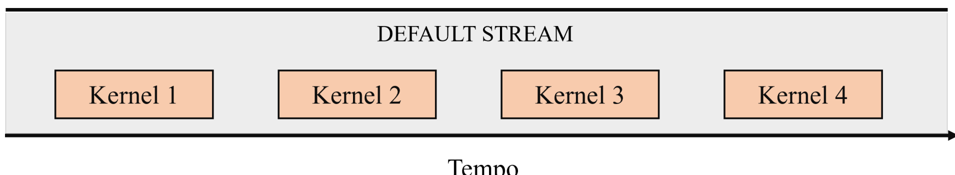


Figura 4.5. Comportamento da execução de diferentes *kernels* no *default stream*

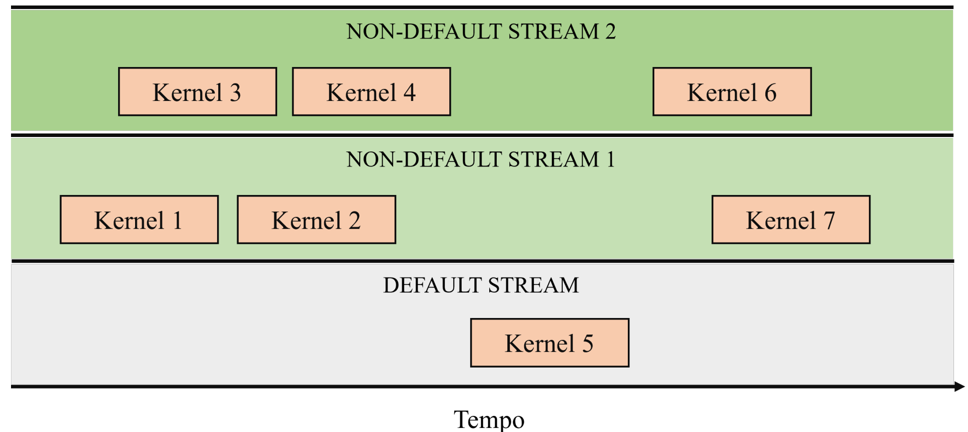


Figura 4.6. Explorando concorrência no nível de *streams*

ser realizadas de maneira concorrente, em *streams* diferentes, conforme ilustrado na Figura 4.6. Usando múltiplos *streams* pode adicionar uma camada extra de paralelização à aplicação, e oferecer mais oportunidades para otimização da aplicação. Existem algumas regras considerando o comportamento de CUDA *streams* que devem ser aplicados de modo a utiliza-los eficientemente:

- As operações realizadas internas a um *stream* ocorrem em ordem.
- Não existe a garantia de que as operações em diferentes *non-default streams* seja realizada em qualquer ordem específica relativa a cada uma.
- O *default stream* é bloqueante e irá aguardar todos os outros *streams* completar a execução antes de iniciar, e, irá bloquear a execução de outros *streams* até que seja finalizada sua execução.

Considerando que as operações em diferentes *streams* podem executar de maneira concorrente e ser intercaladas, o desempenho de uma aplicação pode ser melhorado de maneira significativa. Para exemplificar esta situação, considere a Figura 4.7 que destaca três operações: cópia assíncrona de dados do *host* para o dispositivo; a execução do *kernel* na GPU; e a cópia dos dados do dispositivo para o *host*. Quando apenas o *default stream* é utilizado (serial), não existe concorrência entre as operações. Ao adicionar múltiplos *non-default streams*, a concorrência pode ser explorada de diferentes maneiras. Na concorrência *2-way*, duas operações irão ocorrer no mesmo instante de tempo, como por exemplo, a execução de um *kernel* em um *non-default stream* sobreposta com

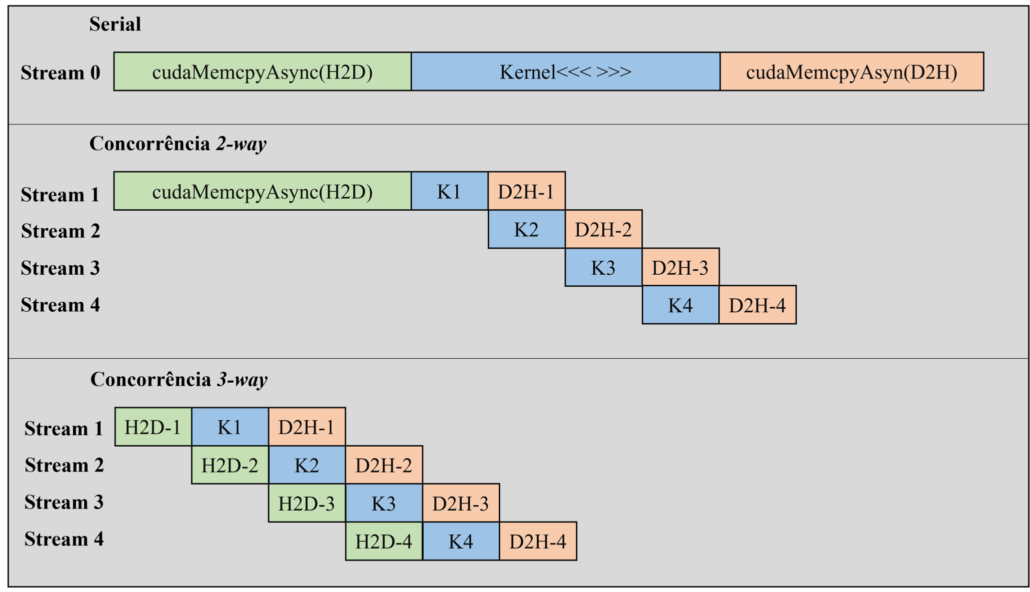


Figura 4.7. Quantidade de concorrência em múltiplos *streams*

a comunicação do dispositivo para o *host* em outro *stream*. Ao adicionar um novo nível de concorrência, (*3-way*), a cópia de dados do *host* para o dispositivo também passa a ser realizada de maneira sobreposta a computação.

Um exemplo da aplicabilidade de múltiplos *streams* em uma aplicação é destacado no Algoritmo 6. Inicialmente, um *stream* deve ser declarado com o tipo `cudaStream_t`. A criação de um *stream* de maneira assíncrona se dá através da função `cudaStream-Create`, que recebe como parâmetro um ponteiro para o identificador do *stream*. Então, para garantir que o despacho do *kernel* seja realizado em diferentes *streams*, o quarto parâmetro de configuração de execução deve conter o identificador do respectivo *stream*. Isto irá lançar o *kernel* para ser executado num *non-default stream*. O terceiro argumento da configuração de execução permite o programador fornecer o número de *bytes* na memória compartilhada que serão dinamicamente alocados por bloco para o respectivo *kernel*. Para deixar o número padrão de *bytes* alocado por bloco na memória compartilhada, basta incluir o valor 0 neste argumento. Por fim, o *stream* é finalizado através da função `cudaStreamDestroy`.

Algorithm 6 Explorando múltiplos *streams*

```
1:  ...
2:  for (int i = 0; i < totalStream; i++) do
3:     cudaStream_t stream;
4:     cudaStreamCreate(&stream);
5:
6:     GPUFunction<<<num_blocos, num_threads, 0, stream>>>();
7:
8:     cudaStreamDestroy(stream);
9:  end for
10: ...
```

4.6. Conclusão

Considerando o avanço iminente das arquiteturas GPUs em sistemas de alto desempenho, *desktops* e até mesmo em dispositivos móveis, se torna extremamente importante o aprendizado de técnicas para tirar o maior proveito possível de tais dispositivos. Neste sentido, este capítulo de livro apresentou fundamentos relacionados a programação massivamente paralela com CUDA, cobrindo aspectos principais, como por exemplo, exploração do paralelismo em laços, otimização da troca de dados entre CPU-GPU, sobreposição de comunicação e computação para melhorar a concorrência e a utilização de múltiplos *streams* para elevar o nível de exploração de paralelismo. Por fim, recomenda-se a leitura de material fornecido pela NVIDIA em seus *blogs*, zona do desenvolvedor e documentação do CUDA para enriquecer ainda mais seus conhecimentos em programação paralela em ambientes heterogêneos.

Referências

[Chandra et al. 2001] Chandra, R., Dagum, L., Kohr, D., Menon, R., Maydan, D., and McDonald, J. (2001). *Parallel programming in OpenMP*. Morgan kaufmann.

[Chien et al. 2019] Chien, S., Peng, I., and Markidis, S. (2019). Performance evaluation of advanced features in cuda unified memory. In *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*, pages 50–57. IEEE.

[Farber 2016] Farber, R. (2016). *Parallel Programming with OpenACC*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition.

[Han and Sharma 2019] Han, J. and Sharma, B. (2019). *Learn CUDA Programming: A beginner's guide to GPU programming and parallel computing with CUDA 10.x and C/C++*. Packt Publishing.

[Knap and Czarnul 2019] Knap, M. and Czarnul, P. (2019). Performance evaluation of unified memory with prefetching and oversubscription for selected parallel cuda applications on nvidia pascal and volta gpus. *The Journal of Supercomputing*, 75(11):7625–7645.

[Knorst et al. 2021] Knorst, T., Jordan, M. G., Lorenzon, A. F., Rutzig, M. B., and Beck, A. C. S. (2021). Etcf – energy-aware cpu thread throttling and workload balancing framework for cpu-fpga collaborative environments. In *2021 XI Brazilian Symposium on Computing Systems Engineering (SBESC)*, pages 1–8.

[Landaverde et al. 2014] Landaverde, R., Zhang, T., Coskun, A. K., and Herbordt, M. (2014). An investigation of unified memory access performance in cuda. In *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE.

[Navarro et al. 2020] Navarro, A., Lorenzon, A. F., Ayguadé, E., and Beltran, V. (2020). Enhancing resource management through prediction-based policies. In Malawski, M. and Rzadca, K., editors, *Euro-Par 2020: Parallel Processing*, pages 493–509, Cham. Springer International Publishing.

[Sanders and Kandrot 2010] Sanders, J. and Kandrot, E. (2010). *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional.

[Stone et al. 2010] Stone, J. E., Gohara, D., and Shi, G. (2010). Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in Science and Engg.*, 12(3):66–73.

[Zone 2019] Zone, N. D. (2019). Cuda toolkit documentation. *NVIDIA,[Online]. Available: http://docs. nvidia. com/cuda/cuda-c-best-practices-guide/# occupancy.[Acedido em Fevereiro 2015].*

Capítulo 5

From Sequence Assembly to Ancestry Testing: HPC Challenges for Bioinformatics

Mariana Carmin (UFPR), Cláudio Torres Júnior (UFPR), André Ricardo Abed Grégio (UFPR) e Marco Antonio Zanata Alves (UFPR)

Abstract

The bioinformatics field makes several types of analyses possible, such as verifying susceptibility to certain diseases and identifying samples in forensic science. Despite the growing use, the problems solved with bioinformatics techniques are still limited by the processing time and memory storage capacity, since the algorithms used are complex, often of quadratic or cubic orders, and may rely on large volumes of data. Therefore, it is necessary to have a correct understanding of the basic functioning of those techniques and the limitations and main challenges related to biological problems. As a contribution, in this chapter, we aim to provide an introductory view of embracing the collection and analysis of biological data. To accomplish that goal, we will discuss the main algorithms and databases and how to improve the algorithms applied to bioinformatics problems.

5.1. Introduction

Computers are powerful tools to help in solving today's biological problems. Most of these problems involve a huge amount of data. Commonly, this data comes in the form of images [2] or, as we will discuss in this Chapter, biological problems also make use of data from biological sequences [1].

When the problem involves working with images, those may be pictures or another data format used to construct images (e.g., magnetic resonance imaging (MRI) scan or recreation of the 3D protein structure[5]). For example, Klaczko and Blanche [3] calculate the size and shape of D. mediopunctata's wing using a photography of the microscope slide. To do so, they adjust an ellipse to fit the wing contour, and by using the ellipse parameters, they can calculate the size and shape of the wing. This paper can be applied to studies aiming to analyze the impacts of the selection process, evolution, and environmental genetic factors, and how these aspects influence Drosophila's wings [3].

Another widely studied field is predicting 3D protein structure. The multiple struc-

ture alignment (MSA) can be used to understand the conserved and divergent areas during evolution and to predict the function of the protein analyzed [5].

When we analyze the problems that use a biological sequence as data, the main focus of this chapter, we face many problems—from plague control to genetic disease prevention. We provide some specific examples below.

The Moniliophthora perniciosa causes the witches' broom disease (WBD) in cocoa and is an example of the use of bioinformatics applied to a real problem. In Brazil, there have been severe cases of WBD, in which phyto-sanitation practices and resistance breeding programs were applied to control the disease. However, these methods alone were not enough to solve the problem. In addition, the largest producers of cacao in the world are Africa and Asia, and if WBD reaches these areas, the worldwide production of chocolate could be affected [40]. Thus, sequencing the Moniliophthora perniciosa genome helped develop better control strategies for WBD.

Genome studies are also applied to humans. The Human Genome Project is the most significant collaborative work, which started in 1989 and ended in 2003. It encompasses scientists worldwide with a shared mission: to sequence all the human genome. Therefore, it is essential to notice the complexity associated with this task, as the human genome contains more than 3 billion base pairs (bp). Nevertheless, these complex steps are essential to advance genetic researches since knowing our genome code could help us understand how we function at the chemical level. Also, it could explain the role of genetic factors in many diseases, such as cancer, Alzheimer's disease, and schizophrenia, all of which decrease the lifespan of millions of people [41].

The human genome made it possible to perform another analysis. Another example is considering sequence variation, in which we can use this information for understating genetic diseases or investigate the predisposition for some disease or condition, not intending to act as treatment but prevention before any symptoms appear [2].

Furthermore, the information from sequence variation can be used to recognize the different effects/side effects of drugs in individuals, since they can vary from person to person. Therefore, this information can be used to develop more effective pharmaceuticals for everyone or a personalized drug for each patient according to the observed genomic sequence [2].

Nowadays, we cannot talk about bioinformatics research without considering the computation behind it. However, more than that, we also observe a quick adoption of computational analyses in the medical area [1].

Extracting scientific information from biological sequences is known as bioinformatics, by biological sequences we consider deoxyribonucleic acid (DNA), ribonucleic acid (RNA), or protein sequences. These sequences represent a vast amount of data, making it almost impossible to analyze them without a computer [2]. Furthermore, the more complex data lead to a necessity to improve the performance of the tools used.

For that reason, this chapter aims to explore the main areas in bioinformatics and describe the most famous algorithms and how we can improve their performance.

This chapter is organized as follows. First, section 5.2 describes the biological

information flow, from the DNA molecule to the sequence that can be processed and stored in a computer. Next, section 5.3 describes the main steps in bioinformatics, error correction, sequence assembly, and analyses. Next, Section 5.4 describes in detail the sequence alignment process, and finally, Section 5.5 describe the remains challenges in the area.

5.2. Biological information flow

DNA is the basis of all the genetic information used to produce everything necessary for an organism [21]. It's formed by a composition of nucleotides that is composed by a pentose (5-carbon) sugar, a phosphate group and a nitrogen base, which can be of four different types: **Adenine (A)**, **Cytosine (C)**, **Guanine (G)** and **Thymine (T)**. However, the DNA molecule is not an alone strand. Each molecule is formed by two strands forming a 3D spiral structure like a double helix, as seen in Figure 5.1. The strands connect to each other in a specific order: **A's** bases only pairs up with **T's**, and **C's** with **G's**.

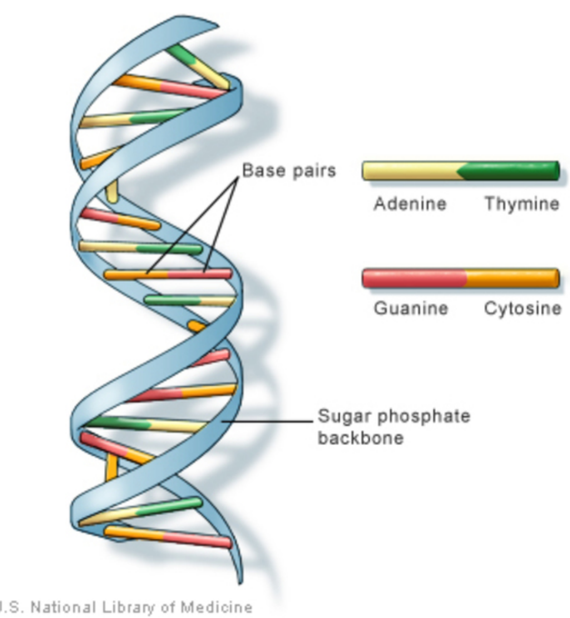


Figure 5.1. DNA molecule. Source: U.S. National Library of Medicine

Before starting the work with the DNA, we need to extract this molecule from the organism that will be studied. For this, many methods can be used and knowing the suitable one can be time and money-saving in the future steps of the experiment [11]. Different studies can have different purposes, so different tissues may need specific preparation methods. However, they share standard steps, like Efficient DNA extraction from the organism, creating necessary samples for the experiment to be performed, cleaning contaminants, and separating the DNA strands with higher quality [12].

When the DNA is ready and processed, the goal is to determine the ordered nitrogen bases from that sequence. For that reason, many technologies have emerged to carry out these processes and allow the realization of the most diverse experiments.

DNA was not the first molecule to be sequenced. In the early 50's, Frederick Sanger found out the constituent elements of insulin protein [22]. After that, many improvements and new methods were created and improved. Finally, in the mid of 70's, Sanger was able to sequence a whole DNA genome, the first one in history. The method used is considered the initial step to the first generation of sequencers [21].

In short, the double strand needs to be separated when a DNA molecule is inside the cell and needs to be duplicated (replication step). The Figure 5.2 show the steps:

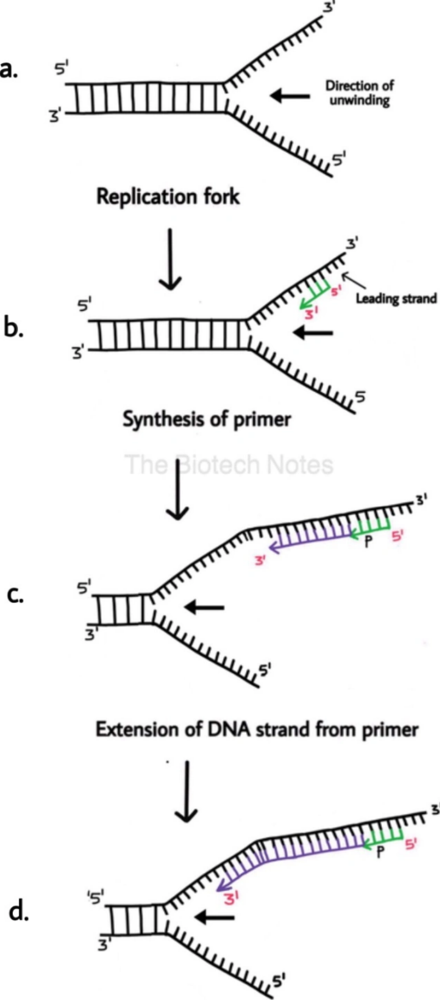


Figure 5.2. DNA replication. Credit: The Biotech Notes

a) An enzyme called *DNA helicase* breaks the bonds between bases.

b) When a portion is separated, a small sequence called *primer* binds to the start of the main strand.

c) The enzyme *DNA polymerase* creates the new strand by binding to the *primer* and extending the sequence with the corresponding base (if the base in the main strand is **A**, the incorporated base will be **T**).

d) In the end, the generated strand will be identical to the separated initial strand (the other strand goes through the same process but with a few more steps because it is read in reverse).

This is the basic principle used in most sequencers.

In 1977 Sanger improved his technique and developed the known Sanger Sequencing, as shown in Figure 5.3:

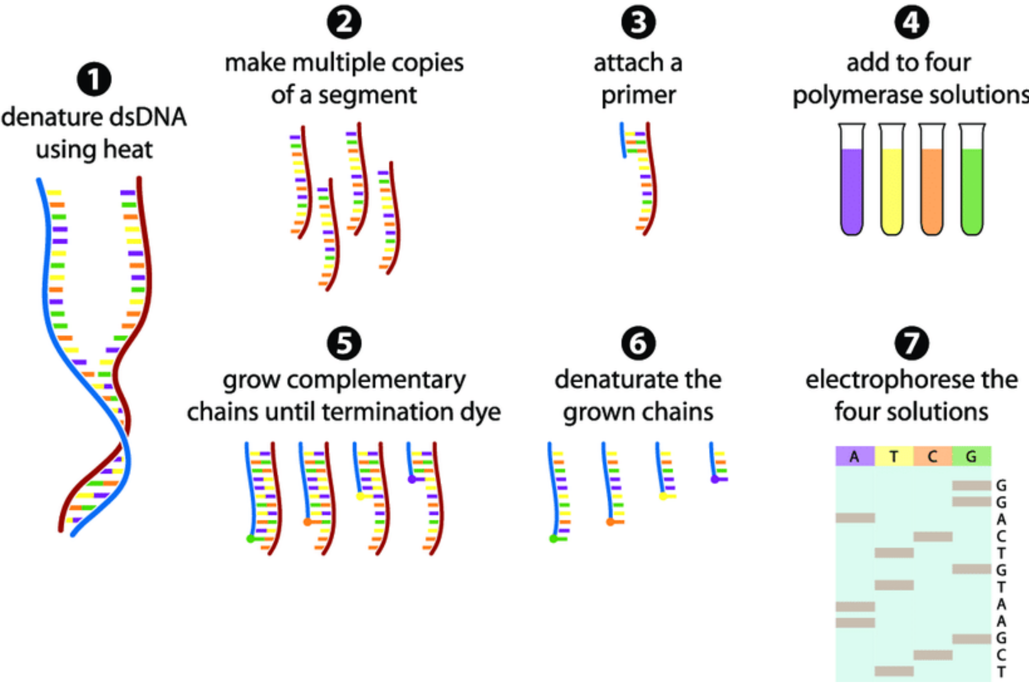


Figure 5.3. Sanger Sequencing [25]

1. Separate the double strand using heat.

2. For each strand (or segment of it) that will be sequenced make multiple copies.

3. Bind a primer to each segment.

4. Prepare four solutions containing enzymes *DNA polymerases* and all four nucleotides. The only difference is that one type of modified nucleotide is added in each solution. In this example, some **A** modified nucleotides were added in purple solution, and in yellow was added some **T**. This nucleotide is called dideoxynucleotide and its function is to block the *DNA polymerase* to continue extension.

5. The *DNA polymerases* binds to the *primer* and starts the replication. Normal nucleotides are added to the segment, and when a modified one is added, the process stops, leaving sequences of different sizes. In Figure 5.4, we have a demonstration of what happens in each solution (the colors do not represent the bases in Figure 5.3). Getting the **G** dideoxynucleotide, every time it's added to the sequence

A
- A
- ATGCA
- ATGCAGCGTTA
- ATGCAGCGTTACCA

C
- ATGC
- ATGCAGC
- ATGCAGCGTTAC
- ATGCAGCGTTACC

G
- ATG
- ATGCAG
- ATGCAGCG
- ATGCAGCGTTACCATG

T
- AT
- ATGCAGCGT
- ATGCAGCGTT
- ATGCAGCGTTACCAT

Figure 5.4. Sanger step 5 example - DNA polymerases prevention extension. Adapted from [21]

end, the *DNA polymerases* stops. So we have in this example four different sequences with different lengths finished with **Guanine**.

6. All sequences are separated again, and we use the constructed strand from the previous step, which we call fragments.

7. The fragments pass through a polyacrylamide gel via electrophoresis technique. The solutions are placed in an equipment, and an electric current is applied to each one making the fragments travel by the gel pores. The smaller the sequences, the faster they go, traveling long distances in the gel. From bottom to top, the last gray box represents the smallest sequence that has **Thymine** as a terminal base, and then comes a sequence ending with **Cytosine**. Because **Thymine** terminal sequence went further, it is assumed that its length is smaller than **Cytosine** one. Therefore in the original sequence, the base **T** comes before **C**. So here, the sequence can be inferred as *TCGAATGTCAGG*, being the complementary sequence of the DNA sample of step 2.

As said, Sanger Sequencing was the starting point of the First Generation of sequencers. The result from all these steps produces a sequence called a read, with at most 1000 bases (one kilobase - kb) [21]. This technique was widely used, but the need to sequence larger genomes began to emerge. An example was the Human Genome Project, which had the objective sequence of approximately 3 billion base pairs from the human

genome with the collaboration of a worldwide team, having an estimated cost of one dollar per base [29]. The project was proposed in 1984 but only started in 1990, finishing on April 14, 2003 [29]. Finished but not completed. Only 92% of the human genome was sequenced because of the technologies until that moment. They could not sequence perfectly repetitive regions like the last 8% were. [30].

To circumvent some limitations of the past generation, in 2005, the Second Generation of sequencers started to appear, like the time needed to prepare the solutions, duplicate the sequences and the high cost of these steps [33]. The principle of this new era was to generate more leads in less time and lower costs. In addition, the reads here are smaller than the reads from Sanger (up to 400 bases long), so many more sequences are generated to compensate for possible gaps during genome assembly at the end of sequencing.

One of the technologies used is Illumina. This sequencer flow is shown in Figure 5.5:

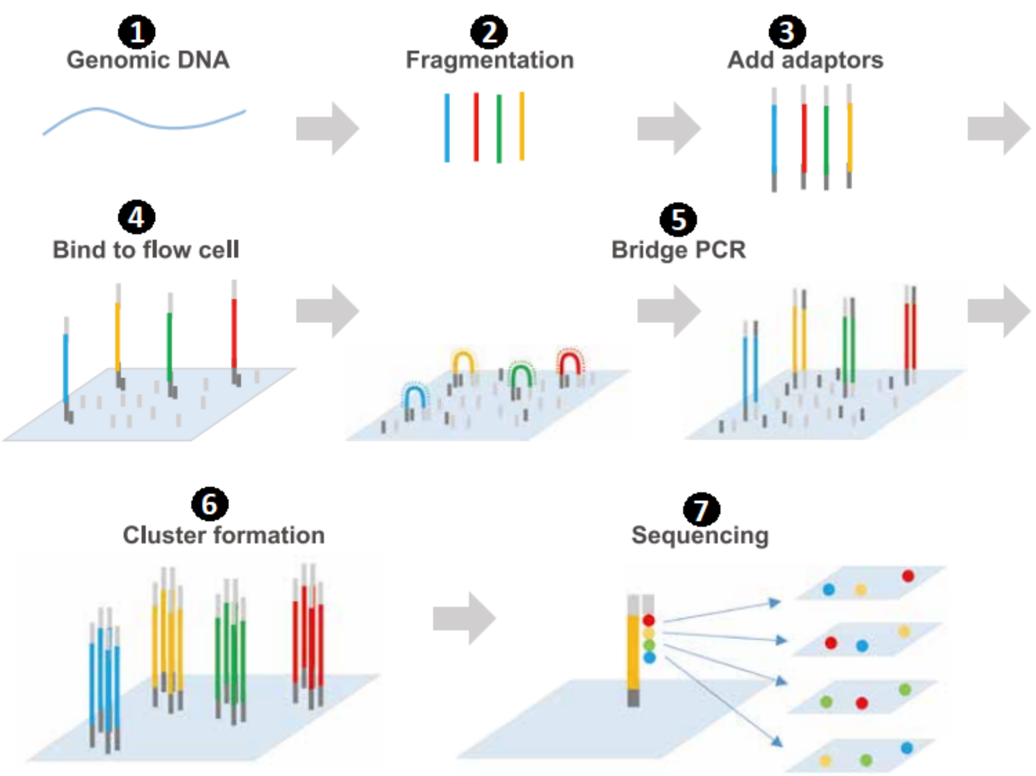


Figure 5.5. Illumina Sequencing. Adapted from [34]

1. Extract DNA strand to study.
2. Make DNA fragmentation from the last DNA strand.
3. Add adaptors to the ends of the fragments. Each adaptor is complementary to the sequence bases.

4. The adaptors will bind to the flow cell that contains others adaptors that are complementary to one of the fragments adaptors.

5. The fragment-free adaptor ends up binding to another adaptor of the cell, making a bridge between the two cell adaptors. Now, the *DNA polymerases* binds to one of the cell adaptors and starts building the complementary fragment. When the enzyme stops, the bridge is undone, and one of the adaptors is released, resulting in two sequences (the original and the copy). Creating a copy of the DNA strand is repeated thousands of times. The process is called Polymerase Chain Reaction (PCR).

6. When all fragments have already been copied, all reverse strands created are cleaved and eliminated from the flow cell. We will have a result cluster containing thousands of samples for each fragment from step 2.

7. *Primers* are attached to the free fragment end, and *DNA polymerases* starts building the complementary fragment. Each nucleotide has a different fluorescent element attached, so a light signal is emitted when the enzyme binds to the correct base. Belonging to a cluster, several signals will be released at the same time and captured by a sensor, which will identify which nucleotide was inserted. Like in Sanger, we will have the complementary sequence of the DNA fragments of step 2.

Unlike the Sanger method, which can sequence only one fragment at a time, Illumina can make the process in parallel, using the many clones of the fragments, drastically decreasing the execution time and increasing the number of reads generated [33]. As a result, nowadays is possible to use Second Generation technologies to sequence the human genome in up to 10 days and at a cost below 10.000 dollars.

Some limitations persisted, and for this reason, new improvements have been made. For instance, the reads generated until now were very small. For some genome assembly, this method can be a problem as thousands of reads need to be generated to deal with repetitive regions (best seen in Section 5.3), making the process expensive and slow. Therefore, the Third Generation came to sequence molecules giving results bigger reads. Another advantage that lower the costs is that it is not necessary to make DNA amplification and we can use the fragment alone, without clones [21].

One of the approaches and currently the most used method is the Single-Molecule real-time (SMRT), used by *Pacific Biosciences of California, Inc.* (*PacBio*) [21]. The idea of reading luminous signals is the same as the past generation. Before sequencing, the DNA sample needs to be prepared; this step takes some time. First, the double strand DNA is connected to adaptors creating a circular sample called *SMRTbell Library*. A *primer* and the *DNA polymerases* are added to the library as seen in Figure 5.6.

In Figure 5.7, we can see the PacBio flow described below.

a) Inside the PacBio machine are cells called SMRT Cell with millions of pores called Zero-mode Waveguides (ZMW). Each library created is mobilized in each ZMW. As in the Illumina method, we have labeled nucleotides with different fluorescent elements attached to them. Light is emitted as *DNA polymerases* attach these bases.

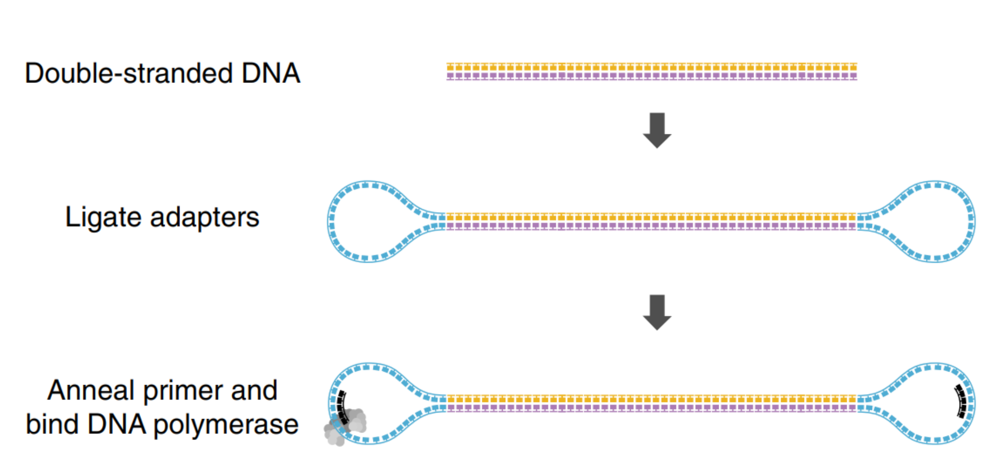


Figure 5.6. SMRTbell Library example. Adapted from [43]

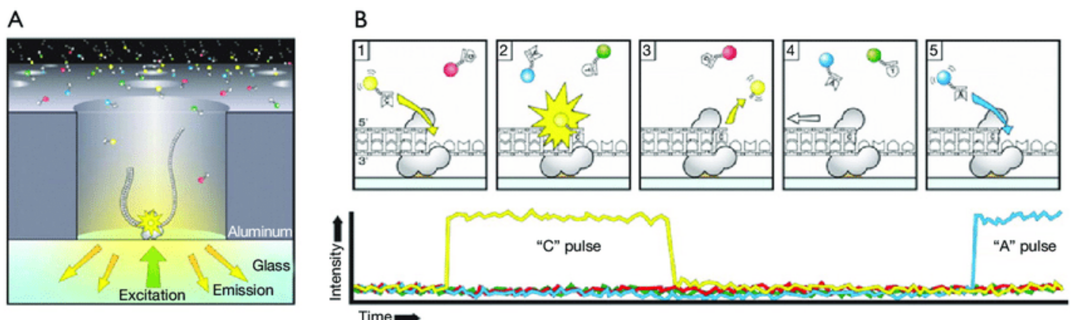


Figure 5.7. PacBio Sequencing [42].

b) With a sensor, nucleotide insertion is captured in real-time and depending on the intensity, we know the sequence of nucleotides.

The *DNA polymerases* works very quickly, and sometimes the signal is misread from the sensor or ends up not being read between one incorporation of bases and another. It is one reason for the generations of reads containing high error rates. Another problem that can occur is error/duplication in library ZMW binding. Each SMRT Cell has about 150.000 ZMW; from that, only 35.000 and 70.000 produce successful reads, with at most 60kb [44]. Without clones to make a consensus fragment, the error rate can reach 20% [17].

PacBio itself has a complementing of its method that reduces this error, using the maximum life of the *DNA polymerases* creating a clone of the sequence being sequenced, called HiFi sequencing. In Figure 5.8, The *DNA polymerases* is creating a Circular Consensus Sequence (CCS). An overlap of the CCS on itself to create a consensus sequence is done, decreasing the error rate. However, depending on the *DNA polymerases* lifetime, the CCS can be much smaller than the read without this technique, and the consensus

sequence can also decrease. With that, the experiment can become more expensive due to the increase in operations necessary to cover a region in the genome and the time spent in the CCS generation as well [44].

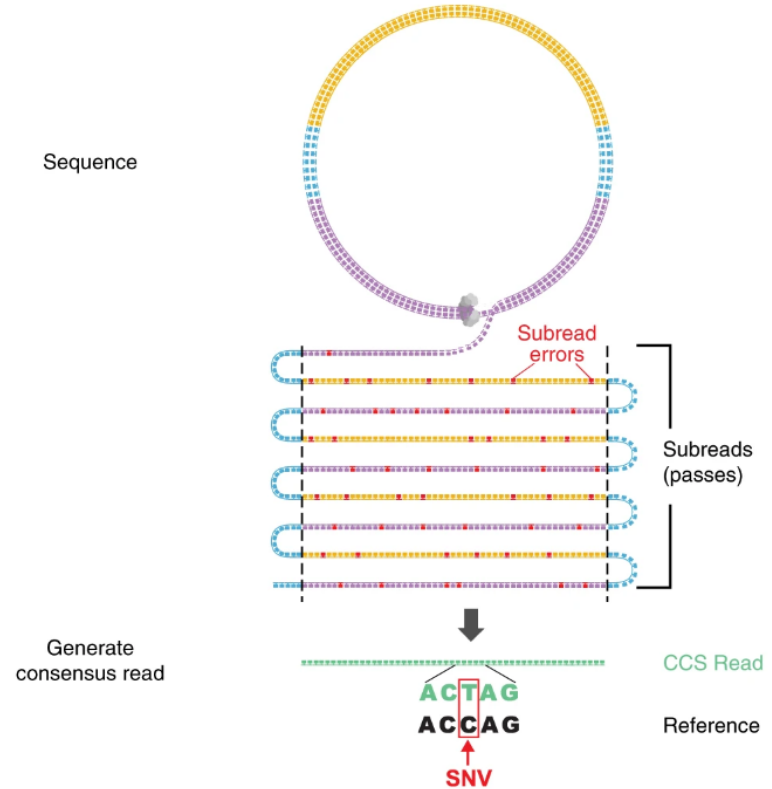


Figure 5.8. HIFI reads. Adapted from [43]

In Table 5.1, we can see a comparison between the technologies explained and all these details should be in mind before starting a study.

Table 5.1. Performance comparison of sequencing platforms of various generations. Adapted from [44].

	Sanger ABI 3730xl	Illumina HiSeq 2500 (High Output)	Illumina HiSeq 2500 (Rapid Run)	PacBio RS II: P6-C4
Read length (bp)	600–1000	2×125	2×250	1.0–1.5×10^4 $on average$
Error rate (%)	0.001	0.1	0.1	13
Reads per run	96	8×10^9 $(paired)$	1.2×10^9 $(paired)$	3.5–7.5×10^4
Time per run	0.5–3 h	7–60 h	1–6 days	0.5–4 h
Cost per million bases (USD)	500	0.03	0.04	0.40–0.80

Using long reads with a low error rate (from more modern sequencers or techniques that correct reads), it is possible to perform studies that were not possible in the

past, as some related to those of the human genome. For instance, in 2022, almost 20 years after the human genome sequencing, the unremitting regions that were not known were completely sequenced [30].

5.3. Error correction, Sequence assembly and Analysis

In this section, we discuss hybrid and self-error correction methods, providing examples, the basic algorithm for reading correction, and examples of sequence assembling and analysis for information discovery.

5.3.1. Error correction method

Depending on the type of experiment being performed and/or the organism being studied, some methods of sequencing from past generations may be better alternatives than current methods, having a reasonable cost benefit. The error rate and the resulting reads are low, and when used in smaller genomes, assembling the sequences is easier than when performed in larger genomes [13]. Getting the human genome, for example, when we try to assemble its regions by overlapping the short reads, we cannot make a consensus sequence due to many regions that are repetitive and complex [14].

As we can see in Figure 5.9, when using a small number of reads in a region much more extensive than them, may occur gaps in the consensus sequence. To work around this, we can generate a massive amount of short reads to increase the overlapping area, until we can assemble a complete consensus sequence. However, with that, we have an expensive experiment.

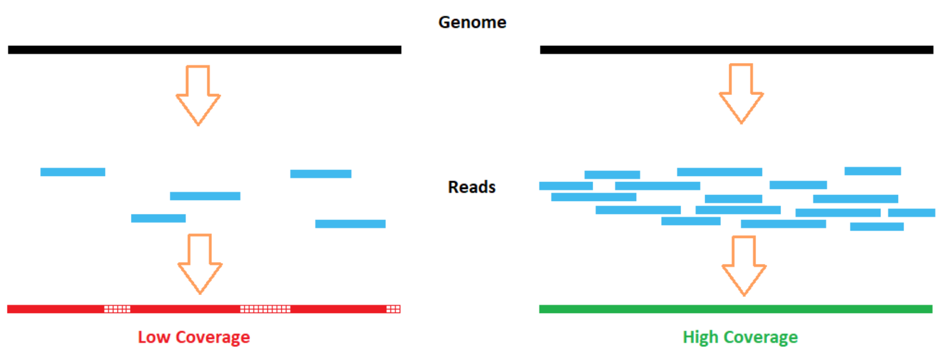


Figure 5.9. Example of a low and a high coverage consensus sequence

Thinking of another alternative, we can use the Third Generation Sequencing, like PacBio, that produces long reads than the previous generations [14]. With this, instead of using 100 small reads to make a consensus for a region, we can sequence this region as a long read with a lower cost but with the disadvantage of having a higher error rate. This is a problem because if we need to use these long reads with others to assemble another more extensive region, the present errors can make the process more complicated and increase the computational cost, in addition, to be imprecise in some repetitive areas [17].

To facilitate studies using larger genomes and allow research centers with low

investment to perform some experiments, we have methods to perform error corrections for long reads, decreasing the error rate. This is the first thing to think about when using long-read sequencing. New versions of PacBio have in it sequencing finalization steps to prevent high error rates (HiFi). However, this method ends up making the experiment more expensive, for example, making it impracticable. For this reason, other software works get the reads with a high error rate and process it, trying to correct the so-called wrong bases, using two main approaches [15].

5.3.1.1. Hybrid correction

This type of correction uses short reads from a known genome to work as a reference. Many techniques can be performed, like comparing the k-mers from a reference genome against the long reads. If we have the genome of the nematode *Caenorhabditis elegans* (*C. elegans*) organism and want to study it, for example, we can sequence it with PacBio. However, as we know, the resulting long reads may have an error rate up to 30%. The *C. elegans* was the first multicellular organism to have its approximately 100 Mega bases completely sequenced. Using the studied and labeled nematode genome as a reference to the experiment is a good approach because we can consider it without errors [16]. Nevertheless, suppose we do not have an organism completely sequenced. In that case, we can use short reads (like the ones from Illumina) as our reference genome because the error rate is minimal, so it is very close to the real genome [17].

For this, we have to create the k-mers from the reference genome. K-mers are substrings with length K inside a string, being this string a nucleotide sequence. For example, in Figure 5.10, we show k-mers of size 16 in a sequence. If the sequence has size L, we will have L - K + 1 k-mers in total.

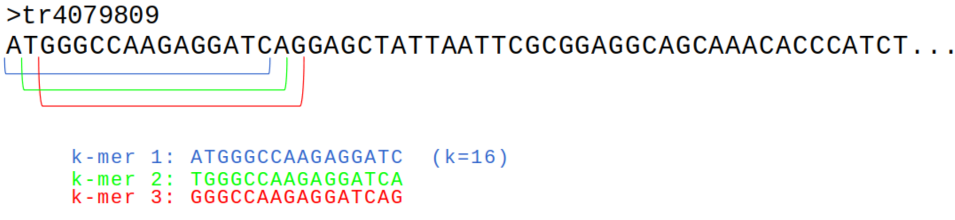


Figure 5.10. Example of 16-mer in a sequence

If we want to know which organism a studied sequence belongs to, we just break it down in k-mers and compare their frequencies with the reference genome. The higher the frequency, the closer these organisms are.

K-mers with errors do not often appear, so their frequency will be minimal. Most of the time, they will not be present in the reference genome (in some cases, if we use other K values, some k-mers with errors can appear in different frequencies and match with any k-mer of the reference).

First, we have to determine the size of the k-mer (in this example, we will use 24)

and create a list with all k-mers of the reference genome. Now, let us suppose we have the following read with some error in it.

AAAAACCGAAAAAAGTGTGGACTTCCCGCGTGAAAAC(...)

In this example, K is 24 and we want to correct this read.

AAAAACCGAAAAAAGTGTGGACTT CCCGCGTGAAAAC

Running the Algorithm 1, a new k-mer is formed by sliding one base from the previous state. The result will be:

A AAAACCGAAAAAAGTGTGGACTTC CCGCGTGAAAAC

Now, we check if this new k-mer is present in the genome reference k-mer list. The algorithm continues sliding the k-mer window until it finds a k-mer that is not in the reference list.

AA AAACCGAAAAAAGTGTGGACTTCC CGCGTGAAAAC

AAA AACCGAAAAAAGTGTGGACTTCCC GCGTGAAAAC

AAAA ACCGAAAAAAGTGTGGACTTCCCG CGTGAAAAC **Not in k-mer list!**

When the problematic k-mer is found, test all possible errors that may have occurred. For each check, verify if the new k-mer is valid.

- **Insertion**: Remove G
- **Deletion**: Substitution of G by AG, CG, GG, TG
- **Mismatch**: Substitution of G by A, C, T

If only one valid k-mer is found, this is the corrected sequence. We go to more complex correction steps if 0 or more than one is found.

If the previous step fails, we can align the sequence with the reference genome. For this, the last valid k-mer found (the one before the problematic k-mer) has a complementary of it in the real genome. For example, if the reference genome is `AAACCCGGGTTT` and K is from size 6, the complementary of the 6-mer green is the 6-mer in cyan, as shown below.

Algorithm 1 Simple read correction

```
0: sum ← 0
0: C ← NULL
  for i ← k+1 to L do
    kmer ←read[i :i+k]
    if not (kmer in reference genome) then
      newKmer ←checkInsertion(kmer, i)
      if newKmer in reference genome then
        sum ← sum+1
        if sum == 1 then
          C ← newKmer
        end if
      end if
      for base in [‘A’, ‘T’, ‘C’, ‘G’] do
        newKmer ←checkDeletion(kmer, i, base)
        if newKmer in reference genome then
          sum ← sum+1
          if sum == 1 then
            C ← newKmer
          end if
        end if
      end for
      for base in [‘A’, ‘T’, ‘C’, ‘G’] do
        newKmer ←checkMismatch(kmer, i, base)
        if newKmer in reference genome then
          sum ← sum+1
          if sum == 1 then
            C ← newKmer
          end if
        end if
      end for
      if sum != 1 then
        C ← NULL
      end if
      return C
    end if
  end for
  return C =0
```

AAACCC GGGTTT

This works like a prefix and a suffix. For example, the green k-mer is the prefix of cyan k-mer. The reference suffix is aligned with the k-mer being analyzed plus its suffix. If the referenced prefix has more than one suffix, the operation occurs with all of it, and the corrected sequenced is the one with optimal alignment.

5.3.1.2. Self correction method

This method is more computationally expensive. A reference genome is not required and therefore is necessary to perform an alignment between all the reads with each other [47].

In Figure 5.11, we can see the workflow of CONSENT, a combination of self-correction methods from the state-of-the-art [47]. First, an overlap of some read's region is computed by an external tool. Then, CONSENT select a template read to correct. Every read region is not part of the template window. CONSENT is removed from the next steps because this region will not be used for correction. Next, windows are defined, and each of them is aligned. When the consensus of the windows is generated, this sequence is polished (corrected) by a local *de Bruijn graph*. In the end, the consensus sequence is aligned with the original template read to perform the correction of it.

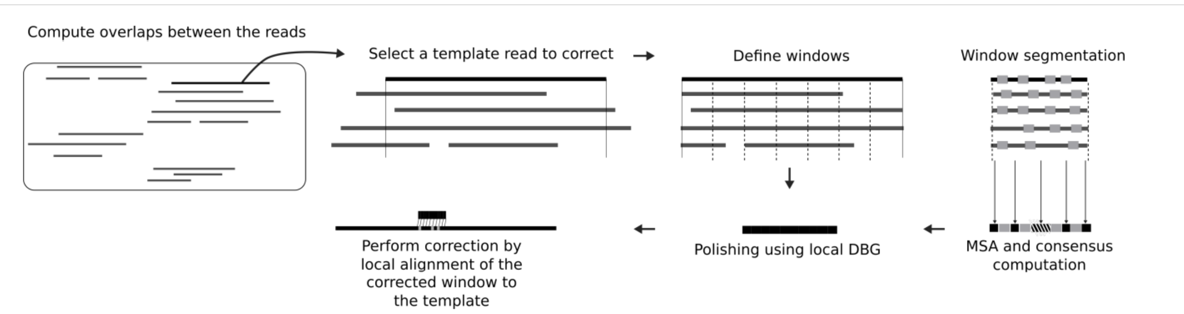


Figure 5.11. Overview of CONSENT workflow. [47]

5.3.2. Sequence assembly

When we have all the reads from the sequencing method, it is time to start reconstructing the original sequence. The sequencers only produce reads from random sizes (within a specific range depending on the technology used). For this reason, we need to use other strategies to overlap these reads and assemble them, as shown in Figure 5.12, in another continuous sequence called *contig*. This *contigs* are ordered, oriented and grouped together to form *scaffolds* [18].

Depending on the type of sequencer used, the number of reads generated per run can be considerable, as seen in Table 5.1. If the study focuses on a more complex organism and needs more extensive genome coverage, this implies a larger data storage. One example is a genome size of approximately 100MB with 40x of coverage, which gives us

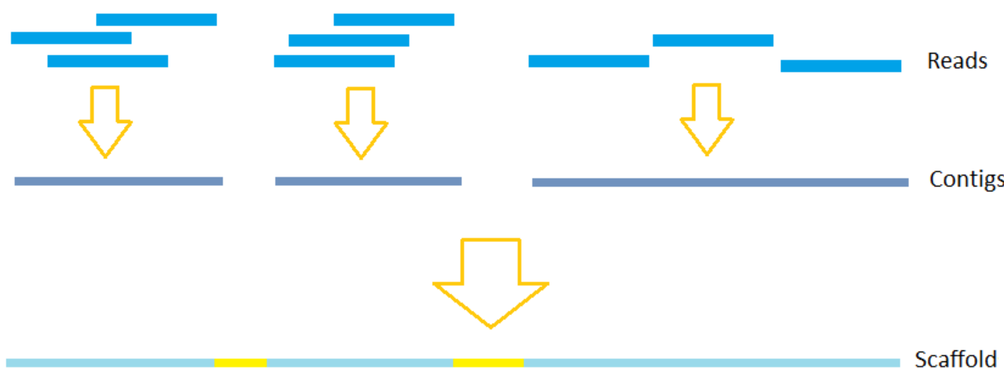


Figure 5.12. Example of assembly

a raw output of 4.5GB [45]. The larger the genome and/or the coverage, the greater the output. Genomes from approximately 3GB and 100x of coverage can reach terabytes of data [18].

All reads from a region need a good overlap with those of the same region to have the best assembly. If this doesn't happen, some *contigs* may not accurately build *scaffolds* and the result will be very fragmented [18]. Some software can help to find low-quality reads and regions, for example, from a database with reference genomes that have already been studied and sequenced by another sequencer [18].

So when we want to reconstruct the genome from an organism, short reads can lead to gaps in assembled sequences (using more short reads can work around this problem). While long reads with a low error rate can map certain regions by themselves, depending on the number of reads generated, making assembly with fewer gaps.

If the genome to be generated is new and/or does not have another sequence as a reference as a template, the approach used is the *De-novo* assembly. One of the algorithms uses the method of overlap. In short, each read is compared to the others to find the closest match and is merged to form a *contig*. The process is repeated until necessary, creating new *contig* and generating *scaffolds* at the end. This process is computationally expensive, since each assembly tool has a specific depending on the study to be carried out. Making it difficult to predict which one is the best to use [18]. If we have a template sequence, we can align the reads against it, and we will have a faster process with less memory consumption. The two types can be seen in Figure 5.13.

5.3.3. Analysis

A new sequence does not necessarily represent a discovery. It is necessary to analyze this sequence. Analyze the new sequence that leads to inferences and discoveries about it.

Sequence analysis started in the 1980s to compare two DNA sequences and find if they are homologous to each other (i.e., if the sequences share common areas) [48].

In the beginning, only a superficial analysis could be performed. After the 1990s, more complex analyses were performed with increased processor speed and storage ca-

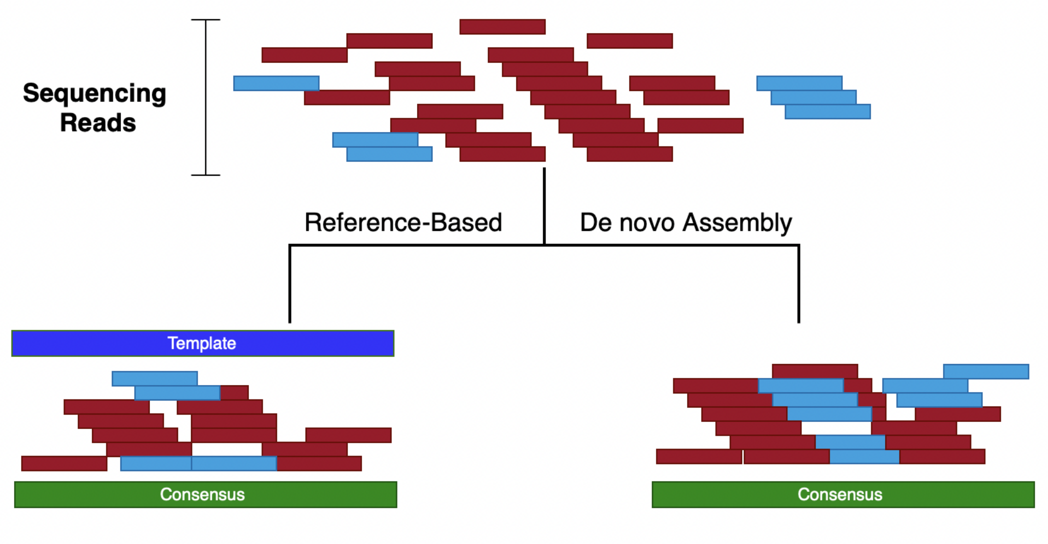


Figure 5.13. Types of sequencing assembly. [46]

pacity. For instance, analyze the family's histories [48]. Also, the software becomes more available for many researchers, enabling studies from different disciplines with a vast number of individual sequences being compared. Quantifying a specific characteristic became possible due to the advancement of technology.

There are vast applications for sequence analyses, such as finding similar sequences in a database, finding variations in the DNA such as point mutations and single nucleotide polymorphism (SNP), map evolution, and genetic diversity [48]. Nowadays, even with servers and supercomputing centers making it possible to do some analyses that seemed impossible in the past, the analysis's complexity and urgency seem to grow more each year. For example, the ancestry exams and the ones that can predict the risk for certain types of diseases are more frequent, increasing the demand for computing resources.

The demand to increase the efficiency of this analysis is evident. The most commonly used algorithm is sequence alignment, in all its variations.

The following subsection describes the sequence alignment, which can be used in many bioinformatics procedures. Also, we detail the different types and discuss the possible improvements in Smith-Waterman and CLUSTAL W algorithms.

5.4. Sequence alignment

After a new gene is found by scientists investigating a disease, the next step consists of finding the gene function. The typical approach used to infer a function is comparing the new one find with all already known gene sequences [8].

This approach has many successful cases. We will describe the first two reported in the literature. In 1984, scientists discovered a new *cancer-causing v-sis oncogene*[1]. Af-

[1]Oncogenes are genes capable of causing cancer, they can be mutated genes or an over-expressed gene. [8].

ter the discovery, they performed a computation technique to compare the new discovery with all the know genes until that moment.

At the first moment, we imagine that the result matches this new gene and another cancer-related gene. However, this is not what the scientists found.

Surprisingly, the gene show similarities with Platelet-Derived Growth Factor (PDGF), a gene involved in growth and development. This discovery was essential for scientists to understand cancer as an over-expressed gene [8].

Another example date back to 1989. The result of an alignment made it possible to associate the cystic fibrosis genes with an Adenosine Triphosphate (ATP) binding protein gene.

Known as the salty kiss disease, this association seen in the genes could explain the salty skin presented by people with cystic fibrosis. In contrast, the ATP binding protein gene is responsible for expanding the membrane cell multiple times to transport the sodium ions [8].

A current example of sequence alignment use is the ancestry exams. Most companies that provide this type of exam create a database with the DNAs sequences of their clients. Then, when a new client submits the collected DNA they compare, using sequence alignment, the new sequence against all the sequences in the built database. Therefore, proving information about the ancestry of the DNA sequence. Using this approach is also possible to find relatives by comparing the DNA sequences [50].

It is possible to notice that sequence alignment was essential for scientific discoveries, including understanding many diseases, helping to create a treatment, or even getting close to the cure. Therefore, sequence alignment is an indispensable procedure in the bioinformatics area.

Sequence alignment compares two or more sequences, which can be either an amino-acid (i.e., protein) or a nitrogenous base (i.e., DNA or RNA) sequence. When we compare only two sequences, we procedure a pair-wise alignment. In contrast, when there are more than two sequences is called a multiple sequence alignment [7].

Sequence alignments can also be separated into two categories, global and local alignment. On the one hand, the global alignment tries to align the entire sequence, considering all the amino-acids or nitrogenous bases until the end of the sequence analyzed. On the other hand, a local alignment aims to find the best subalignments or islands of matches from an aligned sequence. In this case, only a fraction of the sequence is aligned. The result, in this case, is the fraction that represents the highest number of matches. The following subsection discusses in detail each, including examples and applications [7].

In a simplified way, two sequences are aligned by writing one below another in two separate rows. The alignment is procedure character by character. We have a match if the two characters in the same column are identical. We have identical sequences if all the characters from both sequences are equal.

Otherwise, if the character is nonidentical, they can be characterized as a miss match, or a gap can be inserted. Gaps are used to optimize alignments, performing a sequence shift. An optimal alignment can include the addition of gaps to reach as many

matches as possible [7].

5.4.1. Global alignment

A global alignment can be executed in a protein, DNA, or RNA sequence. The primary process is the same for both types of sequences. In a protein, amino acids are aligned and in DNA and RNA sequences, nitrogenous bases are aligned [7].

The global alignment is stretched over the length of the entire sequence. The aim is to reach the highest number of matches in the sequences, namely the alignment score. Gaps can be added in any of the sequences, when a gap is added, the score suffers a penalty. The global alignment is performed up to and including the end of the sequences [7].

We analyze an alignment from column to column, comparing the characters presented in each sequence. If they are identical, we have a match. On the other hand, if the characters presented in a column are nonidentical, we have a miss match. Also, in this case, a gap can be added to improve the number of matches, is essential to notice that gaps added need to respect the length of the sequences, not exceeding it[7].

Figure 5.14 illustrates an example of global alignment in a hypothetical protein sequence. We can see two sequences and the matches and miss matches from each column. A match is observed in the first column, amino-acid L, which stands for Alanine. The second column is a miss match. Next, a gap is added in the third column of the second sequence, and we also see a gap in the 13th column of the first sequence.

Analyzing the entire sequence we observe seven matches (Alanine(**L**) in 1th column, Lysine(**K**) in 6th and 10th column, Glycine(**G**) in 9th and 11th column, Arginine(**R**) in 15th column and Aspartic acid(**D**) in 18th column) and eleven miss matches. Furthermore, two gaps were added.

Figure 5.14. Example of global alignment. Adapted from [7].

In summary, when sequences are similar and have the same length a global alignment is suitable to be performed in these sequences [7]. In most cases, global alignment is performed when sequences are related along their entire length. A local alignment can be used when only a region of similarity is sought.

5.4.2. Local alignment

Similar to a global alignment, sequences from proteins, DNAs, and RNAs can also be used in a local one. Although different from the global approach, in a local alignment, only a fraction of the sequence is aligned, searching in the sequence for the fraction representing the highest number of matches.

Figure 5.15 shows the same sequence illustrated in Figure 5.14, although, as we

can see, the optimum local alignment considers only the fraction of G-K-G (Glycine-Lysine-Glycine) (i.e., the three consecutive matches).

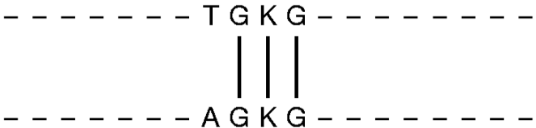


Figure 5.15. Example of local alignment. Adapted from [7].

This type of alignment is used to align sequences with conserved regions or domains, the ones with similarity in only a part of the sequence. Additionally, since only a fraction of the sequences are aligned, they can differ in length.

5.4.3. Pair-wise and Multiple sequence alignment

Pair-wise consists of aligning only two sequences. The sequences can be proteins, DNA or RNA. An extension of the pair-wise alignment is the multiple sequence alignment when the alignment is performed in three or more sequences.

For instance, a new sequence can be aligned against a sequence database to recognize the family or species, for example, [7]. Another example is finding conserved residues and identifying structural and functional domains when aligning more than two protein sequences.

5.4.4. Alignment scoring functions

For all sequence alignment types (i.e., local or global), we need to define the values associated with each scenario in the alignment, to calculate the alignment score.

When two sequences are being aligned, four scenarios are possible: (1) a gap in the first sequence, (2) a gap in the second sequence, (3) a match in both sequences, and (4) a miss-match between sequences.

Keeping these four scenarios in mind, a scoring function must attribute a value to each possible situation.

We will consider a simple scoring function in this chapter. -1 is assigned if a gap is added or a miss-matches is observed. For a match, +1 is attributed. This simple scoring function is describe in Equation 1 [1].

$$\begin{aligned} \sigma(-,a) = \sigma(a,-) = \sigma(a,b) = -1 \ \forall(a \neq b) \\ \sigma(a,b) = 1 \ \forall(a = b) \end{aligned} \tag{1}$$

In a global alignment, this function is used to score the sequence alignment since this alignment considers the entire sequence. Considering the example illustrated in Figure 5.16 and Equation 1, we have seven matches, three gaps, and two miss-matches. In this case, the result of the scoring function is equal to two [1].

$$A(\mathbf{s},\mathbf{t}) = \begin{array}{cccccccccccc} \texttt{V} & \texttt{I} & \texttt{V} & \texttt{A} & \texttt{L} & \texttt{A} & \texttt{S} & \texttt{V} & \texttt{E} & \texttt{G} & \texttt{A} & \texttt{S} \\ | & | & | & | & & | & & | & & & & | \\ \texttt{V} & \texttt{I} & \texttt{V} & \texttt{A} & \texttt{D} & \texttt{A} & \texttt{-} & \texttt{V} & \texttt{-} & \texttt{-} & \texttt{I} & \texttt{S} \end{array}$$

Figure 5.16. Example of the score of a global alignment. Adapted from [1].

The scoring function for the example in Figure 5.16, is describe in Equation 2.

$$M(a) = 7 - 2 - 3 = 2. \quad (2)$$

The alignment score is directly associated with the function score. Therefore, the choice of the scoring function should try to reflect the biological behavior[1].

5.4.5. Substitution matrices

Substitution matrices can be used to consider multiple scoring functions and attribute different scoring for the substitution of one letter for another.

These matrices can be used for proteins or DNA and RNA sequences. The matrix shows the substitution cost between amino acids in a protein scenario. While in a DNA or RNA scenario, the cost reflects the cost of nucleotide substitution. Also, the match and gap values are shown in the matrices.

Using a substitution matrix is possible to expect ambiguous characters or even mutations and changes arising from the evolution process and assign a higher value for these changes [7].

Figure 5.17 illustrates a 5x5 matrix that can be used as a nucleotide substitution matrix. All pairs of $\sigma(a,b)$ nucleotide or gaps are shown. Note that $\sigma(-,-)$ has a not determined (**N/D**) value since an alignment between two gaps is not needed.

	A	C	G	T	−
A	+1	−1	−1	−1	−1
C	−1	+1	−1	−1	−1
G	−1	−1	+1	−1	−1
T	−1	−1	−1	+1	−1
-	−1	−1	−1	−1	N/D

Figure 5.17. Example of a substitution matrix. Source: [1].

5.4.6. Smith–Waterman

The Smith-Waterman [19] is an algorithm that uses dynamic programming to find, in any sequences of any lengths, where an optimum alignment can be found. The sequences can be either RNA, DNA or protein sequences.

5.4.6.1. The algorithm

The main steps of Smith-Waterman consist of:

- Initialization of a scoring matrix
- Filling the matrix with the calculated scores
- Trace back the sequences to find the best alignment

Considering two sequences, the first sequence's characters are placed at the top of the matrix, and the character from the second sequence are placed vertically on the left side. Next, the alignment score for each character pair (i.e., $\sigma(a,b)$) is placed in the corresponding matrix position. The code is described in Algorithm 2.

Algorithm 2 Smith–Waterman algorithm

0: $S[0,0] \leftarrow 0$
  for $i \leftarrow 1$ **to** M **do**
    $S[i,0] \leftarrow 0$
  end for
  for $j \leftarrow 1$ **to** N **do**
    $S[0,j] \leftarrow 0$
    for $i \leftarrow 1$ **to** M **do**

$$S[i,j] \leftarrow \text{MAX} \begin{cases} 0 \\ S[i-1,j-1]+\sigma(x_i,y_j) \\ S[i,j-1]+\sigma(-,y_j) \\ S[i-1,j]+\sigma(x_{i,)} \end{cases}$$

    end for
  end for
  return $S[M,N]$ =0

To illustrate the functionality, we will describe the alignment of the sequences *PARALLELDNAALIGNMENT* and *DNASEQUENCE*. Analyzing both sequences, we can see that the subsequence DNA is present in both. Thus, the solution for this alignment is **DNA** as described in Figure 5.18.

Figure 5.18. DNA subsequence alignment.

Since the solution is already known, we will describe step by step the Smith-Waterman execution for the two sequences as inputs. The first step is configuring the score matrix. To create it, sequences are placed on top of the matrix and vertically on the left side, as illustrated in Figure 5.19.

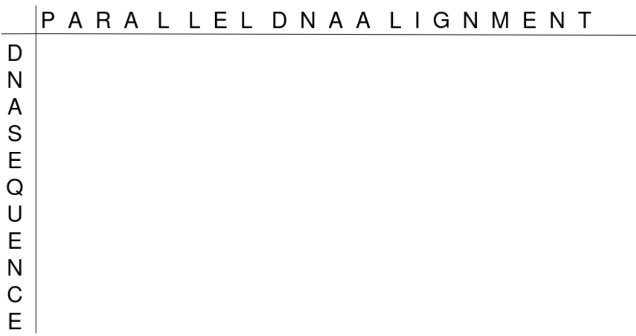


Figure 5.19. Create matrix based in the sequences.

	P	A	R	A	L	L	E	L	D	N	A	A	L	I	G	N	M	E	N	T
D	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
N	0																			
A	0																			
S	0																			
E	0																			
Q	0																			
U	0																			
E	0																			
N	0																			
C	0																			
E	0																			

Figure 5.20. Initialize matrix with zeros.

After is the initialization, which consists of adding zeros to the first column and first line of the matrix, the result of this step is illustrated in Figure 5.20.

Next, we will use the scoring function described in Equation 1 to score all the pairs in the sequences. The result of this step is shown in Figure 5.21.

	P	A	R	A	L	L	E	L	D	N	A	A	L	I	G	N	M	E	N	T
D	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
N	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0	1	0	0	1	0
A	0	1	0	1	0	0	0	0	0	0	3	1	0	0	0	0	0	0	0	0
S	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
E	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0
Q	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
U	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
E	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0
N	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0
C	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
E	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0

Figure 5.21. Score function calculation.

The last step consists of a trace-back procedure, starting with the highest element observed in the table of Figure 5.21 until a zero element in the trace-back path [1]. The

highest element in this example is three. Performing a trace-back, we find element two (i.e., a match between character N and character N). The next step is element one, the result of the match between D and D, and the next element is a zero, which stops the process. The result of the trace-back is shown in Figure 5.22.

	P	A	R	A	L	L	E	L	D	N	A	A	L	I	G	N	M	E	N	T
D	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0	0	0	0
N	0	0	0	0	0	0	0	0	0	**2**	0	0	0	0	0	1	0	0	1	0
A	0	1	0	1	0	0	0	0	0	0	**3**	1	0	0	0	0	0	0	0	0
S	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
E	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0
Q	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
U	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
E	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0
N	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0
C	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
E	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0

Figure 5.22. Trace-back procedure.

After all the steps, the alignment described in Figure 5.18 is reached. This way, finding the optimum alignment for the sequences.

5.4.6.2. Parallel solutions

It is possible to notice that Smith-Waterman is a time-consuming algorithm associated with all the complex procedures and comparisons. When we consider two sequences of lengths m and n. The computational and space complexity equals $O(mn)$ and $O(m)$, respectively. When we consider multiple alignments, the execution time gets even more significant, where $O(mn)$ is multiplied by the number of sequences being compared.

Many parallel solutions were developed to accelerate the algorithm, using technologies such as vector-level parallelism, thread-level parallelism, process-level parallelism, and heterogeneous approaches [20].

The first step when paralleling an algorithm is identifying and solving the dependencies. The Smith-Waterman algorithm has a dependency on the previously calculated scores. As described in Equation 3.

$$MAX \begin{cases} 0 \\ S[i-1, j-1] + \delta(x_i, y_j) \\ S[i, j-1] + \delta(-, y_j) \\ S[i-1, j] + \delta(x_{i,)} \end{cases} \quad (3)$$

The current position depend on the S[i-1, j-1], S[i-1, j] and S[i, j-1] positions, this dependency is illustrated in Figure 5.23. The first step in implementing a parallel solution is to solve this dependency so that the position calculation for the score matrix can be separated into groups.

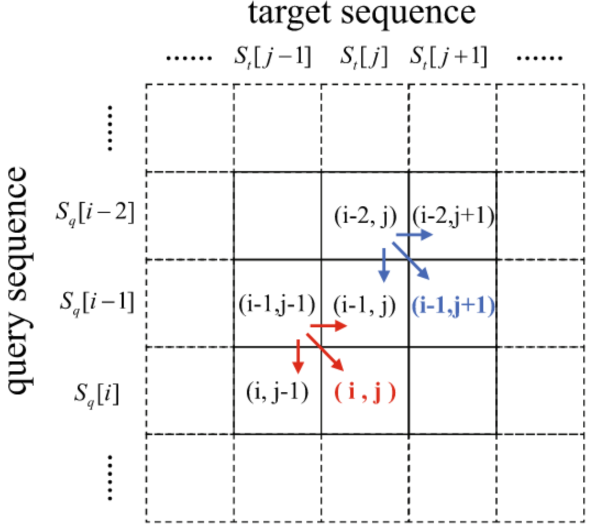


Figure 5.23. Existing dependencies while calculating the score matrix [20].

One possible technology for active parallelism is vector-level parallelism, also known as Single Instruction Multiple Data(SIMD), this type of parallelization performs the same operation in a group of data, namely spatial parallelism. For example, using a controller to control multiple processors, executing one instruction in multiple data [20]. Figure 5.24 shows the difference between vector and scalar operations.

Scalar	Vector							
A	A7	A6	A5	A4	A3	A2	A1	A0
+	+	+	+	+	+	+	+	+
B	B7	B6	B5	B4	B3	B2	B1	B0
A+B	A7+B7	A6+B6	A5+B5	A4+B4	A3+B3	A2+B2	A1+B1	A0+B0

Figure 5.24. Vector and scalar operations [20].

When we consider solutions in vector-level parallelization, some possible strategies are reported in the literature to solve the dependencies.

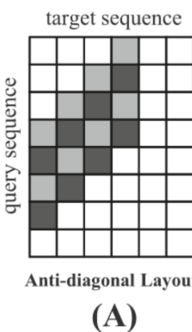
In inter-sequence, the parallelization is performed in a pair of sequences. The available solutions are organized in anti-diagonal, sequential, or striped, described in Figure 5.25.

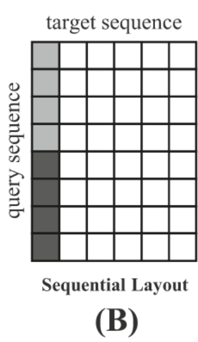
Anti-diagonal layout [26] works in the observation that calculation cell S[i,j] and S[i-1, j+1] is an independent task, thus possible to be calculated in parallel. Despite that, this approach does not change computation and space costs.

Another layout is the sequential [27]. Algorithm 3 describes the pseudo-code for the Lazy-F evaluation, a solution using the sequential layout to solve the data dependencies using auxiliary vectors.

Algorithm 3 Sequential-layout for Smith-Waterman algorithm

0: VECTOR4 vGo = $[\Delta, \Delta, \Delta, \Delta]$
0: VECTOR4 vGe = $[\delta, \delta, \delta, \delta]$
for $j \leftarrow 0$ **to** n/4 **do**
    $Hb \leftarrow [0,0,0,0]$
    $Eb \leftarrow [0,0,0,0]$
end for
for $i \leftarrow 0$ **to** m **do**
    $vHx \leftarrow [0,0,0,0]$
    $vF \leftarrow [0,0,0,0]$
    for $j \leftarrow 0$ **to** n/4 **do**
        $vH \leftarrow Hb$
        $vE \leftarrow Eb$
        $vTem1 \leftarrow vH >> 3$
        $vH \leftarrow (vH << 1)|vH$
        $vH \leftarrow vTem1$
        $vH \leftarrow vH + M[i][j]$
        $vH \leftarrow max(vH, vE)$
        $vF \leftarrow (vH << 1)|(vF >> 3)$
        $vF \leftarrow vF - vGo - vGe$
        if any values in vF > 0 **then**
            $vTem2 \leftarrow vF$
            while any values in vTem2 >0 **do**
                $vTem2 \leftarrow (vTem2 << 2) - vGe$
                $vF \leftarrow max(vF, vTem2$
            end while$vH \leftarrow max(vH, vF)$
            $vF \leftarrow max(vH, vF + vGo)$
        else
            $vF \leftarrow vH$
        end if$Hb \leftarrow vH$
        $Eb \leftarrow max(vH - vGo, vE) - vGe$
        $vMAX = max(vMAX, vH)$
    end for
end for=0





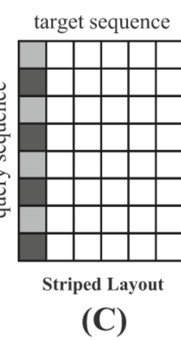


Figure 5.25. Inter-sequence parallel layouts [20].

In the sequential-layout, one character of the target sequence is aligned to the entire query sequence, as illustrated in Figure 5.25. The parallel solution came from dividing this process into equal parts and executing each part in parallel. For instance, the process is separated into four parts in the Algorithm 3. This number could be adapted for the SIMD technology used[26].

This solution still uses a computation of $O(mn)$ and space of order $O(m)$, nevertheless is faster than the previous solutions.

Finally, the striped layout [31] is a refined version of the sequential layout. The query elements are reorganized in this approach and, similar to sequential layout computation, remain O(mn).

Another way to achieve parallelism is using shared memory architectures to perform thread-level parallelism. In Figure 5.26 (A), we illustrate a shared memory architecture, in which the cores share access to the memory through an interconnection, differently from a distributed architecture, illustrated in Figure 5.26 (B), where each core has its own memory.

Some tools that provide parallelization using threads are POSIX Threads (Pthreads) [23] and OpenMP [24].

When we have the Figure 5.26 (b) architecture, we can reach parallelization using a process level. In this case, each process has an independent code segment and memory, which means the need to exchange information from one process to another. This exchange is done by communication. It is essential to notice that passing messages from one process to another represents an extra overhead to this technique. A commonly used protocol is Message-Passing Interface (MPI), a programming interface available for many programming languages.

Therefore, solutions that use shared memory in thread-level parallelism are available in the literature. For instance, KSW and KSW2 [51, 52], libssa[53], SeqAn[54], SWIPE[55], etc. One solution applied in thread parallel versions of Smith-Waterman is similar to the abovementioned solutions. Dividing the sequence to be aligned in subsets, normally equal to the number of threads available, each thread is responsible for its sequence [20].

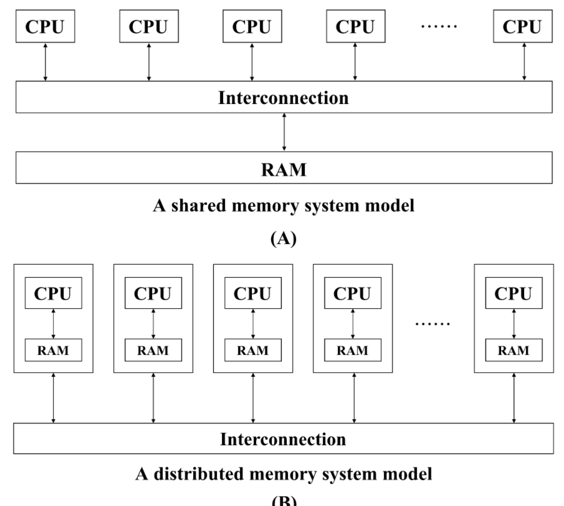


Figure 5.26. (A) Shared memory architecture; (B) Distributed memory architecture [20].

Nowadays, graphic processing units (GPUs) are becoming more common in personal computers. For that reason, heterogeneous parallel solutions, including the Central Processing Unit (CPU) and different accelerators such as GPUs are used [20].

Furthermore, in Python, a package named Biopython, have the alignment algorithm already implemented. Therefore, the algorithm automatically chooses the appropriate alignment algorithm between Needleman-Wunsch, Smith-Waterman, Gotoh, and Waterman-Smith-Beyer global. The decision is made based on the values of the gap scores observed in a pair-wise alignment.

Another widely used alignment algorithm is the one in which Smith-Waterman is based: Needleman-Wunsch. While Smith-Waterman procedure a local alignment, Needleman-Wunsch performs a global alignment.

5.4.7. CLUSTAL W

CLUSTAL W [32] is the most cited algorithm in bioinformatics. It is a global multiple alignment algorithm applied for three or more sequences, with a computational complexity of $O(n^2)$. For this reason, there is a need to improve CLUSTAL W performance since it is high complexity and vastly used algorithm. The algorithm uses dynamic programming to align the sequences, both amino acid and nucleotide sequences are accepted. The algorithm is divided into three main steps:

a) Calculate the distance matrix for each pair of sequences.

b) Determinate the topology of the progressive alignment.

c) Obtain the multiple alignments progressively.

In the literature, we have solutions in both shared and distributed memories. For instance, in shared memories, a solution used in most of the CLUSTAL W servers is proposed by Mikhailov et. al. [35], which uses OpenMP.

When analyzing distributed memory, a solution proposed by Liu [36] uses MPI to improve the average performance. All three algorithm steps were parallelized using the fixed-size chunking strategies, combining fine and coarse-grained division of work. They are reaching a speedup of 4.3% when 16 cores are used.

Another widely used algorithm is T-Coffee. This algorithm is also a multiple sequence aligner; although different from CLUSTAL W, it can be used for global or local alignment.

5.5. Open challenges

The importance of bioinformatics in our life is undeniable, and improvement and development still need to be done in many problems in this area. This section describes the complexity associated with the most used algorithms and the remaining challenges in bioinformatics studies.

One topic of study is redundancy removals in raw reads resulting from the DNA sequencing process. The first example is the MapReduce Duplicate Removal tool (MarDRe) [38], a tool to remove duplicate and near-duplicate DNA reads through the clustering of single-end and paired-end sequences from the FASTQ/FASTA dataset. Madre shows a computational complexity of O(logn). Similar to FastUniq [39], a tool with a functionality similar to MarDRe and an O(log N) complexity. Since the data volume is increasing, faster solutions are needed to process increasingly complex data. Improving these algorithms' performance is an open area.

Another new approach used in bioinformatics is machine learning. This technique could be used in many solutions, from predicting a gene function to analyzing MRI images. For instance, machine learning could be a tool to facilitate research or doctor analyses and anticipate some conditions or even a disease that still does not show symptoms.

In addition, considering all the areas listed above, the high-performance computing (HPC) area is essential in all of them. If a solution or algorithm cannot be executed in a feasible time, independently of how useful or innovative it may be, it can not be used.

In conclusion, all the bioinformatics problems and algorithms could be seen as a target for HPC since time is one of the most critical points in this area. Also, considering the increase in the data volume and complexity, even the faster algorithms for our current data could be insufficient in the future.

5.6. References

[1] Cristianini, N. and Hahn, M.W., 2006. Introduction to computational genomics: a case studies approach. Cambridge University Press.

[2] Stevens, Tim J., and Wayne Boucher. Python programming for biology. Cambridge University Press, 2015.

[3] Klaczko, Louis B., and Blanche C. Bitner-Mathe. "On the edge of a wing." Nature 346.6282 (1990): 321-321.

[4] Clark, Dominic A., Geoffrey J. Barton, and Christopher J. Rawlings. "A knowledge-based architecture for protein sequence analysis and structure prediction." Journal of Molecular Graphics 8.2 (1990): 94-107.

[5] Ma, Jianzhu, and Sheng Wang. "Algorithms, applications, and challenges of protein structure alignment." Advances in Protein Chemistry and Structural Biology 94 (2014): 121-175.

[6] Baldi, P. and Brunak, S., 2001. bioinformatics: the machine learning approach. MIT press.

[7] Gollery, M., 2005. Bioinformatics: sequence and genome analysis. Clinical Chemistry, 51(11), pp.2219-2220.

[8] Jones, N.C. and Pevzner, P.A., 2004. An introduction to bioinformatics algorithms. MIT press.

[9] Lehninger, A.L., Nelson, D.L., Cox, M.M. and Cox, M.M., 2005. Lehninger principles of biochemistry. Macmillan.

[10] Zomaya, A.Y., 2005. Parallel computing for bioinformatics and computational biology. Wiley.

[11] Preetha J. Shetty, 2020. The Evolution of DNA Extraction Methods. American Journal of Biomedical Science & Research.

[12] Anandika Dhaliwal, 2013. DNA Extraction and Purification. Materials and Methods volume 3.

[13] Immy Mobley, 2021. https://frontlinegenomics.com/dna-sequencing-how-to-choose-the-right-technology/.

[14] Immy Mobley, 2021. https://frontlinegenomics.com/long-read-sequencing-vs-short-read-sequencing/.

[15] Morisse, Pierre and Marchet, Camille and Limasset, Antoine and Lecroq, Thierry and Lefebvre, Arnaud, 2020. ONSENT: Scalable long read self-correction and assembly polishing with multiple sequence alignment. Cold Spring Harbor Laboratory.

[16] C. elegans Sequencing Consortium, 1998. Genome sequence of the nematode C. elegans: a platform for investigating biology. Science (New York, N.Y.), 282(5396), 2012–2018.

[17] Carvalho, Antonio Bernardo and Dupim, Eduardo G. and Nassar, Gabriel, 2016. Improved assembly of noisy long reads by k-mer validation. Cold Spring Harbor Laboratory.

[18] Ekblom, R. and Wolf, J.B.W. (2014), A field guide to whole-genome sequencing, assembly and annotation. Evol Appl, 7: 1026-1042

[19] Smith T.F., Waterman M.S., and Burks C. 1985. The statistical distribution of nucleic acid similarities. Nucleic Acids Res. 13:645-656.

[20] Xia, Zeyu, et al. "A review of parallel implementations for the smith–waterman algorithm." Interdisciplinary Sciences: Computational Life Sciences (2021): 1-14.

[21] James M. Heather, Benjamin Chain. The sequence of sequencers: The history of sequencing DNA, Genomics, Volume 107, Issue 1, 2016, Pages 1-8.

[22] Alice Maria Giani, Guido Roberto Gallo, Luca Gianfranceschi, Giulio Formenti, Long walk to genomics: History and current approaches to genome sequencing and assembly. Computational and Structural Biotechnology Journal. Volume 18, 2020, Pages 9-19

[23] Butenhof, David R. Programming with POSIX threads. Addison-Wesley Professional, 1997.

[24] Chandra, Rohit, et al. Parallel programming in OpenMP. Morgan kaufmann, 2001.

[25] Gauthier, Michel. (2007). Simulation of polymer translocation through small channels: A molecular dynamics study and a new Monte Carlo approach.

[26] Wozniak A (1997) Using video-oriented instructions to speed up sequence comparison. Bioinformatics 13(2):145–150. https://doi.org/10.1093/bioinformatics/13.2.145

[27] Rognes T, Seeberg E (2000) Six-fold speed-up of smith-waterman sequence database searches using parallel processing on common microprocessors. Bioinformatics 16(8):699–706. https://doi.org/ 10.1093/bioinformatics/16.8.699

[28] Rognes T (2011) Faster smith-waterman database searches with inter-sequence simd parallelisation. BMC Bioinform 12(1):1–11. https://doi.org/10.1186/1471-2105-12-221

[29] H. Zwart (2015) "Human Genome Project: history and assessment". In: International Encyclopedia of Social Behavioral Sciences. 2nd ed. Oxford: Elsevier, 311–317.

[30] Sergey Nurk, et al. The complete sequence of a human genome (2022). Science 376(6588):44-53. https://doi.org/10.1126/science.abj6987

[31] Farrar M (2007) Striped smith-waterman speeds database searches six times over other simd implementations. Bioinformatics 23(2):156–161. https://doi.org/10.1093/bioinformatics/btl582

[32] Thompson,J.D., Higgins,D.G. and Gibson,T.J. (1994) CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. Nucleic Acids Res., 22, 4673–4680.

[33] Gut, I.G. New sequencing technologies. Clin Transl Oncol 15, 879–881 (2013). https://doi.org/10.1007/s12094-013-1073-6

[34] Dr. Josh Wang, Advancing genomics, medicine and health together – by semiconductor DNA synthesis technology, https://www.genscript.com/advancing-genomics-medicine-and-health-together-by-semiconductor-dna-synthesis-technology-summary.html

[35] Mikhailov,D., Cofer,H. and Gomperts,R. (2001) Performance opti- mization of Clustal W: parallel Clustal W, HT Clustal, and MUL- TICLUSTAL, White papers, Silicon Graphics, Mountain View,CA

[36] Li, Kuo-Bin. "ClustalW-MPI: ClustalW analysis using distributed and parallel computing." Bioinformatics 19.12 (2003): 1585-1586.

[37] Henikoff, Steven, and Jorja G. Henikoff. "Amino acid substitution matrices from protein blocks." Proceedings of the National Academy of Sciences 89.22 (1992): 10915-10919.

[38] Expósito, R. R., J. Veiga, J. González Domínguez and J. Touriño, 2017. Mardre: Efficient mapreduce based removal of duplicate DNA reads in the cloud. Bioinformatics, 33(17): 2762 - 2764.

[39] Xu, H., X. Luo, J. Qian, X. Pang, J. Song, G. Qian, J. Chen and S. Chen, 2012. Fastuniq: A fast de novo duplicates removal tool for paired short reads. PloS one, 7(12): e52249.

[40] Costa, Gustavo GL, et al. "The mitochondrial genome of Moniliophthora roreri, the frosty pod rot pathogen of cacao." Fungal Biology 116.5 (2012): 551-562.

[41] Watson, James D. "The human genome project: past, present, and future." Science 248.4951 (1990): 44-49.

[42] Xiao, Tiantian Zhou, Wenhao. (2020). The third generation sequencing: The advanced approach to genetic diseases. Translational Pediatrics. 9. 163-173. 10.21037/tp.2020.03.06.

[43] Wenger, A.M., Peluso, P., Rowell, W.J. et al. Accurate circular consensus long-read sequencing improves variant detection and assembly of a human genome. Nat Biotechnol 37, 1155–1162 (2019). https://doi.org/10.1038/s41587-019-0217-9

[44] Anthony Rhoads, Kin Fai Au, PacBio Sequencing and Its Applications, Genomics, Proteomics Bioinformatics, Volume 13, Issue 5, 2015, Pages 278-289, ISSN 1672-0229, https://doi.org/10.1016/j.gpb.2015.08.002.

[45] PacBio, New Chemistry Boosts Average Read Length to 10 kb – 15 kb for PacBio® RS II, 2014, https://www.pacb.com/blog/new-chemistry-boosts-average-read/

[46] By Erekevan - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=116777958

[47] Morisse, Pierre and Marchet, Camille and Limasset, Antoine and Lecroq, Thierry and Lefebvre, Arnaud, 2020, CONSENT: Scalable long read self-correction and assembly polishing with multiple sequence alignment, bioRxiv - Cold Spring Harbor Laboratory. https://doi.org/10.1101/546630

[48] Brzinsky-Fay, Christian, and Ulrich Kohler. "New developments in sequence analysis." Sociological methods research 38.3 (2010): 359-364.

[49] Hu, Y., Colantonio, V., Müller, B.S.F. et al. Genome assembly and population genomic analysis provide insights into the evolution of modern sweet corn. Nat Commun 12, 1227 (2021). https://doi.org/10.1038/s41467-021-21380-4

[50] Jorde, Lynn B., and Michael J. Bamshad. "Genetic ancestry testing: What is it and why is it important?." JAMA 323.11 (2020): 1089-1090.

[51] Li H (2018) Minimap2: pairwise alignment for nucleotide sequences. Bioinformatics 34(18):3094–3100. https:// doi. org/ 10.1093/bioinformatics/bty191

[52] Suzuki H, Kasahara M (2018) Introducing difference recurrence relations for faster semi-global alignment of long sequences. BMC Bioinform 19(1):33–47. https:// doi. org/ 10. 1186/ s12859-018-2014-8

[53] Rognes T (2011) Faster smith-waterman database searches with inter-sequence simd parallelisation. BMC Bioinform 12(1):1–11. https://doi.org/10.1186/1471-2105-12-221

[54] Rahn R, Budach S, Costanza P, Ehrhardt M, Hancox J, Reinert K (2018) Generic accelerated sequence alignment in seqan using vectorization and multi-threading. Bioinformatics 34(20):3437– 3445. https://doi.org/10.1093/bioinformatics/bty380

[55] Rognes T (2011) Faster smith-waterman database searches with inter-sequence simd parallelisation. BMC Bioinform 12(1):1–11. https://doi.org/10.1186/1471-2105-12-221

Capítulo 6

Ferramentas para Configuração e Gerenciamento de Cluster de Alto Desempenho em Nuvem Pública

Albino A. Aveleda (UFRJ) e Alvaro L.G.A. Coutinho (UFRJ)

Abstract

The High-Performance Computing Center (NACAD) of COPPE / UFRJ is a laboratory specialized in applying high-performance computing to engineering and science problems in general. NACAD also has extensive experience in the administration, management, and tools development to support the Supercomputing Center and to develop and implement innovations in the machine management environment. The mini-course proposes sharing some of the best practices adopted by NACAD-COPPE/UFRJ for deploying an HPC cluster using a public cloud.

Resumo

O Núcleo Avançado de Computação de Alto Desempenho (NACAD) da COPPE/UFRJ é um laboratório especializado na aplicação de computação de alto desempenho a problemas de engenharia e ciências em geral. O NACAD também possui grande experiência na administração, gerencia e ferramentas de apoio ao Centro de Supercomputação e de desenvolver e implementar inovações no ambiente de administração e gerencia da máquina. O presente minicurso propõe compartilhar algumas dessas melhores práticas adotadas pelo NACAD-COPPE/UFRJ para a implantação de um cluster de Alto Desempenho usando uma nuvem pública.

Palavras-chave*: cluster, computação de alto desempenho, nuvem pública, computação em nuvem*

6.1. Introdução

O presente minicurso tem a proposta de compartilhar melhores práticas de como utilizar uma ferramenta para fazer a implantação de um cluster de Alto Desempenho usando uma Nuvem Pública, utilizando a ferramenta de gerenciamento de cluster de código aberto conhecida como *AWS Parallel Cluster*.

Apesar da ideia de computação em nuvem ser conhecida há décadas, ela de fato começou a ser oferecida comercialmente entre 2006 e 2008 através da *Amazon* quando lançou o seu produto *Amazon Web Service* (AWS). Aos poucos, ela foi se popularizando através de vários provedores, tais como: AWS, *Microsoft Azure*, *Google Cloud Platform* (GCP), *IBM Cloud*, *Oracle Cloud*, *Alibaba Cloud* e muitos outras ao redor do mundo. Os maiores provedores de nuvem pública possuem um alcance mundial. Basicamente, eles são divididos por regiões e zonas de disponibilidades. Por exemplo, na AWS pode-se citar algumas regiões, tais como: Oregon, São Paulo, Irlanda, Tóquio, etc. Sendo que cada região possui em média três zonas de disponibilidades. Cada zona de disponibilidade é um datacentro e tem pelo menos 100 Km de distância entre elas. Hoje em dia, grande parte da população mundial usa algum serviço na nuvem.

No caso particular do uso da nuvem para aplicações de computação de alto desempenho (HPC, do inglês, *High-Performance Computing*), o processo foi um pouco diferente. Desde o início houve uma certa resistência em se utilizar uma nuvem pública para fazer o processamento de HPC, já que as aplicações normalmente possuem requisitos bem específicos tais como: necessidade de um hardware de alto desempenho, várias conexões e topologias de rede, alta banda e baixa latência de comunicação, alta taxa de leitura e escrita dos dados, etc. Além disso, a configuração de cada nó de processamento é otimizada e configurada com diversas bibliotecas otimizadas. Consequentemente, todos esses requisitos dificultavam o uso da nuvem. Aos poucos esses itens foram sendo endereçados por alguns provedores, facilitando assim o uso de uma nuvem pública.

Em relação ao hardware, vários provedores fizeram investimentos a fim de atender a esta demanda. Seguem abaixo alguns exemplos:

- AWS *Elastic Fabric Adapter* (EFA) é uma interface de rede que pode ser conectada a algumas instâncias. Sua interface de hardware faz um desvio do sistema operacional (SO) e melhora o desempenho das comunicações entre instâncias. Com o EFA, os aplicativos de HPC que usam o MPI (*Message Passing Interface*) e os aplicativos de aprendizado de máquina (ML) usando NCLL (*NVIDIA Collective Communications Library*) podem ser dimensionados para milhares de CPUs ou GPUs.
- *Amazon FSx for Lustre* é o serviço totalmente gerenciado e compartilhado com escalabilidade e performance do sistema de arquivo paralelo Lustre [3], um dos mais comuns em ambientes de HPC.
- A *Microsoft Azure* possui algumas instâncias com acesso à rede *Infiniband* que possui alta largura de banda e baixa latência. A Microsoft e a HPE-Cray, possuem uma parceria que onde é possível disponibilizar um cluster da HPE-Cray dentro da infraestrutura da Microsoft Azure.

Além do hardware e bibliotecas otimizadas, não haviam ferramentas que auxiliassem a implementação de um cluster na nuvem. Uma das primeiras ferramentas neste sentido foi o *StarCluster* [*1*] desenvolvido pelo MIT (*Massachusetts Institute of*

Technology). O desenvolvimento do *StarCluster* parou depois de 2014. Logo depois a AWS lançou o seu software que era chamado de *CfnCluster*. Posteriormente, ele foi migrado para o *Parallel Cluster* [2], que será abordado neste minicurso em mais detalhes. Pode-se citar outro exemplo, o *Google HPC Toolkit*, que permite iniciar *jobs* rapidamente com desempenho previsível através de VMs (*virtual machines*) de HPC pré-configuradas.

Este minicurso foca no uso de ferramentas para implantar e gerenciar clusters de HPC na Nuvem AWS (*Amazon Web Services*). O cluster de HPC na nuvem tem uma estrutura semelhante à maioria dos clusters tradicionais, isto é, uma instância fazendo o papel de *master node* (com sistema de fila para receber solicitações de *jobs* e outras configurações) e as demais instâncias atuando como *compute node,* que são nós de processamento alocados de acordo com a demanda ou de acordo com o que foi definido na configuração. Essas ferramentas permitem usufruir dos recursos e das melhores características da nuvem, como por exemplo, alocação dinâmica da quantidade de instâncias de forma elástica à carga de trabalho do cluster, permitindo assim que o cluster aumente e diminua automaticamente o número de instâncias computacionais.

Sendo assim, de forma a conduzir esta exposição da forma mais clara possível, este texto está organizado da seguinte forma:

- A seção 7.2 apresenta uma introdução a computação em nuvem fazendo uma comparação entre um cluster tradicional versus um cluster numa nuvem e suas respectivas ferramentas;
- A seção 7.3 aborda as principais características dos serviços utilizados na nuvem para uma implementação de um cluster de HPC (VPC, instâncias, armazenamento, autenticação, etc.);
- A seção 7.4 mostra o uma visão geral da configuração de um cluster de forma manual ou com uso de ferramentas para implementação do cluster de HPC na nuvem pública;
- A seção 7.5 discute alguns exemplos de implementação do cluster de HPC na nuvem pública;
- A seção 7.6 faz as considerações finais.

6.2. Cluster tradicional versus Cluster na nuvem

6.2.1. Cluster Tradicional

Existem diversos desenhos de topologias possíveis e de como provisionar um cluster de HPC. A Figura 1 mostra, de forma simplificada, um cluster tradicional de HPC. Nesta figura pode-se ver um cluster formado por um *headnode*, responsável pela gerencia e acesso ao cluster, *nodes* que são os nós de processamento, e switches para a comunicação intra-cluster. Nesta figura não foi incluído um ou mais servidores de armazenamento a fim de simplificar o desenho. A área de armazenamento do cluster pode ser feita no próprio *headnode*, caso seja um cluster pequeno, ou de um ou mais servidores de armazenamento externo, e que normalmente são conectados na rede de alto desempenho através do switch HPC.

Neste exemplo, o *headnode* exerce algumas funções tais como: servidor de acesso e gerenciamento do cluster, *job scheduler*, entre outras. Entretanto, dependendo do tamanho do cluster, várias dessas funções podem ser distribuídas por vários servidores. Por exemplo, o *headnode* pode fazer o papel de gerenciamento e de *job scheduler*, enquanto um ou mais servidores podem servir como servidor de acesso aos usuários para compilação e submissão de *jobs* (tarefas).

Nesta topologia, o *headnode* se conecta à Internet e ao switch interno do cluster para se comunicar com os nós computacionais a fim de alocar ou monitorar os recursos disponíveis no cluster. Em alguns clusters também existem uma ou mais redes de alto desempenho, que possuem uma maior banda e uma menor latência de comunicação, como ilustrado na Figura 1 através do Switch HPC. Por exemplo, switches *Infiniband*, *Slingshot*, etc. Normalmente essa rede é utilizada para comunicações entre nós de processamento, tais como: MPI (*Message Passing Interface*), acesso ao servidor de armazenamento, etc.

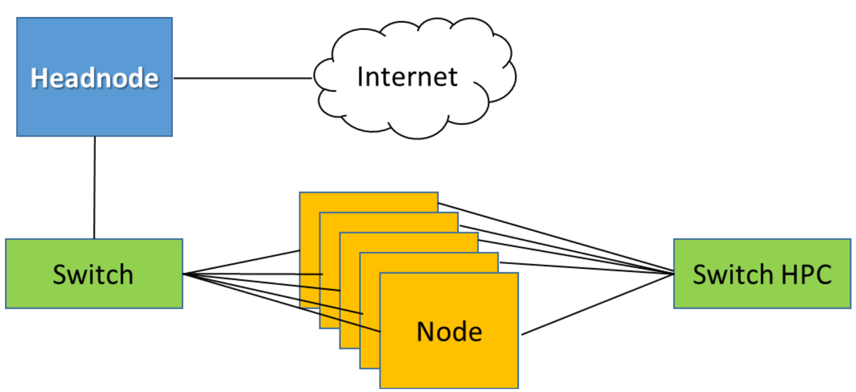


Figura 1: Topologia simplificada de um cluster de HPC

6.2.2. Cluster na Nuvem

Um cluster em uma nuvem pública também pode utilizar uma topologia semelhante à mostrada na Figura 1, porém a nuvem possui algumas peculiaridades que a tornam diferente de um cluster tradicional. Enquanto que um cluster tradicional possui seu custo de aquisição, manutenção e operação, no cluster na nuvem você paga pelo que consome e pode provisionar a qualquer momento. A Tabela 1 mostra as principais diferenças entre um cluster tradicional e um cluster em uma nuvem pública.

	Cluster tradicional	Cluster na nuvem
Quantidade de nós de processamento	Fixo	Variável
Responsabilidade sobre segurança	Total	Compartilhada
Redes	LANs e VLANs	VPCs
Rede de alto desempenho	Pode ter ou não	Depende do provedor
Armazenamento (capacidade)	Fixo	Variável
Custo	Parcela fixa + variável (consumo energético)	Paga apenas pelo que consome

Tabela 1: Resumo das principais diferenças entre um cluster tradicional e um cluster na nuvem pública

Em geral, os provedores possuem um limite em função do tipo de instância e da quantidade que pode ser alocada para evitar supressas na conta do usuário. No caso da AWS, esse limite costuma ser 20 instâncias, dependendo da capacidade de processamento com base no número de unidades de processamento central virtuais (vCPUs). Entretanto, esse limite pode ser aumentado facilmente, bastando fazer uma solicitação ao suporte da AWS.

Para exemplificar a diferença entre os clusters, suponha que para resolver um problema é necessário fazer 8 execuções de um programa MPI que utiliza 5 nós de processamento em cada rodada. Sendo que cada execução demora em média 1 hora e meia. Por parte do usuário, ele simplesmente submete os 8 jobs MPI na fila do cluster.

- Usando um cluster tradicional com 10 nós de processamento: Neste caso, serão executados dois *jobs* ao mesmo tempo. Logo, para fazer a execução dos 8 casos seriam necessárias 6 horas de processamento (8 jobs /2 jobs por vez x 1,5h=6h).
- Usando um cluster na nuvem: como o cluster pode alocar dinamicamente os nós de processamento (instâncias), pode-se executar os 8 *jobs* ao mesmo tempo e ter o resultado em apenas 1,5 hora. Em outras palavras, pode-se alocar 40 instâncias (8 jobs x 5 instâncias) de uma vez. Em relação ao custo na nuvem é indiferente usar 10 instâncias por 6 horas ou usar 40 instâncias por 1,5 hora, já que atualmente a unidade de consumo é medida em segundos.

Cabe aqui lembrar que no início da AWS o cálculo não era tão simples assim. A unidade de custo de cada instância era por hora o que complicava um pouco mais a estimativa de custo. Na época, o *StarCluster* também endereçava esse problema de otimização de custos. Para facilitar o entendimento, suponha o mesmo problema anterior e que o custo da instância fosse $0,10/h. Ao usar as mesmas 10 instâncias fixas, semelhante ao cluster tradicional, se poderia ter uma vantagem em relação a alocação de todas as instâncias. Neste caso, usando 10 instâncias durante 6 horas o custo seria de $6,00. Porém, ao usar 40 instâncias por 1,5 hora, o custo seria de 40 instâncias por 2 horas, já que a unidade mínima era de 1 hora, o que daria $8,00 de custo. A evolução do tempo e concorrência fizeram que a AWS mudasse a unidade de cobrança, beneficiando assim seus clientes.

6.3. Serviços utilizados no AWS *ParallelCluster*

Atualmente existem centenas de serviços disponíveis em uma nuvem pública e no caso particular da AWS, além de ser a mais adotada no mundo, é a que oferece mais serviços aos seus clientes. O *AWS ParallelCluster* em si é gratuito, o que é cobrado são os recursos alocados por ele. Em particular o *AWS ParallelCluster* [*2]* utiliza ou pode utilizar os seguintes serviços descritos abaixo em ordem alfabética.

- *Batch*
  Serviço gerenciado de *job scheduler*, função semelhante ao SLURM (*Simple Linux Utility for Resource Management*). Ele provisiona dinamicamente a quantidade e o tipo de recursos de computação (por exemplo, CPU, GPU, instâncias otimizadas para memória, etc.). Com *AWS Batch* não é necessário instalar ou gerenciar softwares de computação em lote, como o SLURM, ou clusters de servidor adicionais para executar seus trabalhos de maneira eficaz.

- *CloudFormation*
  Serviço de infraestrutura como código que permite provisionar recursos na AWS e de terceiros no ambiente de nuvem. É o principal serviço usado pelo *ParallelCluster*. Cada cluster é representado como uma pilha e todos os recursos necessários para cada cluster são definidos dentro do *ParallelCluster template*. As instâncias que são executadas em um cluster fazem chamadas de HTTPS para o *CloudFormation endpoint* na região onde o cluster é alocado.

- *CloudWatch*
  Serviço de monitoramento e observabilidade que fornece dados e insights acionáveis. Esses insights podem ser usados para monitorar os aplicativos, responder a alterações de desempenho e exceções de serviço, e otimizar a utilização de recursos. O *CloudWatch* é usado como um painel, para monitorar e registrar em logs as etapas de criação da imagem em *Docker* e a saída dos *jobs* da *AWS Batch*.

- *CloudWatch Logs*
  É um dos principais recursos do *CloudWatch*. Utilizado para monitorar, armazenar, exibir e pesquisar os arquivos de log de muitos dos componentes usados pelo *ParallelCluster*.

- *CodeBuild*
  Serviço de integração contínuo gerenciado que compila o código-fonte, executa testes e produz pacotes que estão prontos para implantação. O *CodeBuild* é usado para criar imagens do *Docker* de forma automática e transparente quando os clusters são criados e é usado somente com *AWS Batch*.

- *DynamoDB*
  Serviço de banco de dados *NoSQL* rápido e flexível. Ele é usado para armazenar as informações mínimas do estado do cluster. O *headnode* principal rastreia instâncias provisionadas em uma tabela do *DynamoDB*.
- *Elastic Compute Cloud* (EC2)
  Serviço que oferece a capacidade de computação para o cluster. O nó *headnode* e os nós de computação são instâncias do EC2. Qualquer tipo de instância que ofereça suporte ao HVM pode ser selecionada. O *headnode* e os nós de computação podem ser de diferentes tipos de instância. Entretanto, cabe aqui uma observação, caso o *headnode* também faça o serviço de NFS (*Network File System)* para os nós de processamento, deve-se atentar para verificar se a instância selecionada também possui a mesma banda de comunicação de rede. Dependendo do tamanho da instância, ela pode ter uma banda de rede diferente. Além disso, se forem usadas várias filas, alguns ou todos os nós de computação também poderão ser executados como uma instância *spot*. As instâncias *spot* do EC2 permitem aproveitar a capacidade não utilizada do EC2 na nuvem AWS. Em comparação com a definição de preço sob demanda, as instâncias *spot* oferecem descontos de até 90%.
- *Elastic Container Registry* (ECR)
  Serviço de registro de contêiner do *Docker* totalmente gerenciado que facilita o armazenamento, o gerenciamento e a implantação de imagens de contêiner do *Docker*. O ECR armazena as imagens do *Docker* que são criadas quando os clusters são criados. As imagens do *Docker* são então usadas pelo *AWS Batch* para executar os contêineres para os trabalhos enviados. O ECR é usado somente com o *AWS Batch*.
- *Identity and Access Management* (IAM)
  Serviço usado para fornecer um papel do IAM menos privilegiado para as instâncias EC2 de cada cluster. As instâncias recebem acesso apenas às chamadas de API específicas que são necessárias para implantar e gerenciar o cluster. As funções do IAM também são criadas para os componentes envolvidos no processo de criação de imagem do *Docker* quando os clusters são criados. Esses componentes incluem as funções do *Lambda* que têm permissão para adicionar imagens do *Docker* ao repositório do ECR. Eles também incluem as funções que permitem excluir o *bucket* do S3 que é criado para o cluster e o projeto *CodeBuild*. Também há funções para recursos, instâncias e tarefas do *AWS Batch*.
- *Lambda*
  Serviço de computação sem servidor e orientado a eventos que permite executar as funções que orquestram a criação de imagens do *Docker*. O *Lambda* também gerencia a limpeza de recursos de cluster personalizado, como imagens do *Docker* armazenadas no repositório do ECR e no S3.
- NICE DCV
  O NICE DCV é um protocolo de exibição remoto de alto desempenho que fornece uma maneira segura de fornecer desktops remotos e streaming de aplicativos para qualquer dispositivo em diferentes condições de rede.
- *Route 53*

Serviço de *Domain Name System* (DNS) na nuvem altamente disponível e escalável. Ele é usado para criar zonas hospedadas com *hostnames* e *fully qualified domain* para cada um dos nós de computação.

- *Virtual Private Cloud* (VPC)
  É um serviço que permite criar uma nuvem privada localizada dentro de uma nuvem pública. A VPC define uma rede usada pelos nós do cluster.

Armazenamento:

Existem quatro tipos de armazenamento, que podem ser usados no *ParallelCluster*, que estão descritos a seguir.

- *Elastic Block Store* (EBS)
  Serviço que disponibiliza volumes de armazenamento persistente em blocos de alto desempenho projetado para o EC2, isto é, normalmente utilizado para o sistema operacional e outras aplicações.
- *Elastic File System* (EFS)
  Serviço que oferece um sistema de arquivos NFS elástico, simples, escalável e totalmente gerenciado para uso na nuvem.
- *FSx for Lustre*
  O FSx for Lustre fornece um sistema de arquivos distribuído, paralelo e de alto desempenho que usa o sistema de arquivos Lustre [3] de código aberto.
- *Simple Storage Service* (S3)
  Serviço de armazenamento de objetos que oferece escalabilidade, disponibilidade de dados, segurança e performance. Com classes de armazenamento econômicas e recursos de gerenciamento fáceis de usar, pode-se otimizar custos, organizar dados e configurar controles de acesso ajustados para atender a requisitos específicos de negócios, organizacionais e de conformidade.

Serviços que foram removidos a partir da versão 2.11.5 do *ParallelCluster*:

- *Auto Scaling*
  Serviço que monitora os aplicativos e ajusta automaticamente a capacidade para manter um desempenho constante e previsível pelo menor custo possível.
- *Simple Notification Service* (SNS)
  Serviço de mensagens totalmente gerenciado para a comunicação de aplicação para aplicação (A2A) e de aplicação para pessoa (A2P).
- *Simple Queue Service* (SQS)
  É um serviço de filas de mensagens gerenciado que permite o desacoplamento e a escalabilidade de micros serviços, sistemas distribuídos e aplicações sem servidor.

6.4. Configuração do cluster

6.4.1. Visão Geral

A implementação de forma manual de um cluster, apesar de ser possível, é trabalhosa e podem ocorrer em alguns erros durante o processo da implantação. No caso de se usar uma infraestrutura na nuvem, mesmo sem um software de gerenciamento de cluster (o que acontece em alguns provedores), pode-se otimizar e minimizar a possibilidade de erros durante a implantação. Basta usar a infraestrutura como código (IaC - *Infrastructure as Code*), que é um dos recursos disponíveis na nuvem, e ferramentas DevOps (*Development and Operations*).

Além das ferramentas de infraestrutura fornecida pelos próprios provedores tais como: *AWS CloudFormation*, *Azure Resource Manager*, *Google Cloud Deployment Manager*, etc. Existem diversas outras ferramentas que podem ser usadas em clusters na nuvem ou *on-premise*. A seguir são listadas algumas delas.

- Terraform [4]: É uma das ferramentas de IaC mais populares do mercado. É um projeto de código aberto com flexibilidade, suportando as principais plataformas de nuvem, incluindo; AWS, GCP, Azure, DigitalOcean, GitHub, Cloudflare, OpenStack e muitos outros. Além disso, o Terraform também permite a manipulação de recursos por meio do controle de origem. Esse recurso é essencial ao manipular nuvens híbridas, onde os planos podem ser feitos em vários provedores e infraestruturas de nuvem, tudo isso usando o mesmo fluxo de trabalho.
- Pulumi [5]: É uma ferramenta de IaC que se diferencia das demais plataformas por oferecer maior flexibilidade. Ele suporta várias linguagens de programação, como Python, JavaScript, C#, Go e TypeScript. Não possui a mesma portabilidade que o Terraform, porém oferece suporte para os principais provedores da nuvem, como: AWS, GCP e Azure.
- Ansible [6]: É uma ferramenta de orquestração e configuração da Red Hat. O *Ansible* usa módulos de configuração chamados "*Playbooks*" escritos em YAML (*YAML Ain't Markup Language*), onde você pode configurar o estado final desejado de sua infraestrutura. O *Ansible* melhora o desenvolvimento automatizando muitas tarefas repetitivas e complexas, economizando muito tempo ao instalar pacotes ou configurar um grande número de servidores.
- Chef [7]: É uma das ferramentas de infraestrutura como código mais populares no mercado. Esta ferramenta usa "*recipes*" e " *cookbooks*" contando com uma linguagem específica de domínio (DSL) baseada em Ruby. O *Chef* é independente da nuvem, trabalhando com grandes provedores de nuvem, como AWS, GCP e Azure Cloud e também suporta APIs de provisionamento permitindo ser usada em conjunto com o *Terraform*. Sua flexibilidade é escalável e aplicável em qualquer pipeline de CI/CD (*Continuous Integration/Continuous Delivery*) existente.
- Puppet [8]: O Puppet tem muitas semelhanças com o Chef e faz parte da base de muitos pipelines de CI/CD criados por engenheiros de DevOps. Ele também usa uma DSL baseada em Ruby, onde você pode declarar o estado final de sua infraestrutura e o que deseja que ela faça. Se ocorrer algum desvio de configuração

após esse ponto, o Puppet monitora e corrige automaticamente quaisquer alterações incorretas. Atualmente, esse projeto de código aberto oferece suporte a todas as plataformas de nuvem proeminentes, como AWS, GCP, Azure Cloud, permitindo a automação em vários provedores.

- Crossplane [9]: É uma ferramenta de código aberto *Kubernetes Infrastructure as Code* que oferece suporte a todos os principais provedores de nuvem. Ele visa gerenciar e provisionar infraestruturas e serviços em nuvem usando o *kubectl*.

Além das ferramentas acima alguns provedores oferecem alguns serviços que se integram com as ferramentas descritas anteriormente, como por exemplo o *AWS OpsWorks*, que é um serviço de gerenciamento de configurações que permite as instâncias sejam gerenciadas tanto pelo *Chef* como pelo *Puppet*. Outros provedores oferecem essas ferramentas em seus *marketplaces*.

6.4.2. Boas práticas para rodar HPC na AWS

A seguir são descritas as melhores práticas para a execução de uma aplicação de HPC na infraestrutura da AWS.

- Gerenciar clusters com o *AWS Parallel Cluster*. Esta ferramenta de código aberto é totalmente mantida e suportada pela AWS, tornando mais fácil gerenciar e implementar clusters de HPC na AWS.
- Usar um *Placement Group* que são agrupamentos lógicos de instâncias na mesma zona de disponibilidade (AZ). O uso do *Placement Group* oferece conectividade de alta taxa de transferência e baixa latência entre instâncias do mesmo grupo, especialmente quando a maior parte do tráfego de rede está limitado às instâncias do grupo, como é o caso de um cluster.
- Evite usar *Hyper-Threading* (HT), a tecnologia Intel HT permite que várias *threads* sejam executadas simultaneamente no mesmo núcleo de CPU Intel Xeon, com cada *thread* atua como uma vCPU em uma instância do EC2. Para trabalhos de HPC, o *hyper-threading* pode ser lento, especialmente se o trabalho exigir cálculos de ponto flutuante. Cada núcleo possui dois threads que compartilham a mesma unidade de processamento (Unidade Lógica e Aritmética, cache, etc) e podem, assim, bloquear um ao outro. Certifique-se de que o HT esteja desabilitado nas instâncias do EC2.
- Use uma conexão rápida entre nós. Algumas instâncias do EC2 podem fornecer uma taxa de transferência de rede de até 100 Gbps. Estas instâncias podem beneficiar aplicações que dependem de uma comunicação rápida, *data lakes* e caches de memória.
- Usar instâncias do EC2 com GPU para tarefas gráficas ou de Machine Learning (ML). As instâncias P3 fornecem HPC baseado em nuvem com taxa de transferência de rede de até 100 Gbps e oito GPUs NVIDIA V100 Tensor Core para aplicativos de HPC e ML.
- Use o *Elastic Fabric Adapter* (EFA), esta interface de rede para instâncias do EC2 permite que os clientes executem aplicativos HPC que exigem comunicação intensa entre instâncias. Esta interface possui uma latência menor e mais

consistente e maior *throughput* do que os canais tradicionais de TCP, permitindo sua melhor escalabilidade. Entretanto apenas alguns tipos de instâncias suportam o EFA. Recomenda-se o uso junto com o *Placement Group*.

- Usar o *Amazon FSx for Lustre*, sistema de arquivos paralelo, para cargas de trabalho de ML e HPC. Esse sistema de arquivos totalmente gerenciado permite que você inicie um sistema de arquivos *Lustre* para processar grandes volumes de dados com uma taxa de transferência de centenas de GB/s e milhões de IOPS, garantindo latência abaixo de milissegundos.
- Escolha o tipo de instância certo para sua aplicação. Use instâncias otimizadas para computação ou de uso geral para aplicativos vinculados à CPU que exigem processadores de alto desempenho.

6.5. Implementação do cluster de HPC

Este minicurso irá focar na versão 3.x do *ParallelCluster*. A versão 2.x possui arquivos de configuração diferentes e estes não serão abordados aqui. Caso deseje utilizar as duas versões, recomenda-se o uso de um ambiente virtual para isolar ambos pacotes.

6.5.1 Instalação do *ParallelCluster*

Esta instalação no Linux pode ser feita no notebook, servidor, máquina virtual, na nuvem e praticamente em qualquer lugar. Recomenda-se o uso de um ambiente virtual para a instalação. Caso o *virtualenv* não esteja instalado execute os comandos abaixo.

```
[linux@ip ~]$ python3 -m pip install --upgrade pip
Successfully installed pip-22.2.2
[linux@ip ~]$ python3 -m pip install --user --upgrade virtualenv
```

Crie um ambiente virtual chamado, por exemplo, *pcluster* e ative ele, conforme mostrado abaixo. Repare que o *prompt* do comando mudou em função da ativação do ambiente virtual.

```
[linux@ip ~]$ python3 -m virtualenv ~/env/pcluster
[linux@ip ~]$ source ~/env/pcluster/bin/activate
(pcluster)[linux@ip ~]$
```

Caso ainda não esteja instalado, instale o AWS CLI, com os comandos a seguir.

```
(pcluster)[linux@ip ~]$ curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o "awscliv2.zip"
(pcluster)[linux@ip ~]$ unzip awscliv2.zip
(pcluster)[linux@ip ~]$ sudo ./aws/install
```

Configure o AWS CLI para permitir acesso a nuvem da AWS, mas primeiro é necessário se entrar na console da AWS (Figura 2) e criar as chaves de acesso. Entre no serviço IAM selecionando o ícone de matriz ou digitando o serviço, como mostrado na Figura 3, selecione seu usuário, depois selecione a aba "*Credenciais de segurança*", e por fim clique no botão "*Criar chave de acesso*", conforme ilustrado na Figura 4.

Figura 2: Pagina http://aws.amazon.com para acesso à console

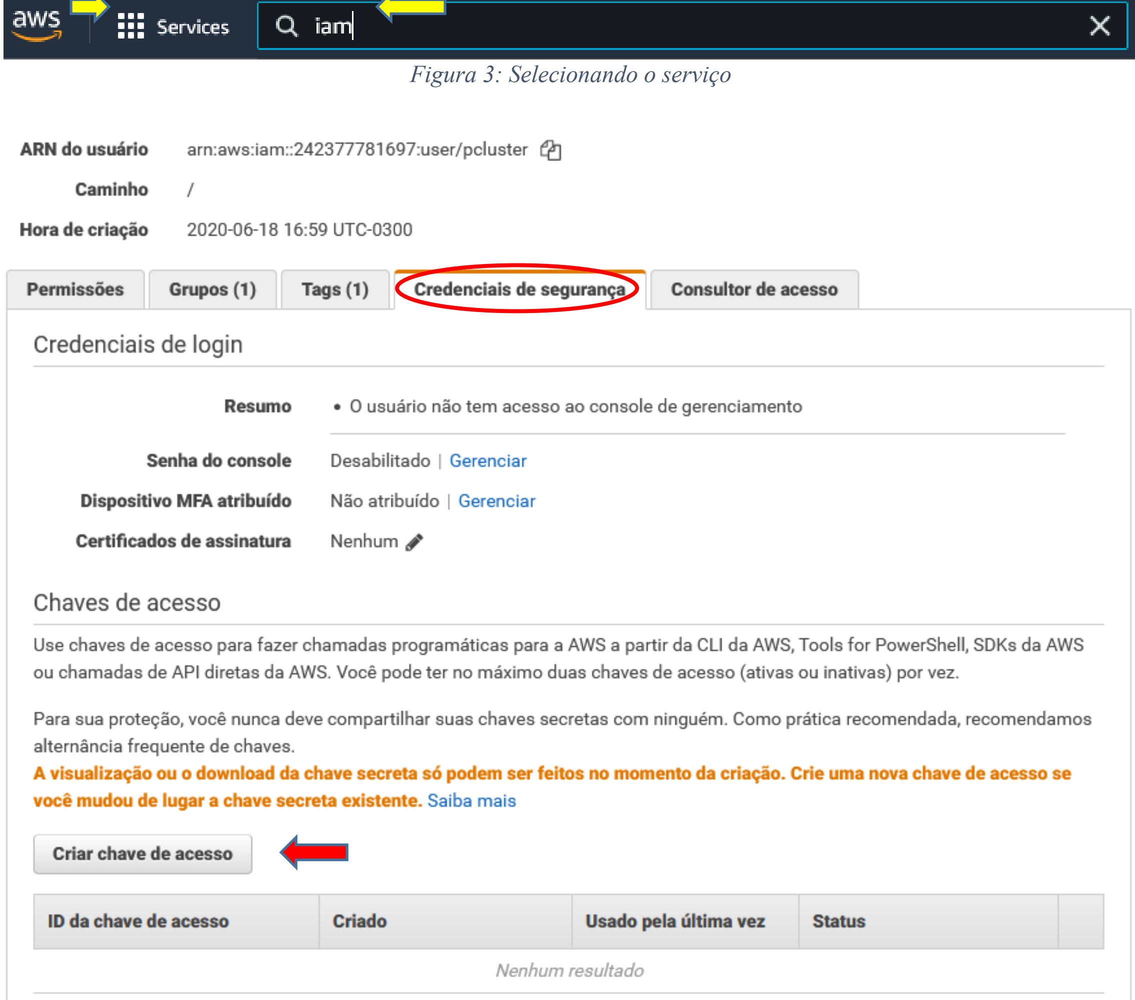


Figura 3: Selecionando o serviço

Figura 4: Serviço IAM para criação da chave

O acesso ao par de chave é somente neste momento. Pode-se visualizar ou fazer o download das chaves, como mostrado na Figura 5.

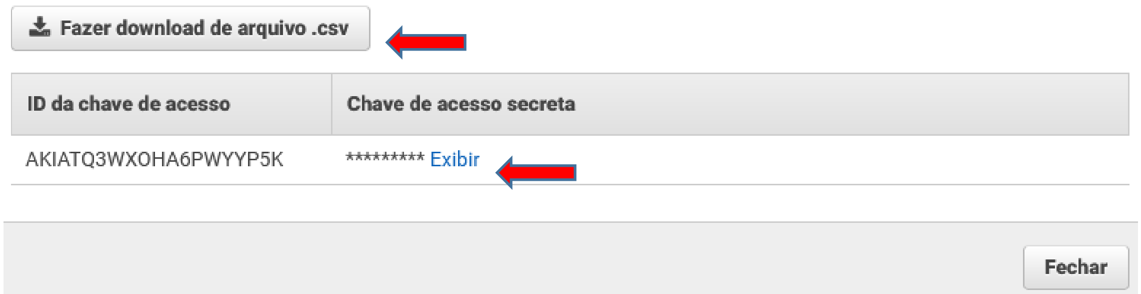


Figura 5: Acesso ao par de chaves

Dispondo do par de chaves pode-se configurar o AWS CLI através do comando.

```
(pcluster)[linux@ip ~]$ aws configure
AWS Access Key ID [None]: AKIATQ3WXOHA6PWYYP5K
AWS Secret Access Key [None]: yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
Default region name [None]: us-east-1
Default output format [None]:
```

Depois instale o *ParallelCluster* e verifique a versão instalada com os comandos abaixo.

```
(pcluster)[linux@ip ~]$ python3 -m pip install --upgrade \
                        aws-parallelcluster
(pcluster)[linux@ip ~]$ pcluster version
{
  "version": "3.2.0"
}
```

Instale o *Node Version Manager* e o *Node.js* dentro do ambiente virtual. Ele é necessário devido ao uso do *AWS Cloud Development Kit* (CDK) para a geração de *templates*.

```
(pcluster)[linux@ip ~]$ curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.38.0/install.sh | bash
(pcluster)[linux@ip ~]$ chmod ug+x ~/.nvm/nvm.sh
(pcluster)[linux@ip ~]$ source ~/.nvm/nvm.sh
(pcluster)[linux@ip ~]$ nvm install --lts
(pcluster)[linux@ip ~]$ node --version
v16.17.0
```

Antes de configurar o *ParalleCluster*, recomenda-se gerar um par de chaves SSH para o cluster, conforme ilustrado na Figura 6. Escolha um nome para a chave e selecione o formato desejado. O formato PEM normalmente é usado para o Linux, Mac OS X e alguns softwares do Windows, enquanto que o formato PPK costuma ser usado para o programa PuTTY no Windows. Crie as chaves usando o botão "Criar par de chaves" e o download delas será feito automaticamente.

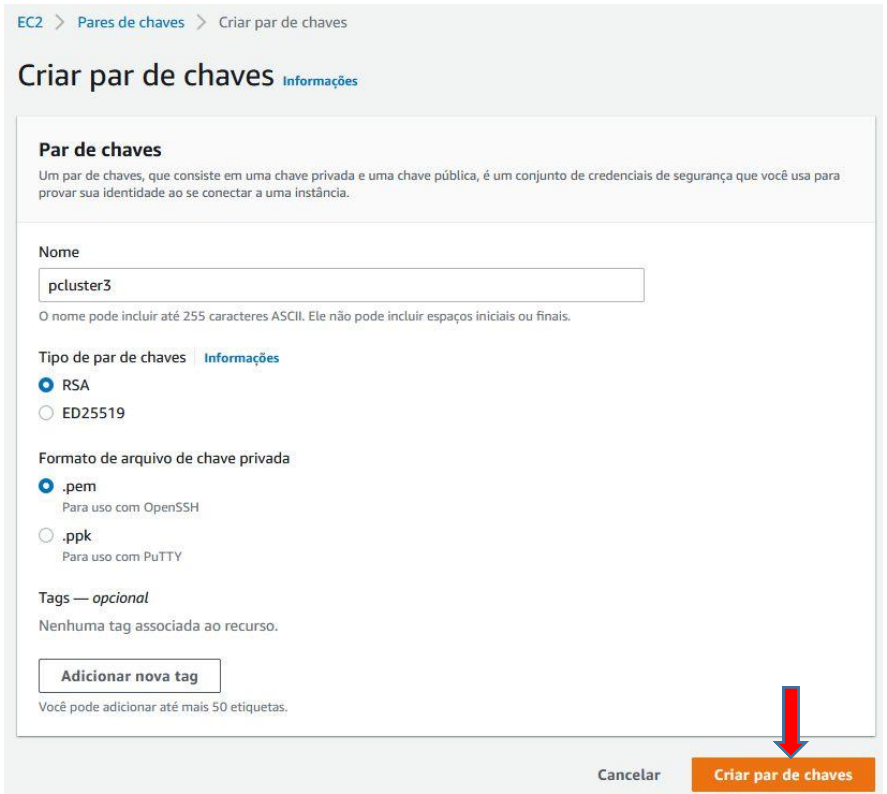


Figura 6: Gerar par de chaves do cluster para acesso via SSH

Então finalmente pode-se configurar o *ParallelCluster*. Ao executar o comando abaixo e gerar o arquivo de configuração no formato YAML serão feitas diversas perguntas sobre o cluster. Segue a seguir um exemplo das possíveis respostas.

Primeiro, escolha uma das regiões da AWS. A região *sa-east-1* corresponde a região de São Paulo. Esta região é a que possui a menor latência de comunicação, pois fica no Brasil. Outra região com baixa latência e a `us-east-1`.

```
(pcluster)[linux@ip ~]$ pcluster configure --config pcluster-config.yaml
INFO: Configuration file pcluster-config.yaml will be written.
Press CTRL-C to interrupt the procedure.
Allowed values for AWS Region ID:
1. ap-northeast-1
2. ap-northeast-2
3. ap-south-1
. . .
16. sa-east-1
. . .
AWS Region ID [us-east-1]:
```

Depois selecione o par de chaves gerado anteriormente na mesma região, vide Figura 6.

```
Allowed values for EC2 Key Pair Name:
1. pcluster3
EC2 Key Pair Name [pcluster3]: 1
```

Selecione o SLURM [10] como o *job scheduler*, muito comum em clusters de HPC. O AWS Batch não será abordado neste minicurso.

```
Allowed values for Scheduler:
1. slurm
2. awsbatch
Scheduler [slurm]: 1
```

Escolha qual o Sistema operacional do cluster. O *alinux2* (*Amazon Linux*) é uma distribuição Linux do AWS, originalmente baseada no Red Hat, simplificada otimizada para execução em instâncias do EC2. Ele fornece várias ferramentas para integrar-se perfeitamente aos serviços do EC2. Não é uma distribuição Linux independente e está disponível apenas para uso em um ambiente EC2.

```
Allowed values for Operating System:
1. alinux2
2. centos7
3. ubuntu1804
4. ubuntu2004
Operating System [alinux2]:
```

Escolha o tipo de instância, lembrando que o *headnode* pode ter uma instância diferente dos nós computacionais. Entretanto, recomenda-se que a rede tenha a mesma banda de comunicação entre eles, pois o *headnode* faz serviço de NFS com os nós computacionais e se as instâncias tiverem redes de velocidade diferentes pode impactar no desempenho. Neste exemplo, foi escolhido a *t3.micro* por, atualmente, ter melhor desempenho que a *t2.micro* e um custo menor.

```
Head node instance type [t2.micro]: t3.micro
```

Definir quantidade e as características de cada fila.

```
Number of queues [1]: 1
Name of queue 1 [queue1]: work
Number of compute resources for work [1]:
Compute instance type for compute resource 1 in work [t2.micro]: t3.micro
Maximum instance count [10]: 5
```

Depois que as etapas anteriores forem concluídas, decida se deseja usar uma VPC existente ou deixar que o AWS *ParallelCluster* crie uma VPC automaticamente, incluindo informações sobre a zona de disponibilidade. Pode-se usar o *headnode* e os nós de computação na mesma sub-rede pública ou somente o *headnode* em uma sub-rede pública com todos os nós em uma sub-rede privada. Em geral usa-se a segunda opção, porém se o nó de computação precisar acessar a Internet pode ser que a primeira opção seja mais fácil de configurar. Outra observação é que pode atingir o limite de VPCs permitidos em uma região. A cota padrão é cinco VPCs para uma região, mas caso seja necessário pode-se solicitar um aumento de cota no suporte da AWS.

```
Automate VPC creation? (y/n) [n]: y
Allowed values for Availability Zone:
1. us-east-1a
2. us-east-1b
3. us-east-1c
4. us-east-1d
5. us-east-1e
6. us-east-1f
Availability Zone [us-east-1a]:
Allowed values for Network Configuration:
1. Head node in a public subnet and compute fleet in a private subnet
```

```
2. Head node and compute fleet in the same public subnet
Network Configuration [Head node in a public subnet and compute fleet
in a private subnet]: 1
```

Finalmente o script gera toda a infraestrutura selecionada e gera o arquivo de configuração, neste caso o *pcluster-config.yaml*.

```
Beginning VPC creation. Please do not leave the terminal until the
creation is finalized
Creating CloudFormation stack...
Do not leave the terminal until the process has finished.
Stack   Name:   parallelclusternetworking-pubpriv-20220818134347   (id:
arn:aws:cloudformation:us-east-
1:242377781697:stack/parallelclusternetworking-pubpriv-
20220818134347/e814cd70-1efd-11ed-b0d6-0a60bc1e437f)
Status:       parallelclusternetworking-pubpriv-20220818134347       -
CREATE_COMPLETE
The stack has been created.
Configuration file written to pcluster-config.yaml
You can edit your configuration file or simply run 'pcluster create-
cluster  --cluster-configuration  pcluster-config.yaml  --cluster-name
cluster-name --region us-east-1' to create your cluster.
```

A seguir é mostrado o arquivo de configuração *pcluster-config.yaml.*

```
Region: us-east-1
Image:
  Os: alinux2
HeadNode:
  InstanceType: t3.micro
  Networking:
    SubnetId: subnet-0caf3554d75f42d7f
  Ssh:
    KeyName: pcluster3
Scheduling:
  Scheduler: slurm
  SlurmQueues:
  - Name: work
    ComputeResources:
    - Name: t3micro
      InstanceType: t3.micro
      MinCount: 0
      MaxCount: 5
    Networking:
      SubnetIds:
      - subnet-02b21b350c1169c8e
```

Feita a configuração básica pode-se fazer uma customização dos serviços que serão utilizados pelo cluster, alguns deles serão descritos no próximo item.

Recomenda-se que depois de qualquer alteração no arquivo de configuração, faça um teste na nova configuração. Para isto acrescente o parâmetro *"`--dryrun True`"* para não gerar o cluster e apenas verificar se a configuração possui algum erro. Se não houver

erro, a saída do comando de criar cluster será semelhante ao mostrado a seguir, onde o parâmetro "-n" especifica o nome do cluster e o "-c" o arquivo de configuração.

```
(pcluster)[linux@ip ~]$ pcluster create-cluster --dryrun True -n wscad
-c pcluster-config.yaml
{
  "message": "Request would have succeeded, but DryRun flag is set."
}
```

6.5.2. Customização na configuração e boas práticas

6.5.2.1. S3

Um dos serviços mais utilizados na AWS é o S3. Ele é o responsável por fazer o armazenamento de objetos. Para usar este serviço com o cluster recomenda-se primeiro criar os *buckets* e depois editar o arquivo de configuração YAML. O *bucket* S3 tem que ter um nome único dentro do S3 e não apenas na sua conta ou região. Neste minicurso, serão criados dois *buckets* chamados de *aws-pcluster3-dados* e *aws-pcluster3*, sendo que o primeiro será usado pelo cluster apenas como leitura e o segundo como leitura e escrita. Esse recurso é muito interessante quando se quer preservar e proteger os dados originais de algum problema durante a execução. Além de restringir o acesso aos demais *buckets* disponíveis na conta, pois apenas esses dois serão visualizados pelo cluster.

Selecione o serviço S3 de forma semelhante ao mostrado na Figura 3 e clique no botão "*Criar bucket*". Coloque o nome desejado e preste atenção para usar a mesma região em que o cluster foi ou será criado. As demais configurações podem ser deixadas como padrão. Depois vá ao final da página e clique no botão "*Criar bucket*". A Figura 7 ilustra parte do processo.

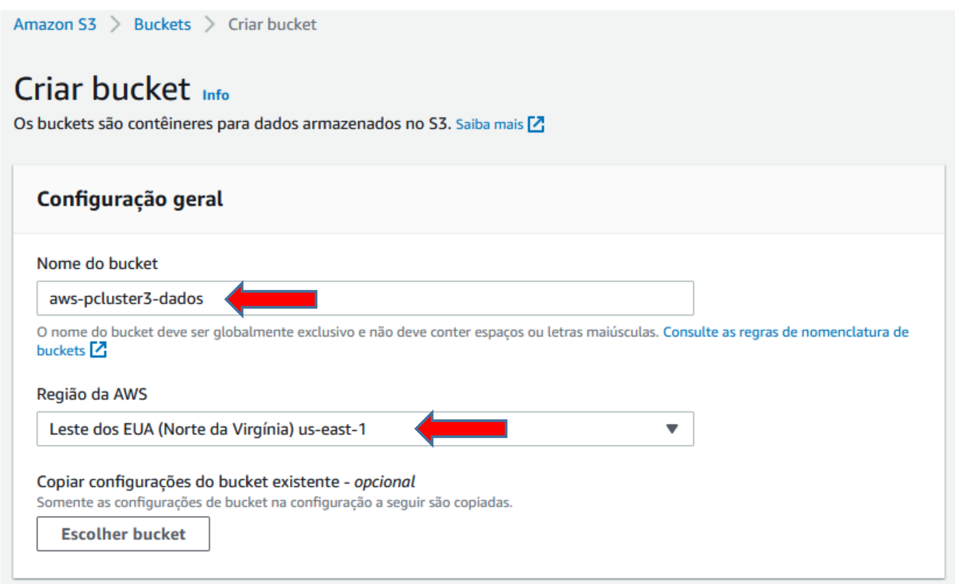


Figura 7: Criando um bucket S3

Adicione o trecho a seguir na configuração do YAML. Esta configuração dará acesso apenas de leitura ao `s3://aws-pcluster3-dados/dados_ro/` e acesso de leitura e escrita no `s3://aws-pcluster3/dados/`.

```
...
HeadNode:
...
Iam:
  S3Access:
    - BucketName: aws-pcluster3-dados
      KeyName: dados_ro/*
      EnableWriteAccess: false
    - BucketName: aws-pcluster3
      KeyName: dados/*
      EnableWriteAccess: true
...
```

Se for necessário ter acesso a todos os *buckets* basta mudar o parâmetro `BucketName` como mostrado a seguir.

```
...
Iam:
  S3Access:
    - BucketName: *
...
```

Dependendo de como vai ser usado o cluster pode-se incluir esta configuração também para os nós computacionais que estão definidos nas filas. Porém, cabe lembrar que o S3 é um sistema de armazenamento de objetos e não de blocos. Normalmente, o processamento de um programa deve usar o armazenamento em blocos e ao terminar e for necessário, pode-se copiar os arquivos para um sistema de armazenamento em objetos como o S3.

```
...
Scheduling:
  Scheduler: slurm
  SlurmQueues:
  - Name: work
  ...
    Iam:
      S3Access:
        - BucketName: aws-pcluster3
          EnableWriteAccess: true
        - BucketName: aws-pcluster3-dados
          KeyName: dados_ro/*
          EnableWriteAccess: false
  ...
```

6.5.2.2. Placement Group

Para configurar o *placement group* basta acrescentar o trecho abaixo no item relativo a rede da fila como mostrado abaixo.

```
…
Scheduling:
  Scheduler: slurm
  SlurmQueues:
  - Name: work
  ...
    Networking:
      SubnetIds:
      - subnet-02b21b350c1169c8e
      PlacementGroup:
        Enabled: true
  ...
```

6.5.2.3. Minimização de custos

A fim de minimizar o custo financeiro pode-se utilizar instâncias Spots no cluster. Em geral, as instâncias Spots possuem um preço menor que a instância sob demanda e às vezes o desconto pode chegar até 90%. Na configuração do cluster pode-se definir o preço máximo que será pago por cada instância Spot do EC2 antes que elas sejam executadas. O valor padrão é o preço sob demanda. Cabe aqui lembrar que se a AWS receber uma oferta maior pela sua instância que está sendo usada para processar o seu *job*, ele pode a qualquer momento desligar ela, mesmo que o seu *job* ainda não tenha terminado. Apesar deste inconveniente, este tipo de instância é muito utilizado em função do retorno financeiro e pela pouca frequência de interrupção, como pode ser vista pelo *Spot Instance Advisor* [*11*] que está mostrado na Figura 8. Nesta figura pode-se ver uma economia alta, em média acima de 70%, e uma frequência de interrupção menor que 5%.

Para exemplificar com números, suponha que o preço da instância *t3.micro* sob demanda esteja $0,0104. Pode-se definir que o preço máximo a ser pago por cada instância seja de $0,008. Neste caso, a configuração do arquivo YAML deve incluir também a linha com o preço máximo além de definir o tipo como SPOT.

```
...
Scheduling:
  Scheduler: slurm
  SlurmQueues:
  - Name: work
    ComputeResources:
    - Name: t3micro
      InstanceType: t3.micro
      MinCount: 0
      MaxCount: 5
      SpotPrice: 0.008
    CapacityType: SPOT
    ...
```

Tipo de instância	vCPU	Memória GiB	Economia em relação aos preços sob demanda*	Frequência da interrupção
c5ad.4xlarge	16	32	78%	<5%
c6g.xlarge	4	8	71%	<5%
c6g.2xlarge	8	16	71%	<5%
c5.large	2	4	78%	<5%
t3a.nano	2	0.5	66%	<5%
r5b.2xlarge	8	64	86%	<5%
m6g.12xlarge	48	192	73%	<5%
r6id.2xlarge	8	64	85%	<5%
r5.metal	96	768	83%	<5%
m5d.16xlarge	64	256	82%	<5%

Exibir todos os tipos de instância 473

Figura 8: Spot Instance Advisor na região de Ohio

6.5.2.4. Hyperthreading

Normalmente recomenda-se que o *hyperthreading* esteja desativado para o processamento de aplicações de HPC por questões de desempenho. Este parâmetro vem ativado por padrão nas instâncias EC2. Costuma-se desabilitar, caso seja necessário, apenas nos nós computacionais. Entretanto, nem todos os tipos de instância é possível desabilitar o *hyperthreading*. A configuração do parâmetro `DisableSimultaneousMultithreading` deve ser feito na fila correspondente. A mudança deste parâmetro não gera nenhum custo adicional.

```
...
Scheduling:
  Scheduler: slurm
  SlurmQueues:
  - Name: work
    ComputeResources:
    - Name: t3micro
      InstanceType: t3.micro
      MinCount: 0
      MaxCount: 5
      DisableSimultaneousMultithreading: true
    ...
```

6.5.2.5. AMI customizada

As aplicações de alto desempenho, normalmente, requerem algumas características diferentes das aplicações tradicionais. E para tanto, as vezes é necessário criar um ambiente bem específico que não são inclusas nas imagens tradicionais da AWS. Para endereçar a este tipo de problema é possível criar uma AMI (*Amazon Machine Image*) customizada.

A forma mais prática de criar uma imagem seria levantar uma instância com o sistema operacional desejado (*Amazon Linux, Rocky Linux, Ubuntu*, etc) e fazer todas as instalações e configurações necessárias para o ambiente da aplicação. Outra forma seria, se já tivesse algum script otimizado que fizesse todas as instalações e configurações. Este script poderia ser executado na hora de iniciar a instancia e depois gerar uma nova AMI, por exemplo, `ami-12345678`.

Estando de posse desta AMI pode-se usa-la como uma imagem, normalmente, dos nós computacionais do cluster conforme mostrado no trecho da configuração abaixo.

```
...
Scheduling:
  Scheduler: slurm
  SlurmQueues:
    - Name: work
      Image:
        CustomAmi: ami-12345678
      ...
```

6.5.2.6. Tempo de espera

A alocação do nó computacional se dá de acordo com a demanda e a definição do tamanho do cluster. Ao submeter um *job, uma ou mais* instâncias são inicializadas para o processamento, porém este processo demora poucos minutos. Entretanto, pode-se definir o tempo máximo que um nó computacional fique aguardando um novo *job,* evitando assim uma nova espera do provisionamento. Se esse tempo for ultrapassado o nó em questão será terminado e removido do cluster, desta forma parando de gerar custo. Por padrão esse tempo é de 10 minutos. O parâmetro que permite esse ajuste é o `ScaledownIdletime`. No trecho de configuração a seguir esse limite foi modificado para 5 minutos.

```
...
Scheduling:
  Scheduler: slurm
  SlurmSettings:
    ScaledownIdletime: 5
        ...
```

6.5.3. Usando o *ParallelCluster*

6.5.3.1. Criando o cluster

Depois da visão geral das principais características do *ParallelCluster* pode-se fazer a implementação do cluster. Neste exemplo será usado a seguinte configuração do YAML. Esta configuração exemplifica a alguns dos parâmetros descritos anteriormente, tais como: acesso ao S3 tanto de leitura e escrita como apenas de leitura, configuração do *idle time*, uso de instâncias spots, etc.

```
Region: us-east-1
Image:
  Os: alinux2
HeadNode:
  InstanceType: t3.micro
  Networking:
    SubnetId: subnet-00af4837d78f42d7f
  Ssh:
    KeyName: pcluster3
  Iam:
    S3Access:
      - BucketName: aws-pcluster3
        EnableWriteAccess: true
      - BucketName: aws-pcluster3-dados
        EnableWriteAccess: false
Scheduling:
  Scheduler: slurm
  SlurmSettings:
    ScaledownIdletime: 5
  SlurmQueues:
  - Name: work
    ComputeResources:
    - Name: t3micro
      InstanceType: t3.micro
      MinCount: 0
      MaxCount: 5
    CapacityType: SPOT
    Networking:
      SubnetIds:
      - subnet-02b23b87c1165c8e
    Iam:
      S3Access:
        - BucketName: aws-pcluster3
          EnableWriteAccess: true
        - BucketName: aws-pcluster3-dados
          EnableWriteAccess: false
```

Para criar um cluster utilize o comando a seguir e aguarde alguns minutos até que o serviço *CloudFormation* faça todas as configurações de rede, segurança e inicialize o *headnode*. Veja o status inicial do cluster `CREATE_IN_PROGRESS`.

```
(pcluster)[ec2-user@ip ~]$ pcluster create-cluster -n wscad \
                           -c pcluster-config.yaml
{
  "cluster": {
    "clusterName": "wscad",
    "cloudformationStackStatus": "CREATE_IN_PROGRESS",
    "cloudformationStackArn": "arn:aws:cloudformation:us-east-1:...",
    "region": "us-east-1",
    "version": "3.2.0",
    "clusterStatus": "CREATE_IN_PROGRESS",
    "scheduler": {
```

```
      "type": "slurm"
    }
  }
}
```

Pode-se acompanhar a implementação do cluster com o comando abaixo até dar por concluído todo o processo, com o status `CREATE_COMPLETE`.

```
(pcluster)[ec2-user@ip ~]$ pcluster describe-cluster -n wscad
{
  "creationTime": "2022-08-26T19:09:50.252Z",
  "headNode": {...},
  "version": "3.2.0",
  "clusterConfiguration": {...},
  "tags": [ {...}, {...} ],
  "cloudFormationStackStatus": "CREATE_COMPLETE",
  "clusterName": "wscad",
  "computeFleetStatus": "RUNNING",
  "cloudformationStackArn": "arn:aws:cloudformation:...",
  "lastUpdatedTime": "2022-08-26T19:09:50.252Z",
  "region": "us-east-1",
  "clusterStatus": "CREATE_COMPLETE",
  "scheduler": {
    "type": "slurm"
  }
}
```

Também é possível listar todos os clusters com o comando a seguir.

```
 (pcluster)[ec2-user@ip ~]$ pcluster list-clusters
{
  "clusters": [
    {
      "clusterName": "wscad",
      "cloudformationStackStatus": "CREATE_COMPLETE",
      "cloudformationStackArn": "arn:aws:cloudformation:us-east-1:...",
      "region": "us-east-1",
      "version": "3.2.0",
      "clusterStatus": "CREATE_COMPLETE",
      "scheduler": { "type": "slurm" }
    }
  ]
}
```

Caso haja interesse em examinar os eventos do *CloudFormation* use o comando a seguir ou consulte diretamente o serviço através da console da AWS.

```
(pcluster)[ec2-user@ip ~]$ pcluster get-cluster-stack-events -n wscad
```

Inicialmente só havia a instancia chamada *Bastion*, onde foi instalado o ambiente virtual com o *ParallelCluster*. Ao executar o comando para criar o cluster foi inicializado o *headnode* do cluster sem nenhum nó computacional, conforme mostra a Figura 9.

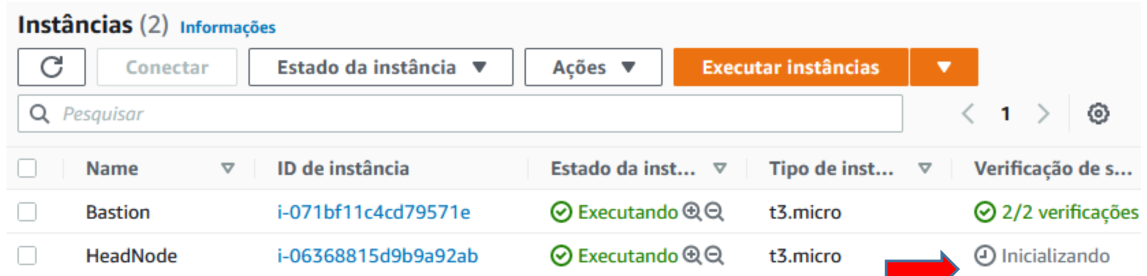


Figura 9: Instâncias na AWS

6.5.3.2. Acessando o cluster

Para acessar o cluster basta usar a chave gerada anteriormente, vide Figura 6.

```
(pcluster)[ec2-user@ip ~]$ pcluster ssh -n wscad -i ~/.ssh/pcluster3.pem
Last login: Fri Aug 26 19:14:53 2022

       __|  __|_  )
       _|  (     /   Amazon Linux 2 AMI
      ___|\___|___|

https://aws.amazon.com/amazon-linux-2/

[ec2-user@ip-10-0-0-201 ~]$
```

Coletando informações da fila, neste acaso apenas uma fila com cinco nós computacionais.

```
[ec2-user@ip-10-0-0-201 ~]$ sinfo
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
work*        up   infinite      5  idle~ work-dy-t3micro-[1-5]
```

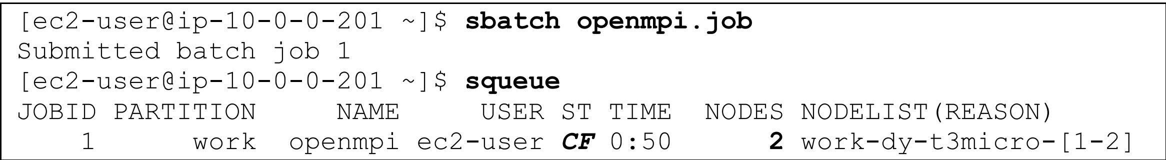
Consulte as áreas compartilhadas pelo *headnode* para os nós computacionais através do serviço NFS.

```
[ec2-user@ip-10-0-0-201 ~]$ sudo exportfs -v
/home                        10.0.0.0/16(rw,sync,wdelay,hide,
    no_subtree_check,sec=sys,secure,no_root_squash,no_all_squash)
/opt/parallelcluster/shared  10.0.0.0/16(rw,sync,wdelay,hide,
    no_subtree_check,sec=sys,secure,no_root_squash,no_all_squash)
/opt/intel                   10.0.0.0/16(rw,sync,wdelay,hide,
    no_subtree_check,sec=sys,secure,no_root_squash,no_all_squash)
/opt/slurm                   10.0.0.0/16(rw,sync,wdelay,hide,
    no_subtree_check,sec=sys,secure,no_root_squash,no_all_squash)
```

A versão do MPI que já vem instalado por padrão no *ParallelCluster* é o OpenMPI [12], vide comando abaixo, e ele tem suporte para a interface EFA. Entretanto para usar o EFA é necessário instalar o software EFA, vide o *ParallelCuster User Guide* [13]. Também é possível usar o Intel MPI e o MVAPICH2-X-AWS [14]. Estas instalações e configurações não serão abordados neste minicurso.

```
[ec2-user@ip-10-0-0-189 ~]$ which mpirun
/opt/amazon/openmpi/bin/mpirun
```

Submetendo um *job* MPI que usará dois nós computacionais. Depois verifica-se o estado da fila. Repare que o status (ST) está como CF, isto quer dizer que o nó ainda está sendo configurado.



```
[ec2-user@ip-10-0-0-201 ~]$ sbatch openmpi.job
Submitted batch job 1
[ec2-user@ip-10-0-0-201 ~]$ squeue
JOBID PARTITION     NAME     USER ST TIME  NODES NODELIST(REASON)
    1      work  openmpi ec2-user CF 0:50      2 work-dy-t3micro-[1-2]
```

Repare os nós de processamento sendo alocados na

Instâncias (4) Informações

Conectar | Estado da instância ▼ | Ações ▼ | Executar instâncias ▼

Pesquisar

Name	ID de instância	Estado da inst...	Tipo de inst...	Verificação de s...
Bastion	i-071bf11c4cd79571e	Executando	t3.micro	2/2 verificações
HeadNode	i-06368815d9b9a92ab	Executando	t3.micro	2/2 verificações
Compute	i-0bf9aaec75422640c	Executando	t3.micro	Inicializando
Compute	i-0859243a5693dd1d9	Executando	t3.micro	Inicializando

Figura 10.

Instâncias (4) Informações

Conectar | Estado da instância ▼ | Ações ▼ | Executar instâncias ▼

Pesquisar

Name	ID de instância	Estado da inst...	Tipo de inst...	Verificação de s...
Bastion	i-071bf11c4cd79571e	Executando	t3.micro	2/2 verificações
HeadNode	i-06368815d9b9a92ab	Executando	t3.micro	2/2 verificações
Compute	i-0bf9aaec75422640c	Executando	t3.micro	Inicializando
Compute	i-0859243a5693dd1d9	Executando	t3.micro	Inicializando

Figura 10: Alocação dos nós de computação

Submetendo um *job* serial e observando que após uns minutos o primeiro *job* já sendo executado enquanto que o *job* serial ainda está sendo alocado.

```
[ec2-user@ip-10-0-0-201 ~]$ sbatch test.job
Submitted batch job 2
[ec2-user@ip-10-0-0-201 ~]$ squeue
JOBID PARTITION     NAME     USER  ST TIME  NODES NODELIST(REASON)
    1      work openmpi. ec2-user   R 4:33      2 work-dy-t3micro-[1-2]
    2      work teste.sh ec2-user  CF 0:12      1 work-dy-t3micro-3
```

Observando novamente a fila, pode-se identificar o nome de DNS dos nós alocados e o mesmo responde pelo seu nome de DNS conforme ilustrado a seguir.

```
[ec2-user@ip-10-0-0-201 ~]$ sinfo
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
work*        up   infinite      2  idle~ work-dy-t3micro-[4-5]
work*        up   infinite      1    mix work-dy-t3micro-3
work*        up   infinite      2  alloc work-dy-t3micro-[1-2]
```

```
[ec2-user@ip-10-0-0-201 ~]$ ping work-dy-t3micro-1
PING work-dy-t3micro-1.wscad.pcluster (10.0.24.168) 56(84) bytes of
data.
64 bytes from ip-10-0-24-168.ec2.internal (10.0.24.168): icmp_seq=1
ttl=255 time=0.201 ms
64 bytes from ip-10-0-24-168.ec2.internal (10.0.24.168): icmp_seq=2
ttl=255 time=0.203 ms
64 bytes from ip-10-0-24-168.ec2.internal (10.0.24.168): icmp_seq=3
ttl=255 time=0.192 ms
^C
--- work-dy-t3micro-1.wscad.pcluster ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 4054ms
rtt min/avg/max/mdev = 0.192/0.200/0.207/0.018 ms
```

Depois que os *jobs* terminaram e se passaram os 5 minutos definidos no arquivo YAML, os nós de processamento foram encerrados automaticamente, como mostrado na Figura 11.

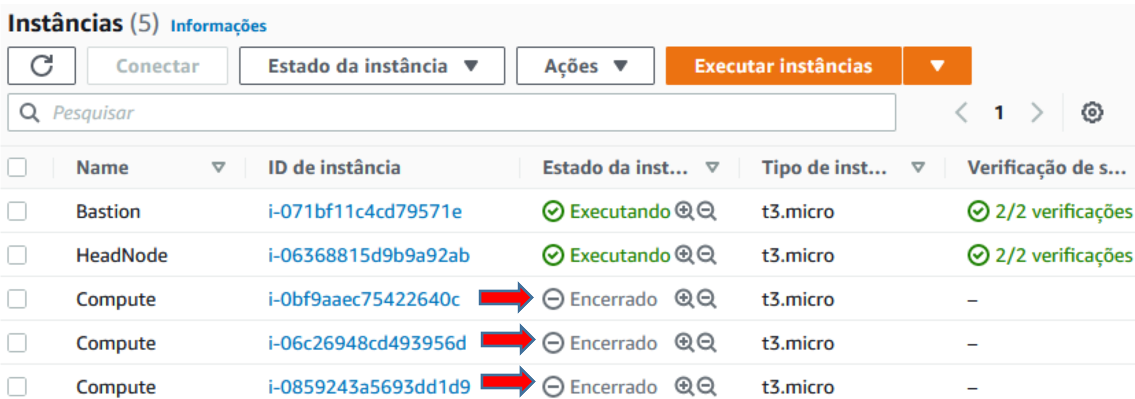


Figura 11: Nós de processamento encerrados

6.5.3.3. S3

Verificando a política do S3 definida no arquivo YAML, que tem permissão de leitura e escrita no *bucket* `aws-pcluster` e só leitura no *bucket* `aws-pcluster-dados`, se está funcionando.

Primeiro tentando listar todos os *buckets* através do AWS CLI e não conseguindo porque só tem acesso aos dois *buckets* definidos anteriormente.

```
[ec2-user@ip-10-0-0-189 ~]$ aws s3 ls
An error occurred (AccessDenied) when calling the ListBuckets operation:
Access Denied

[ec2-user@ip-10-0-0-189 ~]$ aws s3 ls s3://aws-pcluster3/
2022-08-26 19:41:47      28820 pcluster-manager.yaml

[ec2-user@ip-10-0-0-189 ~]$ aws s3 ls s3://aws-pcluster3-dados
```

```
                           PRE dados_ro/
```

Copiando a saída do primeiro *job* para um novo diretório chamado output. Se o diretório não existir o S3 criará ele automaticamente, conforme mostrado a seguir.

```
[ec2-user@ip-10-0-0-189 ~]$ aws s3 cp slurm-1.out \
                            s3://aws-pcluster3/output/
upload: ./slurm-1.out to s3://aws-pcluster3/output/slurm-1.out

[ec2-user@ip-10-0-0-189 ~]$ aws s3 ls s3://aws-pcluster3/
                           PRE output/
2022-08-26 19:41:47      28820 pcluster-manager.yaml

[ec2-user@ip-10-0-0-189 ~]$ aws s3 ls s3://aws-pcluster3/output/
2022-08-28 18:42:04      81723 slurm-1.out
```

Tentando fazer o mesmo no *bucket* que tem permissão apenas de leitura.

```
[ec2-user@ip-10-0-0-189 ~]$ aws s3 cp slurm-1.out \
                            s3://aws-pcluster3-dados/output/
upload failed: ./slurm-1.out to s3://aws-pcluster3-dados/output/slurm-
1.out An error occurred (AccessDenied) when calling the PutObject
operation: Access Denied
```

É possível remover um diretório de forma recursiva com o comando abaixo.

```
[ec2-user@ip-10-0-0-189 ~]$ aws s3 rm s3://aws-pcluster3/ --recursive \
                            --exclude "*" --include "output/*"
delete: s3://aws-pcluster3/output/slurm-1.out
```

Porém é importante prestar atenção na sequência `--exclude` e depois `--include`, se esta ordem for invertida não irá funcionar. Por exemplo, no *bucket* abaixo ainda tem uma cópia do arquivo de configuração YAML. Se colocarmos primeiro o `--include` e depois `--exclude` veja que não funciona. Depois colocando na ordem correta o arquivo foi removido.

```
[ec2-user@ip-10-0-0-189 ~]$ aws s3 ls s3://aws-pcluster3
2022-08-26 19:41:47      28820 pcluster-manager.yaml

[ec2-user@ip-10-0-0-189 ~]$ aws s3 rm s3://aws-pcluster3/ --recursive \
                            --include "*.yaml" --exclude "*"
[ec2-user@ip-10-0-0-189 ~]$ aws s3 ls s3://aws-pcluster3
2022-08-26 19:41:47      28820 pcluster-manager.yaml
[ec2-user@ip-10-0-0-189 ~]$ aws s3 rm s3://aws-pcluster3/ --recursive \
                            --exclude "*" --include "*.yaml"
delete: s3://aws-pcluster3/pcluster-manager.yaml
```

6.5.3.4. Removendo o cluster

Depois de fazer todo o processamento necessário deve-se remover o cluster para não ficar gerando custo. A ideia da nuvem é que você paga pelo que consome, logo se terminou as analises, remova o cluster. O comando para remover o cluster está mostrado a seguir.

```
(pcluster)[ec2-user@ip ~]$ pcluster delete-cluster -n wscad
{
  "cluster": {
    "clusterName": "wscad",
    "cloudformationStackStatus": "DELETE_IN_PROGRESS",
    "cloudformationStackArn": "arn:aws:cloudformation:us-east-1:...",
    "region": "us-east-1",
    "version": "3.2.0",
    "clusterStatus": "DELETE_IN_PROGRESS",
    "scheduler": {
      "type": "slurm"
    }
  }
}
```

Após algum tempo pode-se confirmar que o cluster foi removido pelo comando abaixo.

```
(pcluster)[ec2-user@ip ~]$ pcluster list-clusters
{
  "clusters": []
}
```

6.6. Considerações Finais

Este minicurso mostrou as principais características do *ParallelCluster* para facilitar a implementação de um cluster na nuvem pública da AWS. Entretanto, não foi possível cobrir todos os recursos oferecidos pela ferramenta, nem entrar nos detalhes das configurações. A seguir uma lista de recursos que podem ser interessantes em se aprofundar: interface EFA incluindo o uso com o Intel MPI e o MVAPICH-X, EFS, *FSx for Lustre*, verificar as opções dos comandos, criação de AMIs personalizadas, múltiplas filas, múltiplos usuários, políticas de segurança da AWS, ajustes no SLURM para contabilidade e muitos outros.

Neste minicurso foi feito algumas ações via a console da AWS e outros por linha de comando, porém com o AWS CLI tudo poderia ser feito por linha de comando. O que dependendo da aplicação pode facilitar a automação de processos.

Existe também um *front-end* via web para o *AWS ParallelCluster*, conhecido como *Parallel Cluster Manager* [15]. Basicamente é UI (*User Interface*) que ajuda a montar o arquivo de configuração YAML e gerenciar o cluster.

6.7. Referências

[1] Star Cluster. Disponível em: http://star.mit.edu/cluster/

[2] AWS services used by AWS ParallelCluster. Disponível em: https://docs.aws.amazon.com/parallelcluster/latest/ug/aws-services.html. Acesso em: 15/08/2022.

[3] Lustre. Página inicial. Disponível em: https://www.lustre.org. Acesso em: 15/08/2022.

[4] Terraform. Página inicial. Disponível em: https://www.terraform.io/. Acesso em: 15/08/2022.

[5] Pulumi. Página inicial. Disponível em: https://www.pulumi.com/. Acesso em: 15/08/2022.

[6] Ansible Red Hat. Página inicial. Disponível em: https://www.ansible.com/. Acesso em: 15/08/2022.

[7] Chef. Página inicial. Disponível em: https://www.chef.io/. Acesso em: 15/08/2022.

[8] Puppet. Página inicial. Disponível em: https://puppet.com/. Acesso em: 15/08/2022.

[9] Crossplane. Página inicial. Disponível em: https://crossplane.io/. Acesso em: 15/08/2022.

[10] SLURM. Disponível em: https://slurm.schedmd.com/. Acesso em: 15/08/2022.

[11] Spot Instance Advisor. Disponível em: https://aws.amazon.com/pt/ec2/spot/instance-advisor/. Acesso em: 15/08/2022.

[12] Open MPI: Open Source High Performance Computing. Disponível em: https://www.open-mpi.org/. Acesso em: 15/08/2022.

[13] Amazon Web Service, “AWS ParallelCluster: AWS ParallelCluster User Guide”, July 6, 2022.

[14] MVAPICH: MPI over InfiniBand, Omni-Path, Ethernet/iWARP, and RoCE. Disponível em: https://mvapich.cse.ohio-state.edu/downloads/. Acesso em: 15/08/2022.

[15] ParallelCluster Manager - Make HPC Easy. Disponível em: https://github.com/aws-samples/pcluster-manager. Acesso em: 15/08/2022