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 estiveram 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

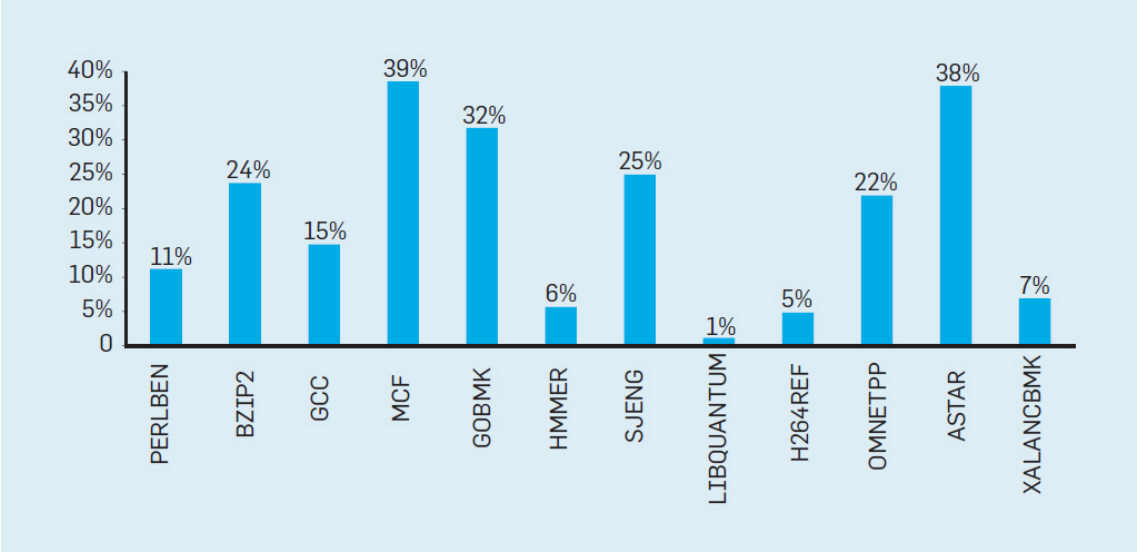


Figura 1.1: Percentual de instruções desperdiçadas em um processador Intel Core i7 para uma variedade de benchmarks de inteiros SPEC (Hennessy & Patterson, 2019).

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*.

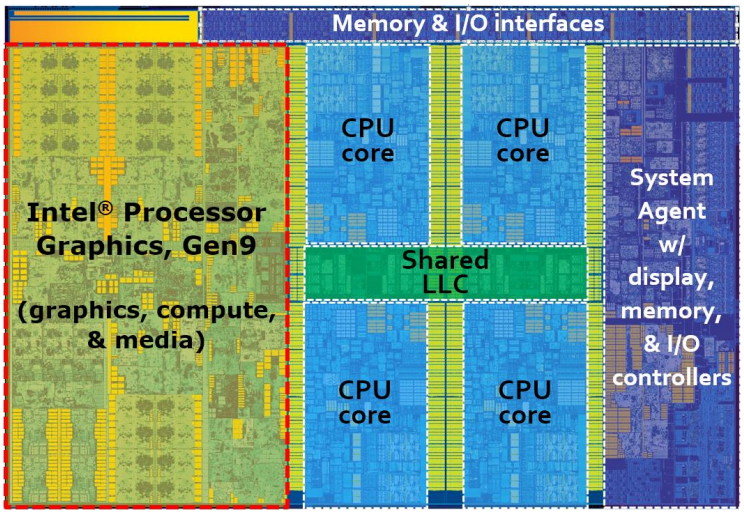


Figura 1.2: Processador Intel® Core™ i7 6700K de 4 núcleos com GPU Gen9 GT2 integrada. Nesta configuração a GPU tem 24 *slices* agrupados em 8 *subslices*, cada um com 8 unidades de execução (EU).

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.

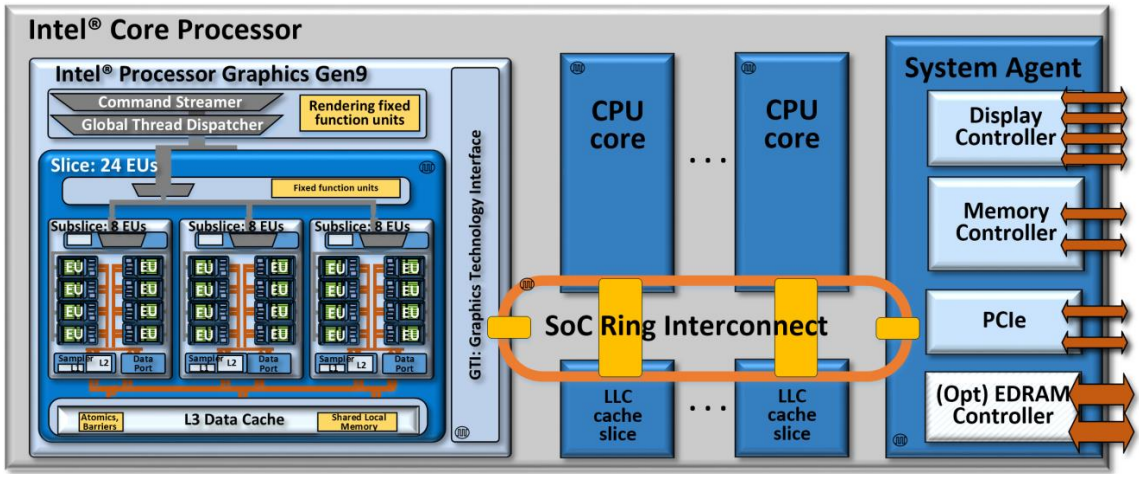


Figura 1.3: Interconexão em anel com múltiplos domínios de relógio.

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-

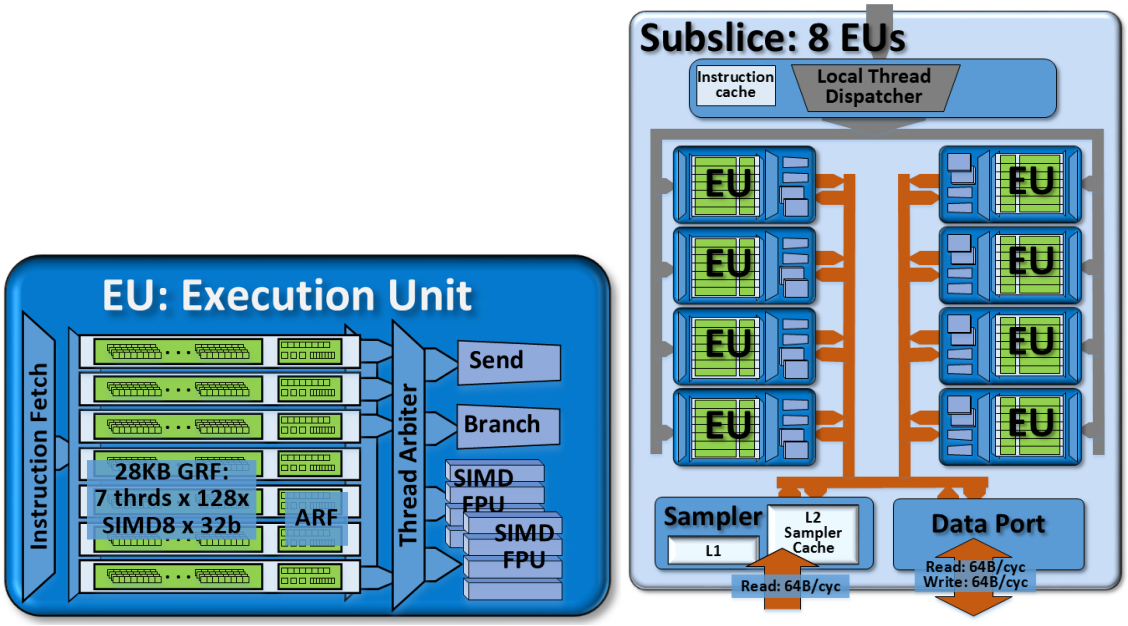


Figura 1.4: Um unidade de execução (EU) e seu agrupamento em um *subslice*.

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)

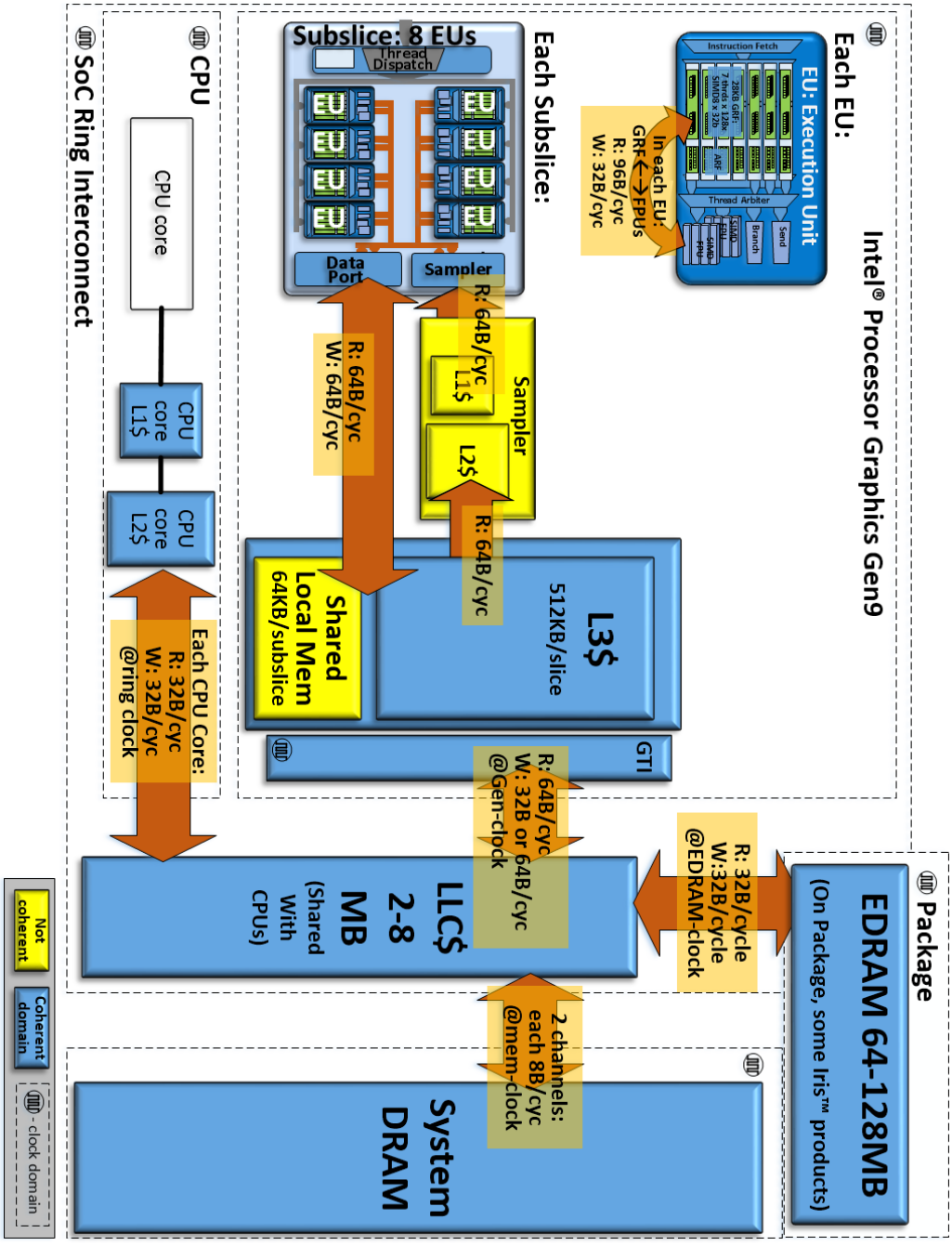


Figura 1.5: Hierarquia de memória com as larguras de banda para acesso.

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).

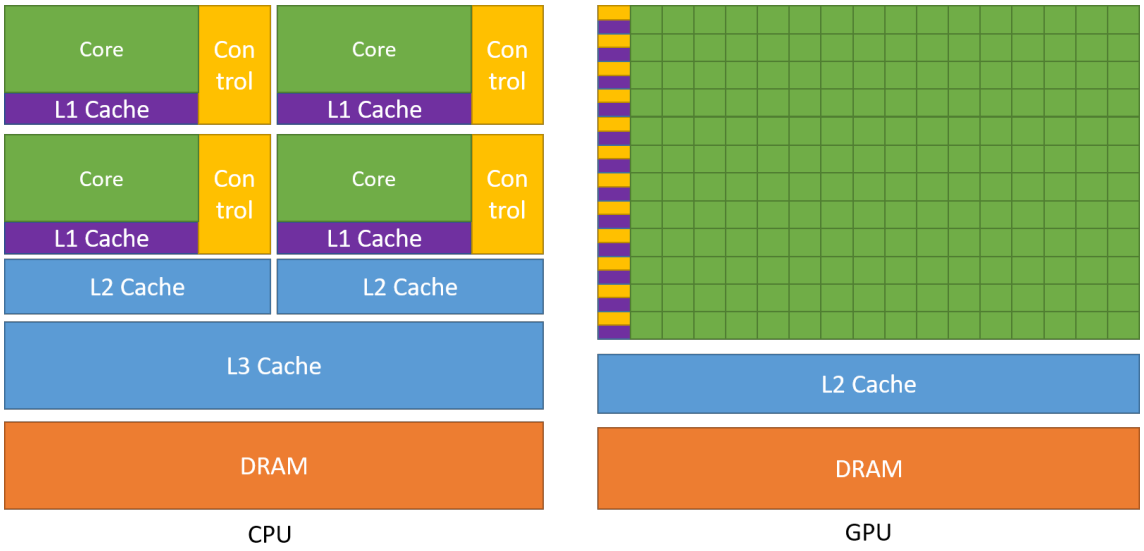


Figura 1.6: A GPU dedica mais transistores ao processamento de dados.

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*.

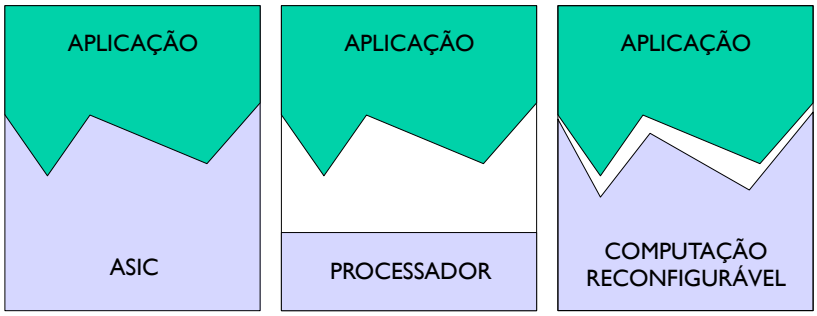


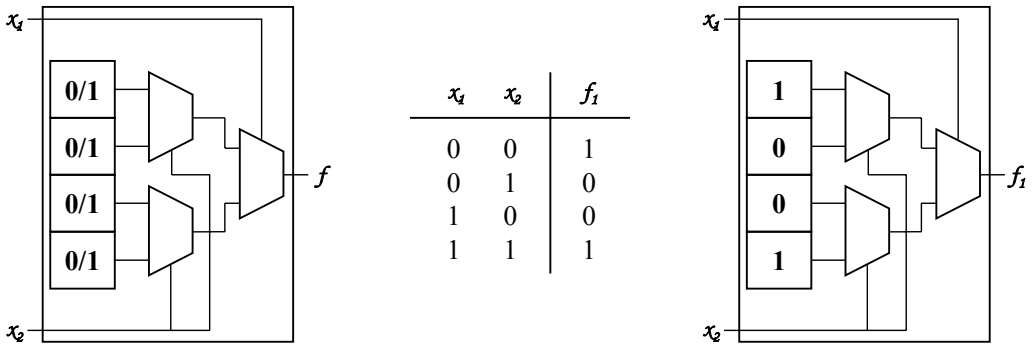
Figura 1.7: Computação reconfigurável comparada às soluções de *hardware* dedicado (ASIC) e *software* executando em processador de propósito geral (Menotti, 2010)

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*



(a) (b) (c)

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 um 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.

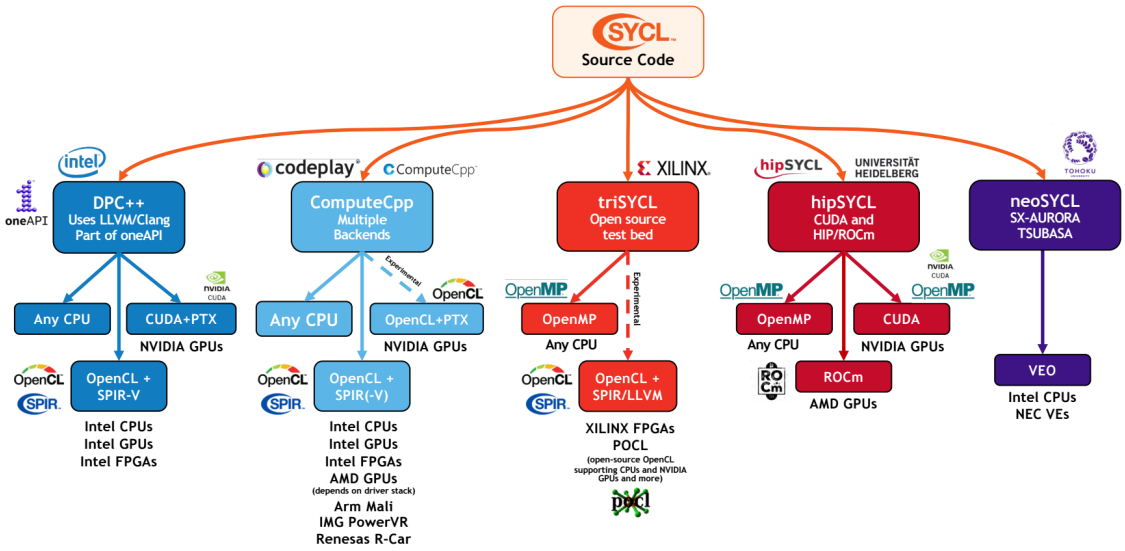


Figura 1.9: Algumas das implementações SYCL disponíveis atualmente

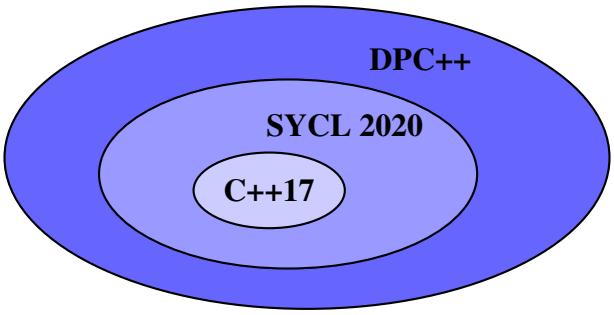


Figura 1.10: Relação de DPC++ com SYCL e C++

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.

```
#include <iostream>

template <class T>
T GetMax (T a, T b) {
    T result;
    result = (a > b) ? a : b;
    return (result);
}

int main () {
    int i = 7, j = 5, k;
    float l = 1.0, m = 5.5, n;
    k = GetMax<int>(i, j);
    n = GetMax<float>(l, m);
    std::cout << k << std::endl;
    std::cout << n << std::endl;
    return 0;
}
```

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.

```
// Cria um buffer de 2 dimensões com 8x8 inteiros
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.

```
1 std::vector<int> v {2, 7, 42};
2 for (std::vector<int>::iterator e = v.begin(); e != v.end();
   ↪  ++e)
3     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.

```
1 queue q;
2 device my_device = q.get_device();
3 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.

```
1  //==============================================================
2  // Copyright © 2020 Intel Corporation
3  //
4  // SPDX-License-Identifier: MIT
5  // =============================================================
6  #include <CL/sycl.hpp>
7
8  using namespace cl::sycl;
9
10 int main() {
11   //# Create a device queue with device selector
12
13   gpu_selector selector;
14   //cpu_selector selector;
15   //default_selector selector;
16   //host_selector selector;
17
18   queue q(selector);
19
20   //# Print the device name
21   std::cout << "Device: " <<
     ↪  q.get_device().get_info<info::device::name>() << std::endl;
22
23   return 0;
24 }
```

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.

```
1 q.submit([&](handler& h) {
2     //COMMAND GROUP CODE
3 });
```

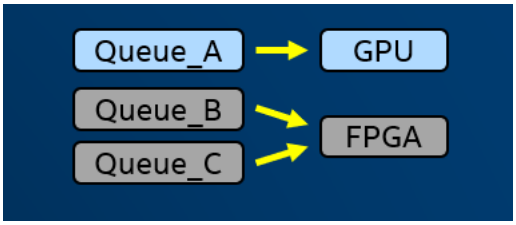


Figura 1.14: Filas associadas a dispositivos

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

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

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.

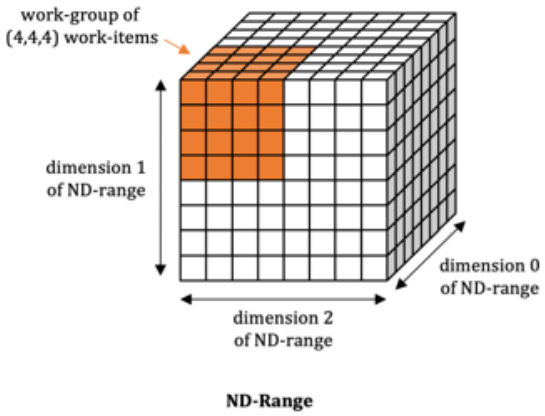


Figura 1.15: *Kernel* NDRange (N-dimensões) com *work-items* em destaque

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 acelerador. 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.

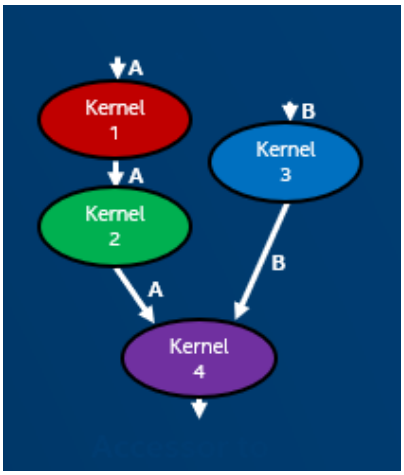


Figura 1.17: Grafo de dependência de dados entre *kernels*

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

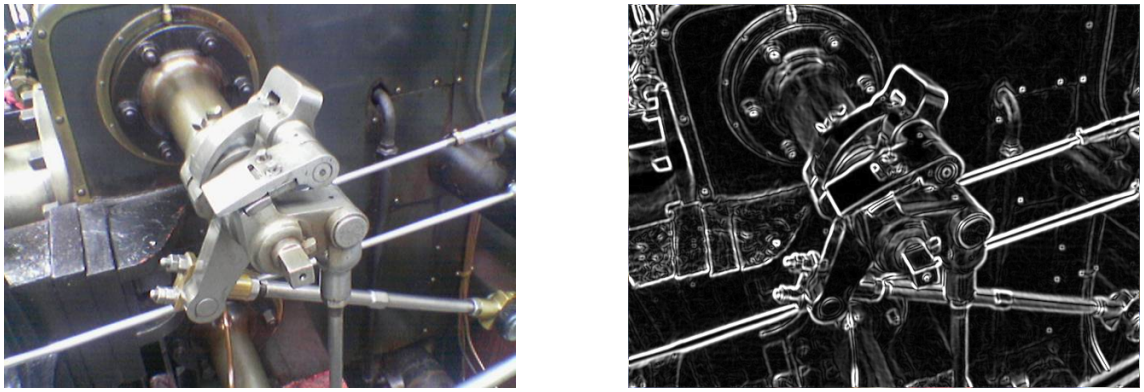


Figura 1.19: Exemplo da aplicação do filtro Sobel

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

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

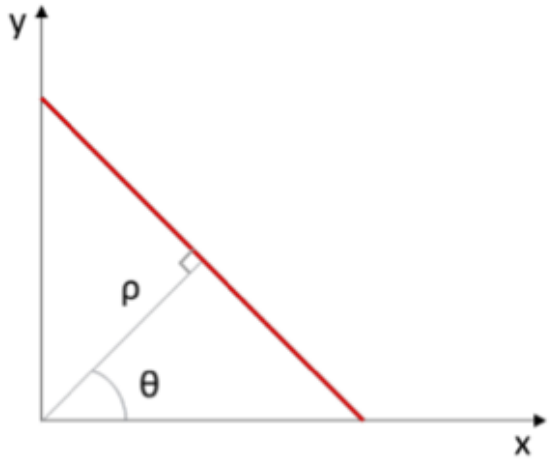
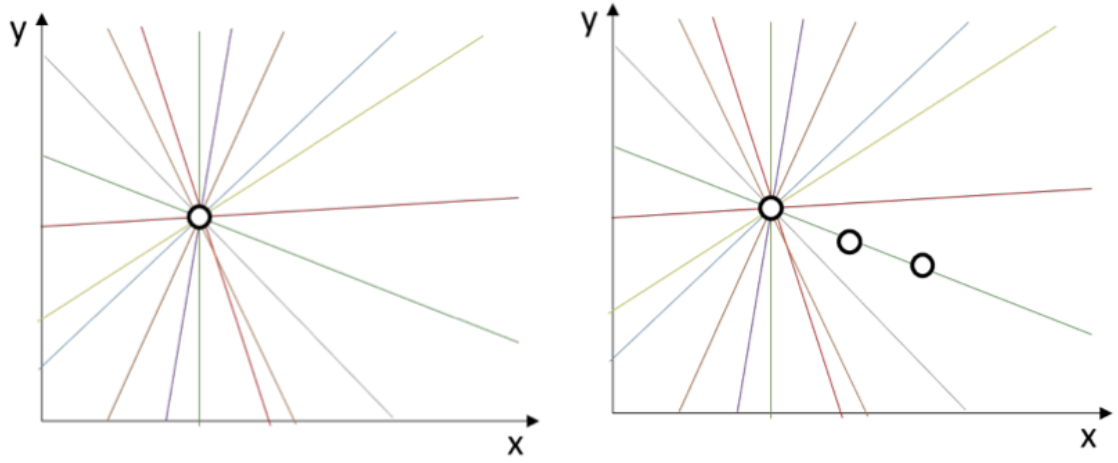


Figura 1.20: Representação dos valores de ρ e θ na equação

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.

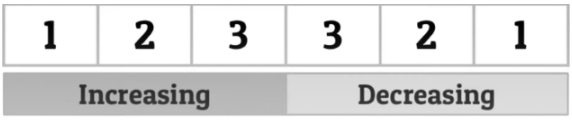


Figura 1.22: Exemplo de sequência bitônica.

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.

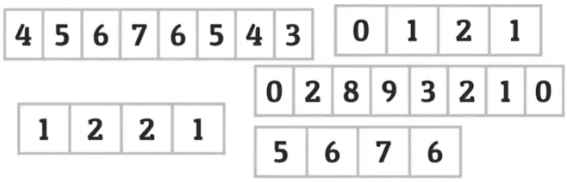


Figura 1.23: Outros exemplos de sequências bitônicas.

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.

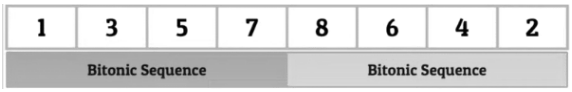


Figura 1.25: Agrupando duas sequências em uma.

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.

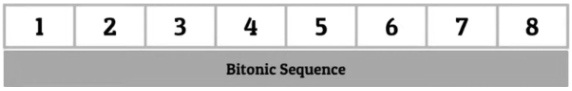


Figura 1.26: Sequência resultante do processo de BitonicSort.

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.

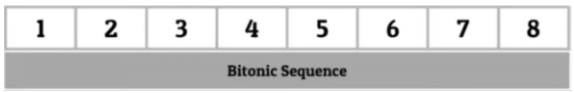


Figura 1.28: Sequências menores mescladas para atingir o resultado final.

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.

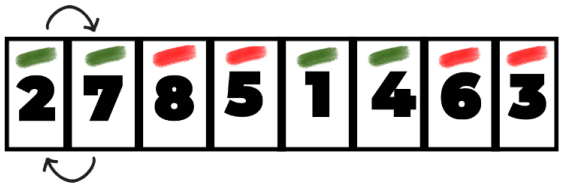


Figura 1.31: Sequências menores de tamanho 2 crescentes e decrescentes de forma intercalada.

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.

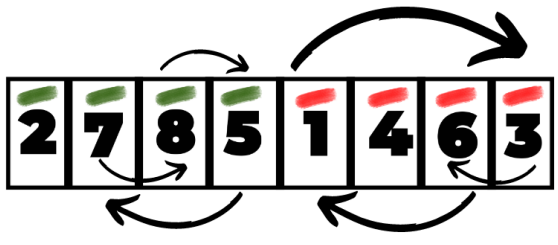


Figura 1.33: Sequências menores de tamanho 2 crescentes e decrescentes de forma intercalada.

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.

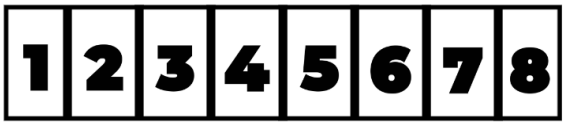


Figura 1.35: Sequência final após a execução do algoritmo.

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.

```
void ParallelBitonicSort(int data_gpu[], int n, queue &q) {
  int size = pow(2, n);
  int *a = data_gpu;

  for (int step = 0; step < n; step++) {
    for (int stage = step; stage >= 0; stage--) {
      int seq_len = pow(2, stage + 1);
      int two_power = 1 << (step - stage);

      // Offload the work to kernel.
      q.submit([&](auto &h) {
        h.parallel_for(range<1>(size), [=](id<1> i) {
          // Assign the bitonic sequence number.
          int seq_num = i / seq_len;
          int swapped_ele = -1;
          int h_len = seq_len / 2;
          if (i < (seq_len * seq_num) + h_len) swapped_ele = i +
  ↪  h_len;
          int odd = seq_num / two_power;
          bool increasing = ((odd % 2) == 0);
          if (swapped_ele != -1) {
            if (((a[i] > a[swapped_ele]) && increasing) ||
                ((a[i] < a[swapped_ele]) && !increasing)) {
              int temp = a[i];
              a[i] = a[swapped_ele];
              a[swapped_ele] = temp;
            }
          }
        });
      });
      q.wait();
    }  // end stage
  }    // end step
}
```

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