Capítulo

1

Desvendando o Uso de Contadores de *Hardware* para Otimizar Aplicações de Inteligência Artificial

Valéria S. Girelli
vsgirelli@inf.ufrgs.br
Grupo de Processamento Paralelo e Distribuído (GPPD)
Universidade Federal do Rio Grande do Sul (UFRGS)
Sala 209, Prédio 67, Instituto de Informática - Campus do Vale
91501-970 - Porto Alegre - RS - Brasil

Félix D. P. Michels
felix.junior@inf.ufrgs.br
Grupo de Processamento Paralelo e Distribuído (GPPD)
Universidade Federal do Rio Grande do Sul (UFRGS)
Sala 201, Prédio 67, Instituto de Informática - Campus do Vale
91501-970 - Porto Alegre - RS - Brasil

Francis B. Moreira
fbm@inf.ufpr.br
High Performance Systems (HiPES)
Universidade Federal do Paraná (UFPR)
Sala 84, Departamento de Ciência da Computação - Centro Politécnico
81531-980 - Curitiba - PR - Brasil

Philippe O. A. Navaux
navaux@inf.ufrgs.br
Grupo de Processamento Paralelo e Distribuído (GPPD)
Universidade Federal do Rio Grande do Sul (UFRGS)
Sala 210, Prédio 67, Instituto de Informática - Campus do Vale
91501-970 - Porto Alegre - RS - Brasil

Resumo

O desempenho dos sistemas computacionais aumentou consideravelmente nas últimas décadas. Tal avanço se deu por meio de mecanismos que nem sempre são visíveis para o usuário final, como o sistema de memória e o sistema de prefetching, que possuem grande impacto no desempenho de processadores modernos. Ao mesmo tempo, algoritmos de Inteligência Artificial (IA) se tornam cada vez mais relevantes em diversas áreas da computação e da sociedade, e requerem um crescente poder computacional. Com isso, para se obter o máximo desempenho dessas aplicações, é necessário garantir que as mesmas estejam utilizando da melhor forma esses recursos. Para auxiliar na avaliação da utilização desses mecanismos transparentes ao usuário, muitos processadores e aceleradores modernos fornecem contadores de hardware, estruturas que permitem o monitoramento de eventos internos, como o número de acessos à memória e a porção de dados encontrados em cada nível de memória. Portanto, neste capítulo abordaremos a utilização de contadores das arquiteturas Intel Xeon Cascade Lake e NEC SX-Aurora TSUBASA para analisar o desempenho das cada vez mais frequentes aplicações de IA. Por meio das ferramentas Linux perf e NEC FTRACE é possível acessar esses contadores e utilizar os resultados para identificar gargalos nessas aplicações.

1.1. Introdução

Aplicações de Inteligência Artificial (IA) vêm ganhando cada vez mais relevância nos mais diversos espaços da sociedade (A et al., 2019). De jogos como Xadrez e Poker, ao tratamento de doenças como câncer, análise de mudanças climáticas, reconhecimento visual, de fala e detecção de fraudes em transações bancárias, são quase incontáveis os campos do nosso dia a dia nos quais a IA tem se introduzido. Diversos trabalhos também propõem a aplicação de IA para aprimorar o desempenho de sistemas computacionais, como mecanismos de *prefetching* (Liao et al., 2009; Peled et al., 2015; BHATIA et al., 2019) e de predição de desvio (Zangeneh et al., 2020; Zhang et al., 2020), que fazem uso de heurísticas na tomada de decisão.

A crescente relevância da área é acompanhada pelo aumento na quantidade de dados sobre os quais as aplicações de IA trabalham. A quantidade de dados e informações digitais no mundo hoje ultrapassa os 44 trilhões de gigabytes. Dessa forma, surge a necessidade de um poder computacional cada vez maior, e muitas empresas migram seus serviços para grandes servidores de processamento de dados com milhares de núcleos e centenas de GPUs em busca de tempos de execução menores. Com a utilização desses sistemas computacionais, surge também a preocupação com o consumo energético e com a refrigeração. É necessário, portanto, garantir que aplicações de Inteligência Artificial estejam utilizando eficientemente os recursos computacionais a sua disposição, e desenvolvedores e desenvolvedoras das diversas bibliotecas voltadas à IA empregam grandes esforços na otimização de suas ferramentas.

Além de aplicações de IA, diversas aplicações de *High-Performance Computing* (HPC) fazem uso desses sistemas equipados com centenas de núcleos e GPUs. A importância de se alcançar o mais alto desempenho possível nessas aplicações levou a uma grande variedade de ferramentas de análise de desempenho. Tais ferramentas permitem a identificação de comportamentos que levam à redução de desempenho e auxiliam no

desenvolvimento de otimizações nas aplicações. Exemplos dessas ferramentas são o *framework* HPCToolkit (ADHIANTO et al., 2009), Periscope (GERNDT; FÜRLINGER; KEREKU, 2005), o projeto TAU (SHENDE; MALONY, 2006), Vampir (KNÜPFER et al., 2008) e Score-P (MEY et al., 2012). No entanto, as informações providas por esses mecanismos são geralmente de mais alto nível e menos detalhadas. Com isso, a identificação de gargalos provenientes da utilização ineficiente dos componentes disponíveis na arquitetura se torna mais desafiadora. Além disso, por serem ferramentas complexas e que coletam informações em diferentes níveis do sistema ao mesmo tempo, o *overhead* e ruído sobre a execução das aplicações tendem a ser altos.

Uma alternativa a essas ferramentas é a utilização de contadores de *hardware*, estruturas encontradas em muitos processadores e aceleradores modernos que permitem o monitoramento de eventos internos a essas arquiteturas. Alguns desses eventos são o número de instruções executadas, o número de ciclos, o número de acessos à memória, dentre outros. Por meio da utilização desses contadores é possível realizar a coleta de informações de forma mais específica e detalhada se comparado às informações obtidas com outras ferramentas de mais alto nível. O usuário tem a possibilidade de identificar as informações que estão disponíveis para sua arquitetura específica e combinar diferentes contadores afim de investigar aspectos distintos. Além disso, contadores de diferentes núcleos também podem ser combinados, analisando-se o sistema como um todo. Dessa forma, utilizar contadores de *hardware* para analisar o desempenho de aplicações paralelas permite que o usuário tenha mais controle sobre o processo, com menos ruído sobre a aplicação.

Portanto, ao longo deste capítulo iremos estudar aspectos de duas arquiteturas distintas, alguns de seus contadores de *hardware* e como utilizar ferramentas de *profiling* para acessar essas informações. Com base nisso, será possível analisar o desempenho de aplicações de Inteligência Artificial e propor otimizações em seus códigos.

O capítulo está organizado da seguinte maneira: na Seção 1.2 é feita uma introdução à arquitetura de computadores, demonstrando conceitos como *pipeline*, arquiteturas superescalares, entre outros. Em seguida, na Seção 1.3 temos uma discussão sobre hierarquia de memória e sistema de *prefetching*. A Seção 1.4 é apresentada uma breve explicação sobre vetorização, exemplificando com a arquitetura vetorial NEC SX-Aurora TSUBASA. A Seção 1.5 expõe os ambientes de execução utilizados na elaboração desse minicurso. A Seção 1.6 apresenta os contadores de *hardware* e a utilização das ferramentas Linux perf e o NEC FTRACE. Na Seção 1.7, conceitos de Inteligência Artificial e Aprendizado de Máquina são apresentados, bem como a aplicação utilizada neste minicurso. Por fim, na Seção 1.8 utilizamos exemplos práticos aplicados em uma implementação de retro-propagação, otimizando-a por meio da utilização de contadores de *hardware*, seguindo para a conclusão deste curso, na Seção 1.9.

1.2. Arquitetura de Computadores

A rápida evolução na área de arquitetura de computadores é baseada em três fatores relacionados: tamanho dos componentes, paralelismo entre componentes, e especulação (HENNESSY; PATTERSON, 2017). O tamanho dos componentes, ou tamanho do processo de fabricação, define a largura em nanômetros dos transistores do sistema. Quanto menor

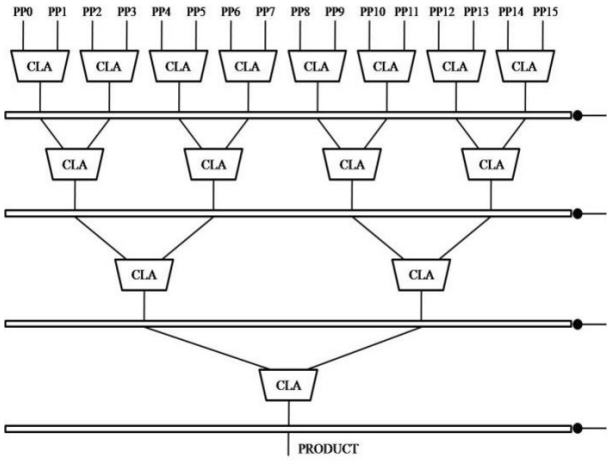

Figura 1.1. Multiplicador de 32 bits com carry pré-calculado. Fonte: Bokade et al. (BOKADE; DAKHOLE, 2016), pág. 5.

o transistor, mais *hardware* conseguimos colocar dentro do *chip* do processador ou memória, menor a latência de fio dos componentes e entre os mesmos, e portanto maior a frequência de operação (FLOYD, 2010). Já o paralelismo implica em usar mais *hardware* para produzir mais trabalho no mesmo período de tempo, sendo usado em todos os níveis de arquitetura. Um exemplo claro é a diferença existente entre um somador por propagação e um somador com *carry* pré-calculado. O somador normal precisa de um tempo igual à soma dos tempos de todos somadores já que cada somador depende do resultado do somador predecessor. Já o somador com *carry* pré-calculado utiliza *hardware* adicional para calcular *carries* em paralelo, permitindo que todos os somadores terminem seu serviço em paralelo. Este paralelismo pode ser considerado o paralelismo a nível de "operação", onde torna-se uma operação mais eficiente ao adicionar *hardware* para tal fim (FLOYD, 2010).

1.2.1. *Pipeline*

Uma técnica muito utilizada para paralelismo é a técnica de *pipeline* (HENNESSY; PATTERSON, 2017). Como operações de multiplicação e divisão são muito demoradas, é comum dividir elas em vários estágios com o uso de registradores intermediários, como na Figura 1.1. Assim é possível reduzir o caminho crítico do sistema e manter uma frequência de operação mais alta. Com isso, tem-se capacidade de realizar mais operações em menos tempo, pois conforme uma instrução termina um estágio, este está pronto para começar uma nova instrução. A ideia de *pipeline* foi adotada em larga escala para todo o funcionamento do processador. Processadores antigos utilizam de frequência mais baixa e um conjunto grande de instruções complexas, recebendo a caracterização de CISC (*Complex Instruction Set Computer*) (ISEN; JOHN; JOHN, 2009). Em 1981, Hennessy desenvolveu o processador MIPS (*Microprocessor without Interlocked Pipeline Stages*) (HENNESSY et al., 1982), que implementa um conjunto pequeno de instruções simples, permitindo uma grande inovação: o uso de *pipeline* para a operação de todas as instruções. Hoje este tipo de computador é conhecido como RISC (*Reduced Instruction Set Computer*) (ISEN; JOHN; JOHN, 2009).

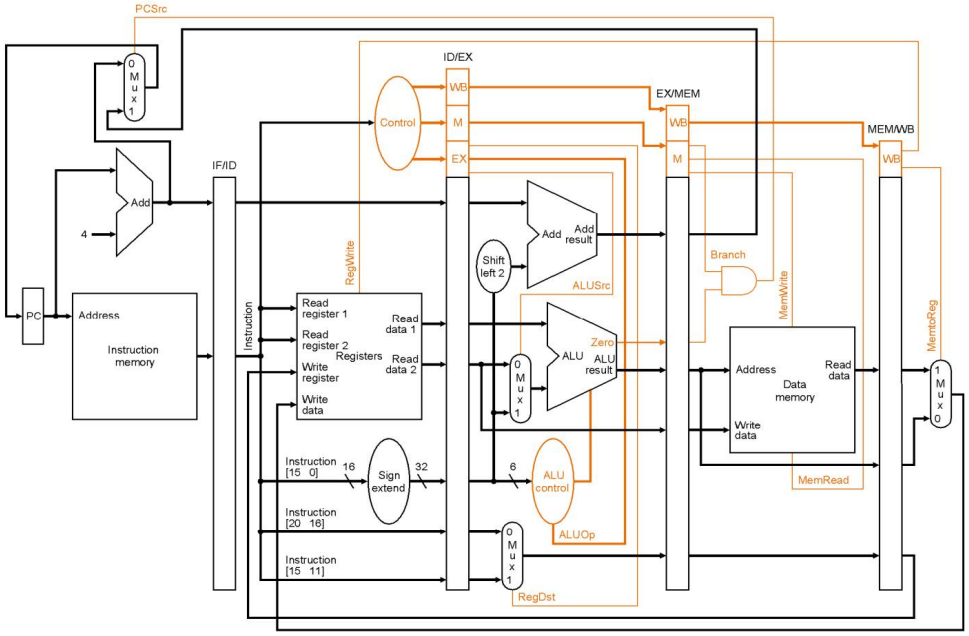

Figura 1.2. *pipeline* da arquitetura MIPS. Fonte: Hennessy & Patterson (PATTERSON; HENNESSY, 2004), pág. 387

A ideia, demonstrada na Figura 1.2, é separar as instruções em 5 estágios: busca (IF), decodificação (DEC), execução (EXE), acesso à memória (MEM), e escrita (WB). Assim, quando uma instrução termina de executar um estágio e passa para o próximo, o processador já pode receber uma nova instrução. Isto permite um paralelismo a nível de operação do processador, pois temos múltiplas instruções no processador, utilizando seus diferentes estágios, embora realisticamente ainda entregaremos no máximo uma instrução por ciclo. O *pipeline* também permite um grande aumento na frequência ao reduzir o caminho crítico para o tempo de um estágio, e faz melhor uso dos recursos do processador, pois boa parte dos recursos ficava inativa enquanto as instruções não estivessem usando elas. Um dos problemas do *pipeline* é que existem dependências entre certas operações como saltos e leitura após escrita, onde é necessário inserir bolhas no *pipeline* para operações terem os valores corretos. Assim o número de instruções entregues por ciclo geralmente é menor que um, o que é resolvido através de técnicas de especulação, como predição de salto (YEH; PATT, 1991).

1.2.2. Arquiteturas Superescalares

Neste mesmo princípio, nota-se que várias unidades funcionais ficam inativas quando outra operação está sendo executada. Assim, desenvolveu-se a arquitetura superescalar (THORNTON, 1980), com o propósito de aumentar o paralelismo a nível de instrução (*instruction level parallelism* – ILP). A arquitetura de *pipeline* superescalar é basicamente uma arquitetura *pipeline* de maior capacidade: o processador é capaz de buscar múltiplas instruções em um ciclo, decodificar múltiplas instruções em um ciclo, mandar múltiplas instruções para execução nas diferentes unidades funcionais, mandar múltiplas requisições para a memória, e fazer múltiplas escritas nos registradores. Uma arquitetura

superescalar não é necessariamente uma arquitetura *pipeline*, pois as técnicas são distintas (HENNESSY; PATTERSON, 2017). Para o correto funcionamento, são necessárias várias adições ao processador, como *buffers* entre os estágios, a adição de estágios como a renomeação de registradores (o qual elimina dependências falsas entre as instruções) e o estágio de despacho (o qual possui lógica inteligente adicional para despachar instruções na melhor ordem conforme o estado do processador), um *buffer* de reordenação de instruções (*reorder buffer* - ROB), *buffers* de ordem para requisições à memória (*memory order buffer* - MOB), entre outros (FOG, 2012).

1.2.3. Threading

A adição de tantas estruturas para melhorar o desempenho de um *pipeline* superescalar inseriu problemas nas arquiteturas modernas. Problemas como a espera por resolução de saltos e por acessos à memória limitam o número de instruções alcançadas em testes reais. O número de instruções por ciclo (*instructions per cycle* – IPC) na maior parte dos programas não passa de 2, apesar de processadores dimensionados para mais de 4 instruções em paralelo. Para dados contíguos, adotou-se vetorização, que é o paralelismo no nível de operação da instrução, a qual agora recebe mais dados para uma operação. Para programas mais complexos, criou-se um novo nível de paralelismo: o paralelismo de *threads* (*Thread Level Parallelism* – TLP). Agora, o processador é capaz de escalonar mais de um fluxo de programa ao manter múltiplos contadores de programa (*Program Counter* - PC) e conjuntos de registradores lógicos para representar o estado de diferentes *threads* (fios). Este suporte ocorre em dois níveis: arquiteturas *multithreaded* e arquiteturas multi-core.

Em arquiteturas *multithreaded*, o núcleo de processamento possui a capacidade de dividir os seus recursos entre múltiplas *threads* (TULLSEN; EGGERS; LEVY, 1995). O *buffer* de reordenamento, as estações de espera por execução em cada unidade funcional, o banco de registradores, e todas as outras estruturas de controle são divididas entre as múltiplas *threads* para que todas ocupem de forma eficiente as unidades funcionais do core. Entre as formas de *multithreading* podemos citar:

- *Multithreading* entrelaçado, no qual o núcleo busca uma instrução de cada *thread* a cada ciclo;
- *Multithreading* em bloco, onde a cada período de ciclos o processador busca várias instruções da mesma *thread*, e troca a *thread* quando acaba o período;
- *Multithreading* simultâneo, hoje adotado pela Intel como *Hyperthreading* (MARR et al., 2002), onde instruções de todas as *threads* são buscadas no mesmo ciclo.

Em arquiteturas multi-core, possuímos múltiplos *cores*, os quais podem ter apenas uma *thread*, ou podem ser *multithreaded* (BLAKE; DRESLINSKI; MUDGE, 2009). O sistema operacional encarrega-se de gerenciar os recursos dos múltiplos *cores*, permitindo o processamento paralelo e o mapeamento da execução conforme desejado nos *cores* disponíveis.

1.2.4. Especulação

Através da evolução da arquitetura, vários problemas foram identificados na busca de execução eficiente. Mecanismos de especulação tentam contornar estes problemas ao prever o comportamento da aplicação em relação a eles. Por exemplo, a latência da memória se tornou cada vez maior devido à evolução muito mais rápida do processador, tornando o acesso a instruções e dados um gargalo. Para diminuir a latência da memória, a memória *cache* foi adotada, a qual serve como armazenamento temporário para linhas de memória, com tamanho e latência muito menores (JACOB; WANG; NG, 2010). A primeira premissa da memória *cache* é de que um dado recentemente usado será reusado em breve, o que é uma especulação ou predição em relação ao comportamento da aplicação, onde assume-se localidade temporal. A segunda premissa comum em memória *cache* ao usar linhas longas com múltiplos endereços é de que dados contíguos serão usados em um curto espaço de tempo, portanto assume-se localidade espacial, o que é praticamente sempre válido para acessos à instruções na memória, por exemplo. Entre outros mecanismos de especulação, podemos citar a desambiguação de leituras, a predição de saltos, *prefetching*, *buffers* de linha na memória principal (*Dynamic Random-Access Memory* – DRAM) (JACOB; WANG; NG, 2010), entre outros.

1.3. Hierarquia de Memória e Sistema de *Prefetching*

Em processadores modernos, uma hierarquia de *cache* em três níveis é comumente usada (MORGAN, 2017; CUTRESS, 2017). Nessa configuração, o primeiro nível de *cache* de dados (L1) e o segundo nível de *cache* de dados (L2) são normalmente privados a cada núcleo do processador. Esses níveis de *cache* são mais próximos fisicamente do processador, possuem menor capacidade de armazenamento, e permitem acesso aos dados de forma mais eficiente. Um terceiro nível de *cache* (L3, também conhecida por *Last Level cache* – LLC) é compartilhada entre todos os núcleos do processador. Seu tempo de resposta é frequentemente maior que o tempo de resposta dos níveis de *cache* privados, mas com a vantagem de permitir uma capacidade de armazenamento maior. Uma representação dessa hierarquia é observada na Figura 1.3.

Quando o processador emite uma requisição por um dado na memória, diversas situações podem ocorrer. Inicialmente, a requisição é entregue à *cache* L1, que é relativamente pequena (32 KiB) e possui uma baixa latência de acesso (4 ciclos de processador) (FOG, 2012; HENNESSY; PATTERSON, 2017). Caso o dado seja encontrado nesse nível de *cache* ele é rapidamente entregue ao processador. No entanto, devido à capacidade limitada da *cache* L1, por muitas vezes os dados não estão presentes, e a busca pelo dado é repetida no próximo nível de *cache*. Cada vez que um dado não é encontrado em um nível de *cache* tem-se um *cache miss* e a necessidade de repetir o procedimento de busca em um nível mais distante do processador, cujo tempo de acesso é maior (e somado aos tempos de acesso dos níveis de memória predecessores). Portanto, encontrar os dados solicitados em níveis de *cache* mais próximos do processador é preferível, caso contrário, a hierarquia da memória pode se tornar um grande gargalo para o desempenho das aplicações (BAKHSHALIPOUR et al., 2019).

Cabe ressaltar, no entanto, que diversas outras ações são necessárias juntamente com o processo de busca por dados descritos acima, como por exemplo o acesso à *Trans-*

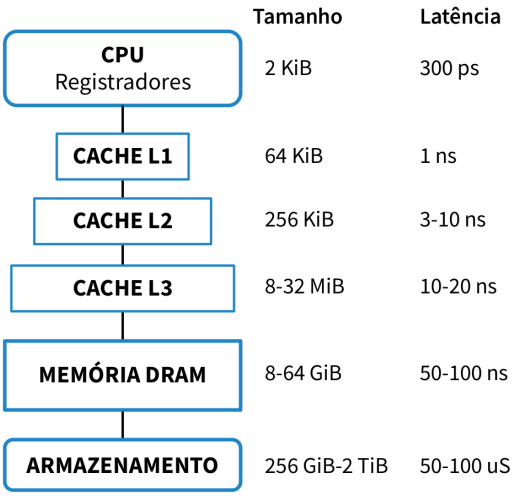

Figura 1.3. Exemplo da hierarquia de memória de um processador moderno. Conforme nos afastamos do processador, as memórias se tornam maiores e mais lentas.

lation Lookaside Buffer (TLB) (onde é possível verificar se a página de dados está fisicamente presente na memória principal e a tradução do endereço virtual para o físico (HENNESSY; PATTERSON, 2017)), possíveis acessos à memória principal para consulta da tabela de páginas (um procedimento custoso devido à alta latência da memória DRAM), e ainda transferências de dados entre duas *caches* de diferentes núcleos do sistema devido à ação do protocolo de coerência de *cache*.

Nos últimos anos, várias melhorias no desempenho do processador têm sido observadas, como o aumento do número de núcleos – o que requer memórias com maior largura de banda de transferência de dados para lidar com as requisições de dados emitidas por esses diversos núcleos, e a capacidade do processador de requisitar vários dados por ciclo (*multiple issue*) (HENNESSY; PATTERSON, 2017). No entanto, as tecnologias de memória não melhoraram tanto quanto os processadores, criando uma lacuna de desempenho referida na literatura como *Memory Wall* (WULF; MCKEE, 1996). Vários problemas podem surgir dessa disparidade de desempenho. Por exemplo, se uma instrução for um *load* (requisição de dado para leitura) e seus dados necessários não forem entregues rapidamente pelo sistema de memória, a execução dessa instrução e das instruções dependentes a ela podem ser interrompidas (HENNESSY; PATTERSON, 2017). Para evitar tais paralisações, deve-se reduzir o número de ciclos desde o momento em que o processador emite uma requisição até o momento em que pode realmente usar os dados deve ser o menor possível. Além disso, dada a natureza de *multiple issue* dos processadores modernos, um grande número de solicitações de memória pode ser emitido em apenas alguns ciclos, possivelmente criando contenção em algum nível da hierarquia de memória.

Diante desses vários problemas, o *prefetcher* foi criado para mitigar a latência da memória (BAER; CHEN, 1991). *Prefetching* é uma técnica implementada em *hardware* que visa prever quais serão os próximos endereços de memória a serem solicitados pelo processador. Ao monitorar as solicitações de memória anteriores, o *prefetcher* é capaz

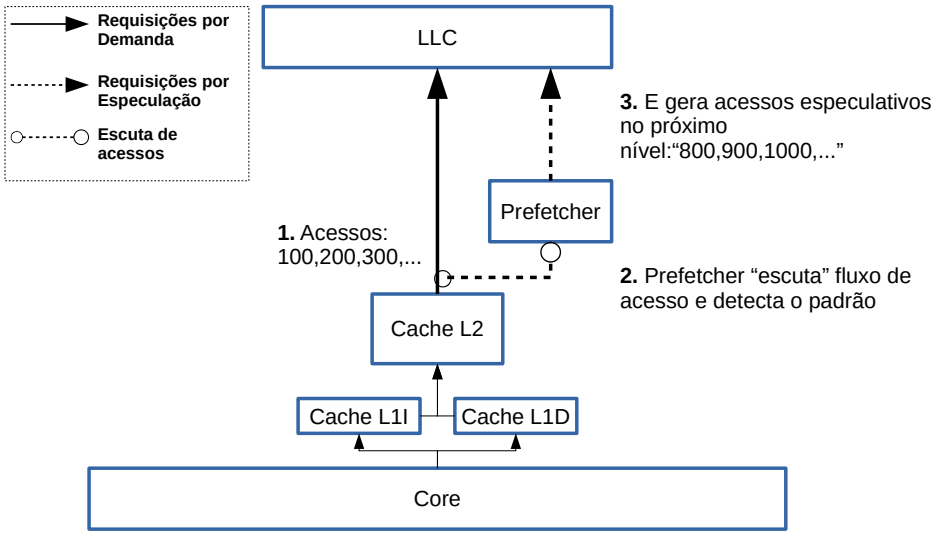

Figura 1.4. Abstração do comportamento de um *prefetcher*.

de identificar possíveis padrões de acesso. Com base nesses padrões, ele especula quais podem ser os próximos endereços a serem solicitados e, em seguida, realiza requisições com antecedência, antes que o processador realmente precise dos dados. Assim, quando o dado for finalmente solicitado pelo processador, ele já estará em níveis de *cache* mais próximos (HENNESSY; PATTERSON, 2017). A latência da memória principal acima mencionada é, portanto, ocultada por outras instruções anteriores à instrução que de fato realizou a requisição dos dados buscados pelo *prefetcher*.

Com os dados já em níveis mais próximos, (i) a crítica *load-to-use latency* pode ser reduzida (KANG; WONG, 2013; Guttman et al., 2015), e (ii) uma importante métrica de desempenho é melhorada, a taxa de acertos da *cache*, também conhecida como *cache hit*. A taxa de *hits* representa a porção de requisições que são encontradas em um determinado nível de *cache* sem a necessidade de se aprofundar na hierarquia de memória – que, consequentemente, resultaria em um tempo de execução maior. Esses ganhos de desempenho permitiram que os *prefetchers* se tornassem um mecanismo predominante nas arquiteturas atuais (MORGAN, 2017; CUTRESS, 2017; FOG, 2012; GIRELLI et al.,). Exemplos de padrões identificados por mecanismos de *prefetcher* comuns são *stride* (CHEN; BAER, 1995) e *stream* (LE et al., 2007).

A Figura1.4 mostra um exemplo de *prefetcher* da *cache* L2 detectando um padrão de acesso *stride*. A *cache* L2 encaminha requisições para a LLC (mostrado na Figura1.4 como o evento **1**). O *prefetcher* da L2, por sua vez, intercepta essas solicitações "escutando"a interconexão da *cache* (evento**2**) e identificando o padrão de acesso que está sendo gerado. Com base no padrão identificado, requisições especulativas são inseridas no *Miss Status Holding Register* (MSHR) da *cache* L2 (**3**), um *buffer* que mantém o controle de eventos de *miss* que ainda precisam ser tratados. Essas requisições especulativas inseridas no MSHR da L2 são feitas primeiramente à *cache* L2 para evitar a busca redundante de um dado que já reside na L2. Esses acessos são vistos como solicitações regulares feitas à L2 pelo *prefetcher*, de modo que a L2 não precisa realmente encaminhar a resposta para a

L1. Se o endereço especulado ainda não estiver presente na L2, a L2 encaminha a requisição por dado para os próximos níveis da hierarquia, como em um acesso normal. Assim, quando o processador precisar de um dado solicitado previamente pelo *prefetcher*, ele já estará em um nível de *cache* mais próximo (neste caso, a *cache* L2).

1.4. Vetorização

A característica chave das arquiteturas vetoriais é seu modelo *Single Instruction Multiple Data* (SIMD). Em processadores superescalares, o dado padrão é uma palavra de normalmente 32 *bits* sobre a qual um conjunto de instruções atua de maneira individual. Já em uma arquitetura vetorial, uma instrução vetorial é aplicada simultaneamente sobre uma coleção de palavras em formato de vetor (HENNESSY; PATTERSON, 2017). Desse modo, tem-se a execução de uma mesma instrução (*single instruction*) sobre múltiplos dados (*multiple data*) em um único ciclo. Por conta disso, processadores vetoriais têm a vantagem de que cada dado é independente entre si, o que permite que a mesma instrução seja realizada sobre todos eles ao mesmo tempo.

Por operar em um número maior de dados de uma única vez, instruções vetoriais resultam em menos buscas por dados e menos *branches* quando os dados estão contíguos. Com isso é possível reduzir-se o número de erros de predição e a latência de acesso à memória, favorecendo o tempo de execução de uma aplicação(KSHEMKALYANI, 2012). Porém, essa vantagem é relevante apenas quando há blocos de memória suficientemente grandes, onde haveria uma grande latência de acesso a memória em um processador escalar tradicional.

Um exemplo de arquitetura vetorial é a SX-Aurora TSUBASA da empresa NEC Corporation. Esse processador possui 8 núcleos de processamento executando com frequência de 1,408 GHz e 3 níveis de memória *cache* (KOMATSU et al., 2018). Uma das vantagens dessa arquitetura em relação as outras existentes é o tamanho das unidades vetoriais da mesma, podendo chegar a 256 elementos de 64 bits. Além disso, o compilador da NEC (NEC, 2020a) toma decisões automaticamente , identificando áreas nas quais é possível gerar código vetorizável sem que haja a necessidade de alteração do código fonte. Entretanto, o compilador ainda necessita de ajuda do programador para facilitar a interpretação do código. O programador pode utilizar diretrizes específicas e utilizar técnicas de otimização como *loop unrolling* e *inlining*.

1.5. Ambiente de Execução

O ambiente de execução utilizado nos experimentos estão presentes na infraestrutura PCAD[1], no INF/UFRGS. Na Tabela 1.1 tem-se as características do *Vector Engine* (VE) TSUBASA. A máquina vetorial consiste em um ambiente com 8 cores, 48GB de memória global a 900 MHz e *cache* L3 compartilhada de 2 MB. Cada core possui memórias *caches* privadas, L1 de instrução e dados de 32KB cada e L2 de 256KB, uma unidade de processamento escalar (*Scalar Processing Unit* – SPU) e uma unidade de processamento vetorial (*Vector Processing Unit* – VPU), sendo que cada VPU contém *load buffer*, *store buffer*, e 32 *pipelines* paralelos vetoriais (*Vector Parallel Pipeline* – VPP) (NEC, 2020b). A maior parcela dos cálculos é feita pela VPU, sendo a SPU responsável pelos traba-

[1]<http://gppd-hpc.inf.ufrgs.br>

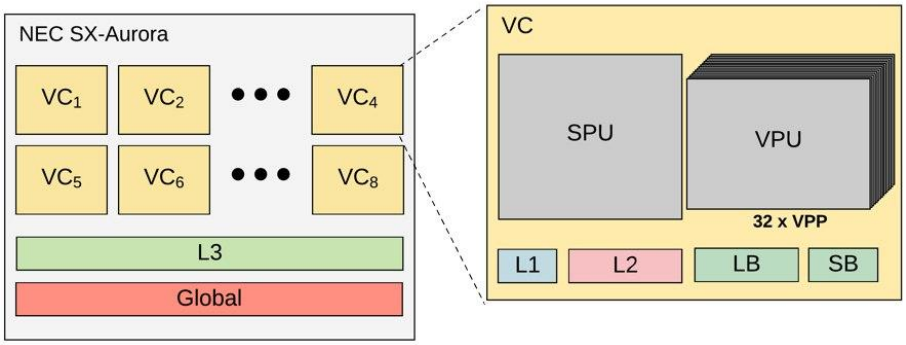

Figura 1.5. Ambiente de execução SX-Aurora.

Tabela 1.1. Arquitetura SX-Aurora (Vector Engine Type 10BE).

Processador	8 *cores* @ 1408 MHz
Microarquitetura	SX-Aurora
Cache	8 X 32 KB L1I; 8 X 32 KB L1D; 8 X 256 KB L2; 8 X 2 MB L3
Memória	HBM2 48 GB, 900 MHz

lhos sequenciais, bem como as tarefas do sistema operacional. Na figura 1.5 temos uma representação gráfica desses componentes.

Já o *Vector Host* (VH) é representado pela microarquitetura Intel Cascade Lake. Na Tabela 1.2 tem-se as especificações do processador Intel Xeon Gold 6226, que possui 12 núcleos operando a uma frequência entre 2.7 GHz e 3.7 GHz. Cada núcleo possui 32 KB de *cache* L1 de dados e instruções, bem como uma cache L2 também privada de 1 MB. A L3, compartilhada entre todos os núcleos, possui capacidade de 16,5 MB, e a máquina ainda apresenta 192 GB de memória DRAM. A microarquitetura Cascade Lake é bastante semelhante à sua predecessora Skylake, porém a Cascade Lake introduz suporte à instruções de Rede Neural Vetorial AVX-512 (*AVX-512 Vector Neural Network Instructions* – VNNI) (PEREZ et al., 2018).

1.6. Contadores de Hardware

Após compreendermos melhor o funcionamento de arquiteturas superescalares e vetoriais e alguns de seus pontos de perda de desempenho, agora é necessário compreender-

Tabela 1.2. Microarquitetura Cascade Lake (Xeon Gold 6226).

Processador	12 *cores* @ 2700 - 3700 MHz;
Microarquitetura	Cascade Lake
Cache	12 X 32 KB L1I; 12 X 32 KB L1D; 12 X 1 MB L2; 16,5 MB L3
Memória	DDR4 192 GB, 2933 MHz

mos como obter informações a respeito do *hardware* no momento da execução de uma aplicação afim de melhorarmos o desempenho de uma aplicação. Arquiteturas modernas permitem a utilização de contadores de *hardware*, estruturas internas que permitem o monitoramento de eventos da execução e funcionamento do processador ou acelerador. Como já citado anteriormente, diversas ferramentas de *profiling* foram desenvolvidas que permitem acesso a contadores de *hardware*. Neste capítulo, nós apresentamos os contadores existentes na arquitetura Intel Cascade Lake e no processador vetorial SX-Aurora TSUBASA e como acessá-los por meio das ferramentas Linux perf e NEC FTRACE, respectivamente.

1.6.1. Linux perf

Linux perf é uma ferramenta de *profiling* para sistemas Linux que permite acesso à interface `perf_events` de eventos de desempenho presentes em arquiteturas superescalares. A utilização da ferramenta é feita por linha de comando por meio do comando `perf`, e é possível listar os eventos disponíveis na plataforma por meio do comando `perf list`. Um comando comumente utilizado para análise de desempenho é o comando `stat`, que permite a coleta de informações referentes à execução de uma aplicação. Com a utilização da opção `-e` é possível listar os eventos que se deseja coletar. O exemplo abaixo representa a linha de comando necessária para a coletar informações a respeito do número de instruções e do número de ciclos de CPU necessários para a execução do comando `ls`:

```
1 perf start -e instructions,cpu-cycles ls
```

Além de realizar a execução do comando `ls` e listar o conteúdo do diretório atual, a execução do comando `perf stat` com os eventos `instructions` e `cpu-cycles` dá um resultado semelhante ao observado abaixo:

```
1 Performance counter stats for 'ls':
2       1.309.712  instructions  # 0,71 insnt per cycle
3       1.848.886  cycles
4     0,000800891 seconds time elapsed
```

É importante notar que as arquiteturas superescalares normalmente possuem um número limitado de contadores de *hardware*. Uma vez que cada contador de *hardware* pode ser utilizado para monitoramento de um único evento a cada dado momento, a quantidade de eventos que podem ser monitorados simultaneamente é limitada. Para possibilitar o monitoramento de um número maior de eventos, a ferramenta perf realiza uma técnica de multiplexação do tempo de monitoramento, permitindo que cada evento possua uma parcela do tempo de utilização do contador. No entanto, o que esta abordagem possibilita é apenas uma estimativa do real comportamento da aplicação, uma vez que os eventos multiplexados não são monitorados o tempo todo com exclusividade. Dessa forma, é aconselhável que cada evento seja monitorado individualmente por meio da instrução `perf stat`, realizando o *profiling* da aplicação várias vezes.

1.6.2. NEC FTRACE

Desenvolvida pela empresa NEC, a ferramenta FTRACE pode ser utilizada para a obtenção de informações de contadores de *hardware* presentes na arquitetura vetorial da NEC. Para utilizar a ferramenta é necessário recompilar a aplicação utilizando o compilador de-

senvolvido pela empresa NEC. Para aplicações escritas em linguagem C usamos o `ncc`, para aplicações C++ usamos o `nc++` e para aplicações Fortran usamos o `nfort`. Além disso, é necessário adicionar a *flag* de compilação `-ftrace`, como exibido no exemplo: `ncc -ftrace source.c`. A partir disso, podemos executar a aplicação como faríamos com um executável comum. Ao fim da execução, o arquivo de informações `ftrace.out` é gerado, podendo ser lido por meio da mesma ferramenta (e direcionando a saída para o arquivo *output*):

```
1 ftrace -f ftrace.out >> output
```

O arquivo gerado possui diversas informações. A Figura 1.7 mostra todas as informações de saída que a ferramenta FTRACE proporciona. Dentre elas podemos destacar o tempo total de execução, o nome das funções executadas, o número de vezes que cada função foi chamada, porcentagem de utilização de core por cada função, informações a respeito do número de *misses* dos diversos níveis de *cache*, dentre outros. É possível ainda controlar o tipo de informações que se deseja por meio dos dois diferentes modos de *profiling* indicados por meio da variável de ambiente `VE_PERF_MODE`. Caso a variável possua valor `VECTOR-OP` ou esteja indefinida (assumindo o valor padrão), o FTRACE gera informações relacionadas principalmente às instruções vetoriais. Caso o valor da variável `VE_PERF_MODE` seja `VECTOR-MEM`, os dados levantados correspondem principalmente a acessos à memória. Dessa forma, pode ser proveitoso fazer uso dos dois modos e agregar seus resultados ao fim. Pode-se alterar o valor da variável de ambiente `VE_PERF_MODE`, para um dos dois modos, da seguinte forma:

```
1 export VE_PERF_MODE=VECTOR-OP
2 export VE_PERF_MODE=VECTOR-MEM
```

1.7. Aplicações de Inteligência Artificial

Com a evolução de arquiteturas de processadores e os sistemas agregados, e.g. memória e aceleradores, algoritmos cada vez mais complexos, com grande volume de dados e operações, acharam aplicações práticas em computadores. Uma subárea da computação tornou-se o expoente atual na demanda por sistemas mais poderosos: desenvolvida na estatística, o Aprendizado de Máquina, mais genericamente conhecido como Inteligência Artificial, é a área em voga nos dias de hoje (GADEPALLY et al., 2019).

A base estatística para aprendizado de máquina é a inferência estatística, um ramo da estatística a qual tem por base o estudo de modelos preditivos baseados em amostras de uma população, tendo como principais escolas a Inferência Frequentista e a Inferência Bayesiana (CASELLA; BERGER, 2021). Quanto melhor a qualidade dessa amostra, melhor a capacidade de predição do modelo, o que muitos pesquisadores interpretam também como uma amostra de tamanho maior. Portanto, estes modelos processam um grande volume de dados para ajustar seus parâmetros de modo a melhorar suas predições.

Boa parte destes modelos assume que existe uma função de "erro", a qual depende dos parâmetros do modelo. O algoritmo para treinar um modelo, portanto, consiste em testar várias amostras contra o modelo atual, e ajustar os parâmetros conforme o erro do modelo em prever o comportamento das amostras. Ao chegar em um mínimo local desta função de erro, avalia-se a qualidade do modelo, potencialmente aplicando-se uma reran-

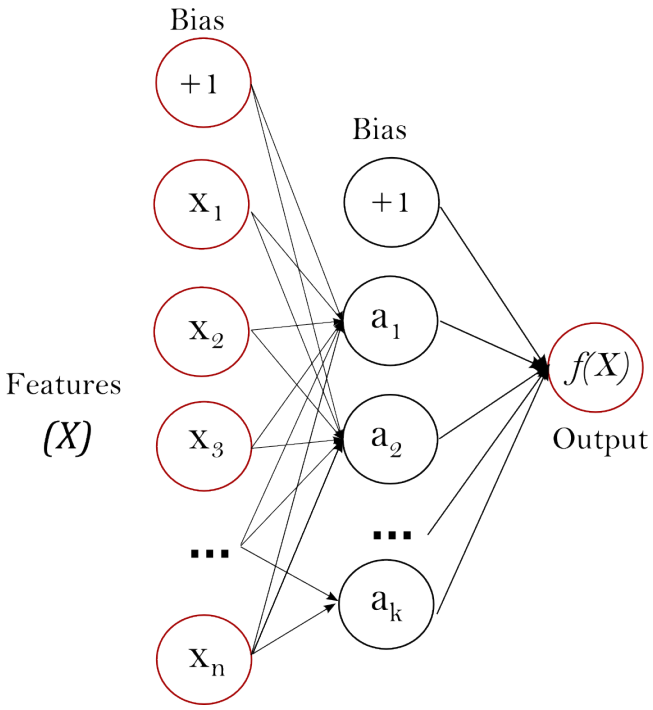

Figura 1.6. Rede Neural do tipo perceptron de múltiplas camadas.

domização de parâmetros para procurar um mínimo local melhor, embora não existam garantias de se achar um mínimo global.

Um dos modelos mais populares para vários problemas é o modelo de Redes Neurais (AGGARWAL et al., 2018). Neste modelo, temos camadas de "neurônios", os quais são representados por uma função de ativação de acordo com suas entradas. Na Figura 1.6, é ilustrado um exemplo de rede neural do tipo perceptron de múltiplas camadas com uma camada de entrada, a qual recebe características da amostra, uma camada oculta, a qual aproxima a função de erro, e uma camada de saída, a qual providencia um valor de predição. Neste exemplo, a rede neural é completamente conectada, onde cada neurônio representa uma soma $\sum(peso[k] * entrada[k])$, onde os valores de entrada vem da camada anterior, e os valores de peso são os parâmetros retidos no neurônio. Portanto, a representação do modelo é a matriz de neurônios e seus parâmetros.

Para atualizar os pesos (parâmetros) de uma rede neural, normalmente utiliza-se o algoritmo de retro-propagação (*Backpropagation*) (GOMEZ et al., 2017). No conjunto de benchmarks Rodinia (CHE et al., 2009), há um exemplo de rede neural onde é possível observar a implementação de *Backpropagation*. Abaixo, podemos observar a função de treino:

```
1 void bpnn_adjust_weights(delta, ndelta, ly, nly, w, oldw)
2 float *delta, *ly, **w, **oldw;
3 {
4   float new_dw;
```

```
5   int k, j;
6   ly[0] = 1.0;
7   //eta = 0.3;
8   //momentum = 0.3;
9
10 #ifdef OPEN
11  //omp_set_num_threads(NUM_THREAD);
12  #pragma omp parallel for \
13     shared(oldw, w, delta) \
14        private(j, k, new_dw) \
15        firstprivate(ndelta, nly)
16 #endif
17  for (j = 1; j <= ndelta; j++) {
18    for (k = 0; k <= nly; k++) {
19      new_dw = ((ETA * delta[j] * ly[k]) + (MOMENTUM *
           oldw[k][j]));
20         w[k][j] += new_dw;
21         oldw[k][j] = new_dw;
22    }
23  }
24 }
```

A função recebe `delta` (o vetor com os erros da próxima camada), `ndelta` (o tamanho deste vetor), `ly` (o vetor com os valores atuais de ativação), `nly` (o tamanho deste vetor), `w` (o novo vetor de pesos a ser criado), e `oldw` (que retém o vetor de pesos anterior). Cada peso é atualizado de acordo com a fórmula `ETA * delta[j] * ly[k]) + (MOMENTUM * oldw[k][j])`, onde `ETA` e `MOMENTUM` são constantes para controlar a velocidade de treino e agressividade de adaptação dos parâmetros. Assim, esta pequena função é chave no funcionamento de redes neurais, e arquiteturas atuais propõe otimizações agressivas para melhorar sua performance. Neste minicurso, utilizamos a implementação do *Backpropagation* acima para ilustrar como podemos explorar a informação obtida por contadores de hardware e ferramentas de *profiling* para se obter desempenho em diversas arquiteturas, notadamente a arquitetura de CPU Cascade Lake e a arquitetura vetorial SX-Aurora TSUBASA.

1.8. Otimizando o Backpropagation com *profiling* e contadores de hardware

1.8.1. NEC SX-Aurora

Analisando o código da aplicação através do trecho abaixo, juntamente com a análise da `ftrace.out` gerado pelo FTRACE observado na Figura 1.7, percebe-se que a função `alloc_1d_dbl` corresponde a 63,7% do tempo de execução. Ao analisar o código abaixo, observa-se que a mesma realiza a alocação dos pesos de forma tradicional, por meio de uma matriz bidimensional. No entanto, essa alocação não favorece o acesso aos dados por conta da falta de localidade espacial dos dados, o que sugere que a alocação de memória pode ser otimizada para uma alocação de memória contígua. Modificar a alocação dos dados gera a necessidade de se alterar várias outras regiões do código, uma

```
*---------------------*
  FTRACE ANALYSIS LIST
*---------------------*

Execution Date : Sun Mar 28 08:41:41 2021 -03
Total CPU Time : 0:02'34"848 (154.848 sec.)


EXECUTION TERMINATED BUT NOT IN MAIN PROCEDURE.

 STOPPED  AT setup
 CALLED FROM main


FREQUENCY  EXCLUSIVE       AVER.TIME     MOPS   MFLOPS  V.OP  AVER.    VECTOR L1CACHE CPU PORT VLD LLC PROC.NAME
          TIME[sec](  % )    [msec]                     RATIO V.LEN      TIME    MISS     CONF HIT E.%

134217738    98.697( 63.7)      0.001    753.4      0.0   0.00   0.0     0.000  23.827    0.000    0.00 alloc_1d_dbl
        1    36.506( 23.6) 36506.441    744.5      0.0   0.00   0.0     0.000   8.087    0.000    0.00 bpnn_free
        4    13.218(  8.5)  3304.383   2853.4      0.0   0.00   0.0     0.000   0.000    0.000    0.00 alloc_2d_dbl
        1     5.524(  3.6)  5524.487    400.6      3.0   0.00   0.0     0.000   0.000    0.000    0.00 load
        1     0.636(  0.4)   636.077   3376.2      0.0   0.00   0.0     0.000   0.005    0.000    0.00 bpnn_read
        2     0.262(  0.2)   131.075   2879.9      0.0  17.78   1.0     0.262   0.000    0.000    0.00 bpnn_zero_weights
       16     0.004(  0.0)     0.267    531.9      0.0   0.01   1.0     0.000   0.000    0.000   93.75 bpnn_layerforward$1
        2     0.003(  0.0)     1.415    541.5      0.0   0.00   1.0     0.000   0.000    0.000   83.33  -thread0
        2     0.000(  0.0)     0.058    488.4      0.1   0.03   1.0     0.000   0.000    0.000  100.00  -thread1
        2     0.000(  0.0)     0.138    520.0      0.0   0.01   1.0     0.000   0.000    0.000  100.00  -thread2
        2     0.000(  0.0)     0.068    512.5      0.1   0.02   1.0     0.000   0.000    0.000  100.00  -thread3
        2     0.000(  0.0)     0.014    264.7      0.3   0.22   1.0     0.000   0.000    0.000  100.00  -thread4
        2     0.001(  0.0)     0.288    528.3      0.0   0.01   1.0     0.000   0.000    0.000  100.00  -thread5
        2     0.000(  0.0)     0.061    499.3      0.1   0.03   1.0     0.000   0.000    0.000  100.00  -thread6
        2     0.000(  0.0)     0.090    516.1      0.0   0.02   1.0     0.000   0.000    0.000   66.67  -thread7
       16     0.000(  0.0)     0.005    335.9      5.4   0.00   0.0     0.000   0.000    0.000    0.00 squash
        2     0.000(  0.0)     0.005    326.1      5.3   0.00   0.0     0.000   0.000    0.000    0.00  -thread0
        2     0.000(  0.0)     0.005    331.2      5.4   0.00   0.0     0.000   0.000    0.000    0.00  -thread1
        2     0.000(  0.0)     0.005    337.2      5.4   0.00   0.0     0.000   0.000    0.000    0.00  -thread2
        2     0.000(  0.0)     0.005    332.2      5.3   0.00   0.0     0.000   0.000    0.000    0.00  -thread3
        2     0.000(  0.0)     0.005    339.0      5.4   0.00   0.0     0.000   0.000    0.000    0.00  -thread4
        2     0.000(  0.0)     0.005    336.7      5.4   0.00   0.0     0.000   0.000    0.000    0.00  -thread5
        2     0.000(  0.0)     0.005    343.9      5.5   0.00   0.0     0.000   0.000    0.000    0.00  -thread6
        2     0.000(  0.0)     0.005    341.0      5.4   0.00   0.0     0.000   0.000    0.000    0.00  -thread7
        1     0.000(  0.0)     0.052    628.4      0.0   0.00   0.0     0.000   0.000    0.000    0.00 bpnn_initialize
        1     0.000(  0.0)     0.043    141.1      0.0   0.00   0.0     0.000   0.000    0.000    0.00 backprop_face
        1     0.000(  0.0)     0.042    361.3      0.0   0.00   0.0     0.000   0.000    0.000    0.00 setup
       16     0.000(  0.0)     0.002    597.1      2.9   1.72   1.0     0.000   0.000    0.000  100.00 bpnn_adjust_weights$1
        2     0.000(  0.0)     0.002    574.2      2.3   1.42   1.0     0.000   0.000    0.000  100.00  -thread0
        2     0.000(  0.0)     0.002    584.5      3.2   1.96   1.0     0.000   0.000    0.000  100.00  -thread1
        2     0.000(  0.0)     0.002    602.7      3.1   1.86   1.0     0.000   0.000    0.000  100.00  -thread2
        2     0.000(  0.0)     0.002    607.9      3.1   1.82   1.0     0.000   0.000    0.000  100.00  -thread3
        2     0.000(  0.0)     0.002    589.3      2.8   1.73   1.0     0.000   0.000    0.000  100.00  -thread4
        2     0.000(  0.0)     0.002    596.6      2.8   1.68   1.0     0.000   0.000    0.000  100.00  -thread5
        2     0.000(  0.0)     0.002    622.5      3.0   1.75   1.0     0.000   0.000    0.000  100.00  -thread6
        2     0.000(  0.0)     0.002    606.8      2.8   1.68   1.0     0.000   0.000    0.000  100.00  -thread7
        2     0.000(  0.0)     0.009    322.5      0.0   0.00   0.0     0.000   0.000    0.000    0.00 bpnn_layerforward
        1     0.000(  0.0)     0.013    503.1      0.0   0.00   0.0     0.000   0.000    0.000    0.00 bpnn_internal_create
        1     0.000(  0.0)     0.011    304.0      0.0   0.00   0.0     0.000   0.000    0.000    0.00 bpnn_train_kernel
        2     0.000(  0.0)     0.002    730.0      0.0   0.00   0.0     0.000   0.000    0.000    0.00 bpnn_adjust_weights
        1     0.000(  0.0)     0.001    281.2      0.0   0.00   0.0     0.000   0.000    0.000    0.00 main
        1     0.000(  0.0)     0.000   1441.9    292.0  49.19  16.0     0.000   0.000    0.000   50.00 bpnn_output_error
        1     0.000(  0.0)     0.000   2461.5      0.0   0.00   0.0     0.000   0.000    0.000    0.00 bpnn_hidden_error
-------------------------------------------------------------------------------------------------------------
134217807   154.848(100.0)     0.001    932.3      0.1   0.09   1.0     0.262  31.920    0.000   88.12 total
```

Figura 1.7. Resultados do comando FTRACE para a versão original do *Backpropagation*.

vez que a forma de acesso e indexação dos dados agora deve ser feita de forma diferente. Dessa forma, o código abaixo é modificado de modo a substituir a alocação de uma matriz de tamanho *m x n* por uma alocação de um espaço contíguo de memória de tamanho *m x n*.

```
1 BPNN *bpnn_internal_create(n_in, n_hidden, n_out)
2 int n_in, n_hidden, n_out;
3 {
4   ...
5
6   newnet->input_weights = alloc_2d_dbl(n_in + 1, n_hidden
        + 1);
7   newnet->hidden_weights = alloc_2d_dbl(n_hidden + 1,
       n_out + 1);
8
9   newnet->input_prev_weights = alloc_2d_dbl(n_in + 1,
       n_hidden + 1);
10  newnet->hidden_prev_weights = alloc_2d_dbl(n_hidden +
       1, n_out + 1);
11
12  return (newnet);
13 }
```

A otimização é apresentada nos trechos de código abaixo. Na linha 2 do primeiro bloco, adicionamos `long unsigned` na declaração da variável, e das linha 8 à 12 substitui-se a função de alocação bidimensional pela função de alocação unidimensional contígua.

```
1 BPNN *bpnn_internal_create(n_in, n_hidden, n_out)
2 long unsigned int n_in, n_hidden, n_out;
3 {
4   ...
5
6   newnet->input_weights = alloc_1d_dbl( (n_in + 1) * (
       n_hidden + 1));
7   newnet->hidden_weights = alloc_1d_dbl((n_hidden + 1) *
       (n_out + 1));
8
9   newnet->input_prev_weights = alloc_1d_dbl((n_in + 1) *
       (n_hidden + 1));
10  newnet->hidden_prev_weights = alloc_1d_dbl((n_hidden +
       1) * (n_out + 1));
11
12  return (newnet);
13 }
```

```
*---------------------*
  FTRACE ANALYSIS LIST
*---------------------*

Execution Date : Sun Mar 28 04:56:48 2021 -03
Total CPU Time : 0:00'09"394 (9.394 sec.)


EXECUTION TERMINATED BUT NOT IN MAIN PROCEDURE.

 STOPPED  AT setup
 CALLED FROM main


FREQUENCY  EXCLUSIVE       AVER.TIME     MOPS   MFLOPS  V.OP  AVER.    VECTOR L1CACHE CPU PORT VLD LLC PROC.NAME
           TIME[sec](  % )    [msec]                    RATIO V.LEN      TIME    MISS     CONF HIT E.%

       1     6.578( 70.0)  6577.831   2611.8      0.0   0.00 138.5     0.000   0.263    0.000  100.00 bpnn_read
      16     1.638( 17.4)   102.395  10404.4   2662.5  75.59  16.0     1.638   0.000    0.000   33.35 bpnn_adjust_weights$1
       2     0.205(  2.2)   102.396  10404.3   2662.5  75.59  16.0     0.205   0.000    0.000   33.35  -thread0
       2     0.205(  2.2)   102.395  10404.4   2662.5  75.59  16.0     0.205   0.000    0.000   33.35  -thread1
       2     0.205(  2.2)   102.395  10404.4   2662.5  75.59  16.0     0.205   0.000    0.000   33.35  -thread2
       2     0.205(  2.2)   102.395  10404.4   2662.5  75.59  16.0     0.205   0.000    0.000   33.35  -thread3
       2     0.205(  2.2)   102.395  10404.4   2662.5  75.59  16.0     0.205   0.000    0.000   33.35  -thread4
       2     0.205(  2.2)   102.395  10404.4   2662.5  75.59  16.0     0.205   0.000    0.000   33.35  -thread5
       2     0.205(  2.2)   102.395  10404.4   2662.5  75.59  16.0     0.205   0.000    0.000   33.35  -thread6
       2     0.205(  2.2)   102.395  10404.4   2662.5  75.59  16.0     0.205   0.000    0.000   33.35  -thread7
       2     1.025( 10.9)   512.373   5042.6      0.0  44.16  17.0     1.025   0.000    0.000    0.00 bpnn_zero_weights
      16     0.152(  1.6)     9.528  28483.8  14086.6  98.91 256.0     0.150   0.000    0.000   69.87 bpnn_layerforward$1
       2     0.020(  0.2)    10.052  27026.0  13351.7  98.81 256.0     0.019   0.000    0.000    2.92  -thread0
       2     0.019(  0.2)     9.502  28559.4  14124.7  98.91 256.0     0.019   0.000    0.000   98.88  -thread1
       2     0.019(  0.2)     9.426  28787.5  14239.7  98.93 256.0     0.019   0.000    0.000   26.30  -thread2
       2     0.019(  0.2)     9.400  28865.9  14279.2  98.94 256.0     0.019   0.000    0.000   99.98  -thread3
       2     0.019(  0.2)     9.449  28717.9  14204.6  98.93 256.0     0.019   0.000    0.000   49.33  -thread4
       2     0.019(  0.2)     9.587  28312.9  14000.5  98.90 256.0     0.019   0.000    0.000   99.34  -thread5
       2     0.019(  0.2)     9.406  28846.3  14269.4  98.93 256.0     0.019   0.000    0.000   82.35  -thread6
       2     0.019(  0.2)     9.403  28856.0  14274.3  98.93 256.0     0.019   0.000    0.000   99.82  -thread7
      17     0.000(  0.0)     0.007    248.2      4.1   0.00   0.0     0.000   0.000    0.000    0.00 squash
       3     0.000(  0.0)     0.005    239.4      5.2   0.00   0.0     0.000   0.000    0.000    0.00  -thread0
       2     0.000(  0.0)     0.007    241.4      3.8   0.00   0.0     0.000   0.000    0.000    0.00  -thread1
       2     0.000(  0.0)     0.007    246.0      3.9   0.00   0.0     0.000   0.000    0.000    0.00  -thread2
       2     0.000(  0.0)     0.007    247.4      3.9   0.00   0.0     0.000   0.000    0.000    0.00  -thread3
       2     0.000(  0.0)     0.007    248.5      3.9   0.00   0.0     0.000   0.000    0.000    0.00  -thread4
       2     0.000(  0.0)     0.007    253.0      4.0   0.00   0.0     0.000   0.000    0.000    0.00  -thread5
       2     0.000(  0.0)     0.007    252.1      3.9   0.00   0.0     0.000   0.000    0.000    0.00  -thread6
       2     0.000(  0.0)     0.007    259.4      4.0   0.00   0.0     0.000   0.000    0.000    0.00  -thread7
       1     0.000(  0.0)     0.052    641.0      0.0   0.00   0.0     0.000   0.000    0.000    0.00 bpnn_initialize
       1     0.000(  0.0)     0.042    374.2      0.0   0.00   0.0     0.000   0.000    0.000    0.00 setup
       1     0.000(  0.0)     0.038    150.1      0.0   0.00   0.0     0.000   0.000    0.000    0.00 backprop_face
      10     0.000(  0.0)     0.002    285.5      0.0   0.00   0.0     0.000   0.000    0.000    0.00 alloc_1d_dbl
       2     0.000(  0.0)     0.009    355.6      0.0   0.00   0.0     0.000   0.000    0.000    0.00 bpnn_layerforward
       1     0.000(  0.0)     0.017    118.9      0.0   0.00   0.0     0.000   0.000    0.000    0.00 bpnn_free
       1     0.000(  0.0)     0.013    513.3      0.0   0.00   0.0     0.000   0.000    0.000    0.00 bpnn_internal_create
       1     0.000(  0.0)     0.010    311.0      0.0   0.00   0.0     0.000   0.000    0.000    0.00 bpnn_train_kernel
       1     0.000(  0.0)     0.009    377.0     15.7  10.18   3.5     0.000   0.000    0.000  100.00 bpnn_hidden_error
       2     0.000(  0.0)     0.003    536.8      0.0   0.00   0.0     0.000   0.000    0.000    0.00 bpnn_adjust_weights
       1     0.000(  0.0)     0.001    336.4      0.0   0.00   0.0     0.000   0.000    0.000    0.00 main
       1     0.000(  0.0)     0.000    789.2     18.4   5.65   1.0     0.000   0.000    0.000   50.00 bpnn_output_error
-------------------------------------------------------------------------------------------------------------------
      75     9.394(100.0)   125.249   4655.8    693.0  44.50  19.6     2.813   0.263    0.000   47.96 total
```

Figura 1.8. Resultados do FTRACE para a otimização de alocação de memória contígua.

Por vezes, o tamanho `m x n` ultrapassa a capacidade do tipo de dados *int*. Dessa forma, faz-se necessário alterar os tipos das variáveis `m` e `n` para `long unsigned` nas demais funções. Além disso, por agora termos um bloco contíguo de dados ao invés de uma matriz bidimensional, faz-se necessário também alterar a forma como esses dados são acessados. Já não se utiliza mais o comum `matriz[i][j]`. Agora é necessário calcular o `findex` que permite realizar o acesso correto aos dados armazenados continuamente na memória. O trecho de código abaixo exemplifica essas duas alterações, com a utilização de `long unsigned` nas linhas 3 e 5, e com o cálculo do `findex` e sua utilização nas linhas 7 e 9, respectivamente.

```
1 bpnn_zero_weights(w, m, n)
2 float *w;
3 long unsigned int m, n;
4 {
5   long unsigned int i, j, findex;
6   for (i = 0; i <= m; i++) {
7     findex = i*(n+1);
8     for (j = 0; j <= n; j++) {
9       w[findex + j] = 0.0;
10    }
11  }
12 }
```

Verificando os resultados dessa otimização através do novo `ftrace.out`, representado na Figura 1.8, temos que o tempo de execução diminui de 154 segundos para 9 segundos, e a função `alloc_1d_dbl` agora corresponde a uma parcela irrisória do tempo de execução.

A última otimização aplicada é relacionada à função `bppn_read`, que, após a primeira otimização, se tornou responsável por 70% do tempo de execução, como visto na Figura 1.8. Nessa seção do código é realizada a leitura dos pesos da Rede Neural a partir de um arquivo texto. No trecho de código abaixo, temos a implementação da função `fastcopy` (bastante utilizada pela função `bppn_read` para copiar os dados lidos de uma estrutura de dados para outra) e as operações de leitura realizadas pela função `bpnn_read` em suas versões originais.

```
1 #define fastcopy(to,from,len)
2 {
3   register char *_to,*_from;
4   register int _i,_l;
5   _to = (char *)(to);
6   _from = (char *)(from);
7   _l = (len);
8   for (_i = 0; _i < _l; _i++) *_to++ = *_from++;
9 }
10
11 BPNN *bpnn_read(filename)
12 char *filename;
```

```
13 {
14   ...
15   int fd, n1, n2, n3, i, j, memcnt;
16
17   if ((fd = open(filename, 0, 0644)) == -1) {
18     return (NULL);
19   }
20
21   printf("Reading '%s'\n", filename); //fflush(stdout);
22
23   // leituras com a funcao read
24   read(fd, (char *) &n1, sizeof(int));
25   read(fd, (char *) &n2, sizeof(int));
26   read(fd, (char *) &n3, sizeof(int));
27
28   ...
29
30   mem = (char *) malloc ((unsigned) ((n1+1) * (n2+1) *
       sizeof(float)));
31   // leituras com a funcao read e loop complexo com a
       funcao fastcopy
32   read(fd, mem, (n1+1) * (n2+1) * sizeof(float));
33   for (i = 0; i <= n1; i++) {
34     for (j = 0; j <= n2; j++) {
35       fastcopy(&(new->input_weights[i][j]), &mem[memcnt],
           sizeof(float));
36       memcnt += sizeof(float);
37     }
38   }
39   free(mem);
40
41   printf("Done\nReading hidden weights..."); //fflush(
       stdout);
42
43   memcnt = 0;
44   mem = (char *) malloc ((unsigned) ((n2+1) * (n3+1) *
       sizeof(float)));
45   // leituras com a funcao read e loop complexo com a
       funcao fastcopy
46   read(fd, mem, (n2+1) * (n3+1) * sizeof(float));
47   for (i = 0; i <= n2; i++) {
48     for (j = 0; j <= n3; j++) {
49       fastcopy(&(new->hidden_weights[i][j]), &mem[memcnt],
           sizeof(float));
50       memcnt += sizeof(float);
51     }
```

```
52   }
53   free(mem);
54   close(fd);
55
56   ...
57 }
```

Como otimização buscamos portanto modificar a leitura do arquivo, substituindo as chamadas à função `read` (da biblioteca `unistd.h`) pela função `fread` (da biblioteca `stdio.h`), e removemos as chamadas à função `fastcopy`. No trecho de código abaixo pode-se ver as modificações realizadas.

```
1  BPNN *bpnn_read(filename)
2  char *filename;
3  {
4    ...
5    FILE *pFile;
6    pFile = fopen( filename, "rb" );
7
8    if (pFile == NULL) {
9      return (NULL);
10   }
11
12   printf("Reading '%s'\n", filename); //fflush(stdout);
13
14   // substituicao das chamadas a funcao read por chamadas
         a funcao fread
15   fread((char *) &n1, sizeof(char), sizeof(long unsigned
       int), pFile);
16   fread((char *) &n2, sizeof(char), sizeof(long unsigned
       int), pFile);
17   fread((char *) &n3, sizeof(char), sizeof(long unsigned
       int), pFile);
18
19   ...
20
21
22   // loop de leitura com fastcopy substituido
23   long unsigned int totalmem = (n1+1) * (n2+1) * sizeof(
       float);
24   fread((char*)new->input_weights, sizeof(char), totalmem
       , pFile);
25   float * reader = mem;
26
27   ...
28
29   // loop de leitura com fastcopy substituido
```

```
30   totalmem = (n2+1) * (n3+1) * sizeof(float);
31   fread((char*) new->hidden_weights, sizeof(char),
        totalmem, pFile);
32   fclose(pFile);
33
34   ...
35 }
```

Analisando o arquivo de saída `ftrace.out` da Figura 1.9, podemos ver que o tempo de execução diminui em quatro vezes, de 9 segundos para 2 segundos.

```
*---------------------*
  FTRACE ANALYSIS LIST
*---------------------*

Execution Date : Sun Mar 28 04:54:59 2021 -03
Total CPU Time : 0:00'02"162 (2.162 sec.)


EXECUTION TERMINATED BUT NOT IN MAIN PROCEDURE.

 STOPPED  AT setup
 CALLED FROM main


FREQUENCY  EXCLUSIVE       AVER.TIME     MOPS   MFLOPS  V.OP  AVER.    VECTOR L1CACHE CPU PORT VLD LLC PROC.NAME
          TIME[sec](  % )    [msec]                      RATIO V.LEN      TIME    MISS     CONF HIT E.% 

      16     1.639( 75.8)   102.438  10400.1   2661.4  75.59  16.0     1.639   0.000    0.000   33.35 bpnn_adjust_weights$1
       2     0.205(  9.5)   102.439  10400.0   2661.4  75.59  16.0     0.205   0.000    0.000   33.35  -thread0
       2     0.205(  9.5)   102.437  10400.1   2661.4  75.59  16.0     0.205   0.000    0.000   33.35  -thread1
       2     0.205(  9.5)   102.437  10400.1   2661.4  75.59  16.0     0.205   0.000    0.000   33.35  -thread2
       2     0.205(  9.5)   102.438  10400.0   2661.4  75.59  16.0     0.205   0.000    0.000   33.35  -thread3
       2     0.205(  9.5)   102.438  10400.1   2661.4  75.59  16.0     0.205   0.000    0.000   33.35  -thread4
       2     0.205(  9.5)   102.438  10400.1   2661.4  75.59  16.0     0.205   0.000    0.000   33.35  -thread5
       2     0.205(  9.5)   102.437  10400.1   2661.4  75.59  16.0     0.205   0.000    0.000   33.35  -thread6
       2     0.205(  9.5)   102.438  10400.0   2661.4  75.59  16.0     0.205   0.000    0.000   33.35  -thread7
       2     0.369( 17.1)   184.693   8765.9      0.0  70.47  17.0     0.369   0.000    0.000    0.00 bpnn_zero_weights
      16     0.153(  7.1)     9.573  28354.0  14021.2  98.90 256.0     0.150   0.000    0.000   70.26 bpnn_layerforward$1
       2     0.021(  1.0)    10.286  26424.0  13048.2  98.76 256.0     0.019   0.000    0.000   55.43  -thread0
       2     0.019(  0.9)     9.472  28649.9  14170.4  98.92 256.0     0.019   0.000    0.000   38.17  -thread1
       2     0.019(  0.9)     9.401  28862.6  14277.6  98.93 256.0     0.019   0.000    0.000   99.52  -thread2
       2     0.019(  0.9)     9.498  28571.5  14130.9  98.92 256.0     0.019   0.000    0.000   80.13  -thread3
       2     0.019(  0.9)     9.532  28471.0  14080.2  98.91 256.0     0.019   0.000    0.000   98.41  -thread4
       2     0.019(  0.9)     9.522  28501.8  14095.7  98.91 256.0     0.019   0.000    0.000    0.03  -thread5
       2     0.019(  0.9)     9.416  28817.2  14254.8  98.93 256.0     0.019   0.000    0.000   99.98  -thread6
       2     0.019(  0.9)     9.453  28704.8  14198.1  98.92 256.0     0.019   0.000    0.000   90.40  -thread7
       1     0.000(  0.0)     0.171    278.7      0.0   2.33 138.5     0.000   0.000    0.000  100.00 bpnn_read
      17     0.000(  0.0)     0.007    239.6      4.0   0.00   0.0     0.000   0.000    0.000    0.00 squash
       3     0.000(  0.0)     0.005    239.1      5.2   0.00   0.0     0.000   0.000    0.000    0.00  -thread0
       2     0.000(  0.0)     0.007    231.2      3.7   0.00   0.0     0.000   0.000    0.000    0.00  -thread1
       2     0.000(  0.0)     0.007    239.7      3.8   0.00   0.0     0.000   0.000    0.000    0.00  -thread2
       2     0.000(  0.0)     0.007    233.7      3.7   0.00   0.0     0.000   0.000    0.000    0.00  -thread3
       2     0.000(  0.0)     0.007    240.9      3.8   0.00   0.0     0.000   0.000    0.000    0.00  -thread4
       2     0.000(  0.0)     0.007    245.0      3.8   0.00   0.0     0.000   0.000    0.000    0.00  -thread5
       2     0.000(  0.0)     0.007    242.5      3.8   0.00   0.0     0.000   0.000    0.000    0.00  -thread6
       2     0.000(  0.0)     0.007    245.3      3.8   0.00   0.0     0.000   0.000    0.000    0.00  -thread7
       1     0.000(  0.0)     0.053    628.2      0.0   0.00   0.0     0.000   0.000    0.000    0.00 bpnn_initialize
       1     0.000(  0.0)     0.042    374.2      0.0   0.00   0.0     0.000   0.000    0.000    0.00 setup
       1     0.000(  0.0)     0.039    148.8      0.0   0.00   0.0     0.000   0.000    0.000    0.00 backprop_face
       1     0.000(  0.0)     0.032    174.6      0.0   0.00   0.0     0.000   0.000    0.000    0.00 bpnn_free
      10     0.000(  0.0)     0.002    284.4      0.0   0.00   0.0     0.000   0.000    0.000    0.00 alloc_1d_dbl
       2     0.000(  0.0)     0.009    349.7      0.0   0.00   0.0     0.000   0.000    0.000    0.00 bpnn_layerforward
       1     0.000(  0.0)     0.013    514.4      0.0   0.00   0.0     0.000   0.000    0.000    0.00 bpnn_internal_create
       1     0.000(  0.0)     0.010    312.4      0.0   0.00   0.0     0.000   0.000    0.000    0.00 bpnn_train_kernel
       1     0.000(  0.0)     0.009    385.8     16.1  10.18   3.5     0.000   0.000    0.000  100.00 bpnn_hidden_error
       2     0.000(  0.0)     0.003    546.5      0.0   0.00   0.0     0.000   0.000    0.000    0.00 bpnn_adjust_weights
       1     0.000(  0.0)     0.001    325.9      0.0   0.00   0.0     0.000   0.000    0.000    0.00 main
       1     0.000(  0.0)     0.000    671.6     15.6   5.65   1.0     0.000   0.000    0.000   50.00 bpnn_output_error
-------------------------------------------------------------------------------------------------------------
      75     2.162(100.0)    28.828  11390.2   3010.8  79.03  19.6     2.158   0.001    0.000   48.12 total
```

Figura 1.9. Resultados do FTRACE para a otimização de leitura

1.8.2. Intel Cascade Lake

Todas as otimizações propostas acima são válidas também para a microarquitetura Intel Cascade Lake. O que faremos nesta seção é demonstrar como podemos perceber quais são as modificações necessárias por meio de contadores de *hardware* e como atestar o ganho de desempenho após elas. Uma vez que tem-se uma ampla gama de contadores disponíveis na arquitetura da Intel, escolher por onde iniciar a análise de desempenho pode ser uma tarefa complicada. Um ponto de início pode ser a verificação do número de *cache misses* nos vários níveis de *cache* (FOG, 2012; INTEL, 2019), uma vez que encontrar o dado o mais próximo do processador possível é importante para se evitar grandes latências da hierarquia de memória (como mencionado na Seção 1.3) (PATTERSON; HENNESSY, 2004). Além disso, verificar o número de instruções de predição de desvio que foram feitas de forma incorreta (*branch mispredictions*) (FOG, 2012; INTEL, 2019) também é um fator importante, já que uma predição de salto feita da forma incorreta pode acarretar na necessidade de se limpar todas as instruções presentes no *pipeline* (PATTERSON; HENNESSY, 2004).

O código abaixo representa a execução da aplicação *Backpropagation* em sua versão original no processador Intel Xeon Gold 6226 utilizando a ferramenta Linux perf e com tamanho de entrada equivalente a 2^{26}. Os contadores de *hardware* escolhidos são o *LLC-loads* e o *LLC-load-misses*, que representam respectivamente o número total de instruções de requisição de dados que chegam à LLC e a parte dessas requisições que deram *miss* nesse nível de *cache*. Logo abaixo é possível ver a saída da execução com a ferramenta perf, com a saída da execução da própria aplicação, os números absolutos de acesso à LLC e a porcentagem desses acessos que resultaram em *miss*, bem como o tempo de execução da aplicação.

```
1 perf stat -e LLC-loads,LLC-load-misses ./backprop
    67108864
2
3 Random number generator seed: 7
4 Input layer size : 67108864
5 Starting training kernel
6 Performing CPU computation
7 Training done
8
9 Performance counter stats for './backprop 67108864':
10
11 2,984,760,563  LLC-loads
12  961,573,754  LLC-load-misses # 32.22% of all LL-cache
      hits
13
14  39.501156910 seconds time elapsed
```

Além disso, a execução é repetida usando contadores semelhantes referentes à predição de desvio, como visto no código abaixo:

```
perf stat -e branch-instructions,branch-misses ./backprop 67108864
Random number generator seed: 7
Input layer size : 67108864
Starting training kernel
Performing CPU computation
Training done

Performance counter stats for './backprop 67108864':

 43,232,691,964 branch-instructions
    167,438,939  branch-misses # 0.39% of all branches

   39.568886150 seconds time elapsed
```

Pode-se notar que a porcentagem de *branch mispredictions* é praticamente irrisória, o que não pode ser afirmado com relação aos *cache misses*, que chegam a 32,22% das requisições de leitura que chegam à LLC. Para analisar melhor o acesso à memória, pode-se utilizar contadores referentes a níveis de *cache* mais próximos do processador. Por fins de simplificação, o código abaixo mostra diretamente as informações obtidas da *cache* L1, novamente em termos de requisições totais e porcentagem que resulta em *miss*. Note que no que, diz respeito à *cache* L1, faz-se necessário indicar que quer-se informações referentes apenas à *cache* de dados (*dcache*), uma vez que o objetivo é justamente analisar a eficiência do acesso aos dados. Para referir-se à *cache* instruções faz-se uso do termo *icache*.

```
perf stat -e L1-dcache-loads,L1-dcache-load-misses ./backprop 67108864
...

Performance counter stats for './backprop 67108864':

54,733,941,264 L1-dcache-loads
 7,941,597,753 L1-dcache-load-misses # 14.51% of all L1-dcache hits

  38.764941014 seconds time elapsed
```

Novamente é possível verificar que a grande porcentagem de *misses* na L1 (14,51%) abre espaço para otimizações. O acesso a dados que não estão contíguos em memória prejudica qualquer aplicação devido à falta de localidade espacial, como já mencionado anteriormente. Dessa forma, as alterações na alocação e na forma de acesso aos dados também se fazem importantes, bem como a aplicação de `long unsigned` no lugar de `int`. Após aplicar tais modificações como proposto na Seção 1.8.1, tem-se tais resultados na análise de desempenho da aplicação:

```
1 perf stat -e L1-dcache-loads,L1-dcache-load-misses ./
    backprop 67108864
2 ...
3
4 Performance counter stats for './backprop 67108864':
5
6 47,274,433,208 L1-dcache-loads
7  2,790,105,769 L1-dcache-load-misses # 5.90% of all L1-
     dcache hits
8
9  24.095061856 seconds time elapsed
```

Alterando a forma como a aplicação aloca e acessa os dados permitiu uma redução de 64% no número de misses relativos na *cache* L1 de dados, e reduziu em 37% o tempo de execução da aplicação. A próxima otimização possível é a substituição da função `fastcopy` utilizada pela função `bpnn_read`, alterando a leitura do arquivo de pesos com a função `fread` no lugar da função `read`, assim como feito na Seção 1.8.1. Abaixo pode-se observar os resultados de acesso à *cache* L1 de dados e o novo tempo de execução da aplicação.

```
1 perf stat -e L1-dcache-loads,L1-dcache-load-misses ./
    backprop 67108864
2 ...
3
4 Performance counter stats for './backprop 67108864':
5
6 22,440,730,866 L1-dcache-loads
7  2,417,265,599 L1-dcache-load-misses # 10.77% of all L1-
     dcache hits
8  14.668708825 seconds time elapsed
```

Nota-se que, apesar de a porcentagem de *misses* relativos ter aumentado em comparação ao resultado anterior, a quantidade total de requisições de leitura que chegam à L1 diminui para menos da metade por conta das modificações feitas no código. Dessa forma, foi possível reduzir ainda mais o tempo de execução da aplicação, que foi de 39 segundos em sua versão original para menos de 15 segundos na versão com algumas otimizações.

1.9. Conclusão

Nesse capítulo de minicurso, apresentamos conceitos de arquitetura de computadores, hierarquia de memória, vetorização e inteligência artificial. Utilizou-se exemplos e técnicas de otimização que podem ser aplicadas em ambas as arquiteturas da Intel e da NEC empregadas sobre uma implementação de retro-propagação da área de inteligência artificial. Com essas técnicas demonstramos ganhos de desempenho consideráveis, aprimorando em até 71 vezes o tempo de execução da aplicação no processador vetorial SX-Aurora, um ganho possível graças à possibilidade de vetorização obtida com as alteralções.

Já no processador da Intel, um Xeon Gold 6226, foi possível reduzir o tempo de execução em 2,6 vezes. Diversas outras otimizações poderiam ser aplicadas para apri-

morar a capacidade de se usar as instruções vetoriais do processador. Uma vez que este é um minicurso de nível básico, nos detemos a explicar as funcionalidades básicas dos contadores em ambas as microarquiteturas.

Referências

A, i. et al. Survey on artificial intelligence. *International Journal of Computer Sciences and Engineering*, v. 7, p. 1778–1790, 05 2019. páginas

ADHIANTO, L. et al. Hpctoolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, v. 22, 01 2009. páginas

AGGARWAL, C. C. et al. Neural networks and deep learning. *Springer*, Springer, v. 10, p. 978–3, 2018. páginas

BAER, J.-L.; CHEN, T.-F. An effective on-chip preloading scheme to reduce data access penalty. In: *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*. New York, NY, USA: Association for Computing Machinery, 1991. (Supercomputing '91), p. 176–186. ISBN 0897914597. Disponível em: <https://doi.org/10.1145/125826.125932>. páginas

BAKHSHALIPOUR, M. et al. Bingo spatial data prefetcher. In: LOURI, A.; VENKATARAMANI, G.; GRATZ, P. (Ed.). *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. [S.l.], 2019. p. 399–411. páginas

BHATIA, E. et al. Perceptron-based prefetch filtering. In: *Proceedings of the 46th International Symposium on Computer Architecture*. New York, NY, USA: Association for Computing Machinery, 2019. (ISCA '19), p. 1–13. ISBN 9781450366694. Disponível em: <https://doi.org/10.1145/3307650.3322207>. páginas

BLAKE, G.; DRESLINSKI, R. G.; MUDGE, T. A survey of multicore processors. *IEEE Signal Processing Magazine*, IEEE, v. 26, n. 6, p. 26–37, 2009. páginas

BOKADE, S.; DAKHOLE, P. Cla based 32-bit signed pipelined multiplier. In: IEEE. *2016 international conference on communication and signal processing (ICCSP)*. [S.l.], 2016. p. 0849–0852. páginas

CASELLA, G.; BERGER, R. L. *Statistical inference*. [S.l.]: Cengage Learning, 2021. páginas

CHE, S. et al. Rodinia: A benchmark suite for heterogeneous computing. In: IEEE. *2009 IEEE international symposium on workload characterization (IISWC)*. [S.l.], 2009. p. 44–54. páginas

CHEN, T.-F.; BAER, J.-L. Effective hardware-based data prefetching for high-performance processors. *IEEE transactions on computers*, IEEE, v. 44, n. 5, p. 609–623, 1995. páginas

CUTRESS, I. *The AMD Zen and Ryzen 7 Review: A Deep Dive on 1800X, 1700X and 1700*. 2017. Disponível em: <https://www.anandtech.com/show/11170/the-amd-zen-and-ryzen-7-review-a-deep-dive-on-1800x-1700x-and-1700>. páginas

FLOYD, T. L. *Digital Fundamentals, 10/e*. [S.l.]: Pearson Education India, 2010. páginas

FOG, A. The microarchitecture of intel, amd and via cpus: An optimization guide for assembly programmers and compiler makers. *Copenhagen University College of Engineering*, p. 02–29, 2012. páginas

GADEPALLY, V. et al. Ai enabling technologies: A survey. *arXiv preprint arXiv:1905.03592*, 2019. páginas

GERNDT, M.; FÜRLINGER, K.; KEREKU, E. Periscope: Advanced techniques for performance analysis. In: *PARCO*. [S.l.: s.n.], 2005. páginas

GIRELLI, V. S. et al. Investigating memory prefetcher performance over parallel applications: From real to simulated. *Concurrency and Computation: Practice and Experience*, n/a, n. n/a, p. e6207. Disponível em: <https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.6207>. páginas

GOMEZ, A. N. et al. The reversible residual network: Backpropagation without storing activations. *arXiv preprint arXiv:1707.04585*, 2017. páginas

Guttman, D. et al. Performance and energy evaluation of data prefetching on intel xeon phi. In: *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. [S.l.: s.n.], 2015. p. 288–297. páginas

HENNESSY, J. et al. Mips: A microprocessor architecture. *ACM SIGMICRO Newsletter*, ACM New York, NY, USA, v. 13, n. 4, p. 17–22, 1982. páginas

HENNESSY, J. L.; PATTERSON, D. A. *Computer Architecture, Sixth Edition: A Quantitative Approach*. 6th. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017. ISBN 0128119055. páginas

INTEL. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. 2019. <https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf>. [Accesed in: 16 Jan. 2020]. páginas

ISEN, C.; JOHN, L. K.; JOHN, E. A tale of two processors: Revisiting the risc-cisc debate. In: SPRINGER. *Spec benchmark workshop*. [S.l.], 2009. p. 57–76. páginas

JACOB, B.; WANG, D.; NG, S. *Memory systems: cache, DRAM, disk*. [S.l.]: Morgan Kaufmann, 2010. páginas

KANG, H.; WONG, J. L. To hardware prefetch or not to prefetch? a virtualized environment study and core binding approach. In: *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages*

and Operating Systems. New York, NY, USA: Association for Computing Machinery, 2013. (ASPLOS '13), p. 357–368. ISBN 9781450318709. Disponível em: <https://doi.org/10.1145/2451116.2451155>. páginas

KNÜPFER, A. et al. The vampir performance analysis tool-set. In: *Parallel Tools Workshop*. [S.l.: s.n.], 2008. páginas

KOMATSU, K. et al. Performance evaluation of a vector supercomputer sx-aurora tsubasa. In: IEEE. *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. [S.l.], 2018. p. 685–696. páginas

KSHEMKALYANI, A. *Vector Processors*. 2012. <https://www.cs.uic.edu/~ajayk/c566/VectorProcessors.pdf>. Accessed: 2021–02-21. páginas

LE, H. Q. et al. Ibm power6 microarchitecture. *IBM Journal of Research and Development*, IBM, v. 51, n. 6, p. 639–662, 2007. páginas

Liao, S. et al. Machine learning-based prefetch optimization for data center applications. In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. [S.l.: s.n.], 2009. p. 1–10. ISSN 2167-4337. páginas

MARR, D. T. et al. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, v. 6, n. 1, 2002. páginas

MEY, D. et al. Score-p: A unified performance measurement system for petascale applications. In: ____. [S.l.: s.n.], 2012. p. 85–97. ISBN 9783642240249. páginas

MORGAN, T. P. *Drilling Xeon Skylake Architecture*. 2017. Disponível em: <https://www.nextplatform.com/2017/08/04/drilling-xeon-skylake-architecture/>. páginas

NEC. *How to Use C/C++ Compiler for Vector Engine*. 2020. <https://www.hpc.nec/api/v1/forum/file/download?id=pgNh9b>. Acessado em: 08/2020. páginas

NEC. *SX-Aurora TSUBASA A100-1 series user's guide*. 2020. <https://www.hpc.nec/documents/guide/pdfs/A100-1_series_users_guide.pdf>. Acessado em: 08/2020. páginas

PATTERSON, D. A.; HENNESSY, J. L. *Computer Organization and Design (4nd Ed.): The Hardware/Software Interface*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004. ISBN 1558604286. páginas

Peled, L. et al. Semantic locality and context-based prefetching using reinforcement learning. In: *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. [S.l.: s.n.], 2015. p. 285–297. páginas

PEREZ, A. F. et al. *Lower Numerical Precision Deep Learning Inference and Training*. 2018. <https://software.intel.com/content/www/us/en/develop/articles/lower-numerical-precision-deep-learning-inference-and-training.html>. Accessed: 2021–02-28. páginas

SHENDE, S. S.; MALONY, A. D. The tau parallel performance system. *Int. J. High Perform. Comput. Appl.*, Sage Publications, Inc., USA, v. 20, n. 2, p. 287–311, maio 2006. ISSN 1094-3420. Disponível em: <https://doi.org/10.1177/1094342006064482>. páginas

THORNTON, J. E. The cdc 6600 project. *Annals of the History of Computing*, IEEE, v. 2, n. 4, p. 338–348, 1980. páginas

TULLSEN, D. M.; EGGERS, S. J.; LEVY, H. M. Simultaneous multithreading: Maximizing on-chip parallelism. In: *Proceedings of the 22nd annual international symposium on Computer architecture*. [S.l.: s.n.], 1995. p. 392–403. páginas

WULF, W.; MCKEE, S. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, v. 23, 01 1996. páginas

YEH, T.-Y.; PATT, Y. N. Two-level adaptive training branch prediction. In: *Proceedings of the 24th annual international symposium on Microarchitecture*. [S.l.: s.n.], 1991. p. 51–61. páginas

Zangeneh, S. et al. Branchnet: A convolutional neural network to predict hard-to-predict branches. In: *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. [S.l.: s.n.], 2020. p. 118–130. páginas

Zhang, L. et al. A dynamic branch predictor based on parallel structure of srnn. *IEEE Access*, v. 8, p. 86230–86237, 2020. páginas