O Impacto da Evolução das Memórias nas Métricas para Tempo de Execução, Complexidade de Algoritmos e Segurança

autor1[1], **autor2**[1], **autor3**[1]

[1] Instituição dos autores
CEP: – Cidade – Estado – Brazil

blind@blind.br

Resumo. *O custo em consumo de energia de uma operação em memória RAM pode ser até três ordens de grandeza maior do que uma operação aritmética. Este trabalho apresenta algumas experiências que mostram o impacto da necessidade de uma visão mais ampla dos sistemas de memórias atuais. Destaca a importância da introdução de conceitos para motivar os alunos e ao mesmo tempo explicar os efeitos colaterais que não podem ser compreendidos devido a adoção das abstrações tradicionais. Estes fatos levam a novos desafios para projeto de algoritmos e avaliação da complexidade em tempo e em consumo de energia, além de questões de segurança. Iremos destacar o impacto da localidade dos dados e como medí-los, experiências didáticas de algumas instituições e novas arquiteturas dos sistemas de memórias da era Big Data.*

1. Introdução

O ensino tradicional de complexidade de algoritmos é focado na ordem de crescimento do número de operações. Por exemplo, a complexidade da multiplicação de matrizes $n \times n$ é $O(n^3)$ adições/multiplicações escalares. Entretanto, o custo em tempo, o consumo de energia e a eficiência energética (número de operações por Watt) das operações com a memória dominam o custo das operações aritméticas na implementação de vários algoritmos e arquiteturas (processadores *multicores*, processadores gráficos (GPU), etc.).

Este artigo[1] ilustra alguns outros pontos que motivam um estudo das estruturas de memória com uma visão ampliada no ensino de complexidade de algoritmos. Serão apresentados tópicos e exemplos de aulas que demonstram que com poucas horas adicionais é possível incluir tópicos nos cursos de programação e arquitetura para motivar os alunos a buscarem mais informações. Um exemplo são as duas falhas *spectre* e *meltdown* descobertas apenas recentemente mas que estão presentes na maioria dos processadores fabricados nas últimas duas décadas, onde uma decisão de projeto para buscar desempenho prejudicou a segurança do sistema. Outros pontos mostram a importância de entender as arquiteturas da última década na era do *Big Data*, onde a forma de implementar suas estruturas de dados e algoritmos tem grande impacto no desempenho [Šidlauskas and Jensen 2014]. A memória RAM (*Random Access Memory*) não tem mais um tempo de acesso uniforme para qualquer posição de memória. O desempenho das RAMs atuais depende das últimas

[1] Observação importante, através de uma página anônima [Anônimo] disponibilizamos os apontadores para as páginas de aulas de algumas universidades, assim como trechos de códigos utilizados.

posições acessadas e do padrão de acesso dos outros processos que executam concorrentemente [Moscibroda and Mutlu 2007] devido a complexa estrutura interna das memórias DDR.

Este artigo apresenta exemplos que estão disponíveis na WEB para serem validados pelos professores/alunos assim como sugestões de aulas disponíveis em vídeo sobre o assunto. O artigo tem a seguinte estrutura. A seção 2 apresenta o exemplo clássico da multiplicação de matrizes e o impacto da implementação sem alterar a complexidade $O(n^3)$ das métricas tradicionais. A seção 3 mostra o impacto das memórias caches na multiplicação de matrizes, algumas ferramentas bem acessíveis para realizar os experimentos e um exemplo de aula de complexidade sobre o assunto. A seção 4 aponta rapidamente alguns aspectos dos compiladores atuais para interpretar os dados. Outros exemplos de aulas para alunos de primeiro ano da Universidade de Zurich relatando quatro mistérios das memórias, incluindo as falhas *spectre* e *meltdown*, serão apresentados na seção 5. A seção 6 apresenta as novas arquiteturas de memória que buscam eficiência energética. Finalmente, na seção 7, são feitas considerações finais sobre o tema e sugestões de futuros trabalhos.

2. Multiplicação de Matrizes

A multiplicação é um exemplo clássico para ensinar o comando *for*, também usado como exemplo na análise de complexidade laços aninhados. A figura 1(a) mostra o código na linguagem C da versão clássica com três laços aninhados e complexidade de operações $O(n^3)$. Considerando que a hierarquia de memória dos computadores possui memórias cache a mais de três décadas, este código é ineficiente devido ao grande número de falhas no acesso as caches, pois percorre a matrix B mudando a linha j fixando a coluna i ($B[j][i]$). As matrizes são organizadas por linha na memória. Este padrão de acesso polui a cache ao buscar uma linha ou parte dela para acessar apenas um elemento. A figura 1(b) apresenta a versão com ladrilhos (*tiled* em inglês), que divide as matrizes em pequenos blocos que são buscados para cache gerando reuso e reduzindo as falhas. Apesar de executar mais operações de adição e o mesmo número de multiplicações, a versão com ladrilhos possui localidade de acesso e pode reduzir tempo de execução. Este é um exemplo simples, onde uma análise superficial da complexidade considerando só as operações não consegue explicar a diferença de tempo de execução que mostraremos na próxima seção.

Transformações nos laços para gerar reuso dos dados é uma técnica usada a mais de três décadas [Wolf and Lam 1991]. Para leitores interessados em uma análise detalhada das falhas em cache para multiplicação de matrizes e para outros algoritmos, sugerimos a referência [Frigo et al. 1999] que foi publicada à duas décadas atrás. Sugerimos também a aula com uma análise de complexidade do exemplo das matrizes apresentada na aula do Prof. Charles Leiserson do MIT [Anônimo].

Porém, além do desempenho, o grande desafio é a compreensão da potência dissipada [Livingston et al. 2017] pelas operações aritméticas, pelos acessos a cache e pelos acessos a memória RAM. Estudos recentes [Dally 2015], mostram que enquanto uma operação aritmética consome 20 pJ, um acesso a memória RAM consome 20 nJ, ou seja, três ordens de grandeza a mais em energia consumida. Portanto, o reuso do dado dentro do processador tem muito impacto no consumo além do impacto no desempenho.

```
void Mult(float C[][N], float a[][N], float b[][N]){
    for (int k = 0; k < N; k++)
        for (int i = 0; i < N; i++){
            C[k][i] = 0;
            for (int j = 0; j < N; j++)
                C[k][i] += a[k][j] * b[j][i];
        }
}
```

(a)

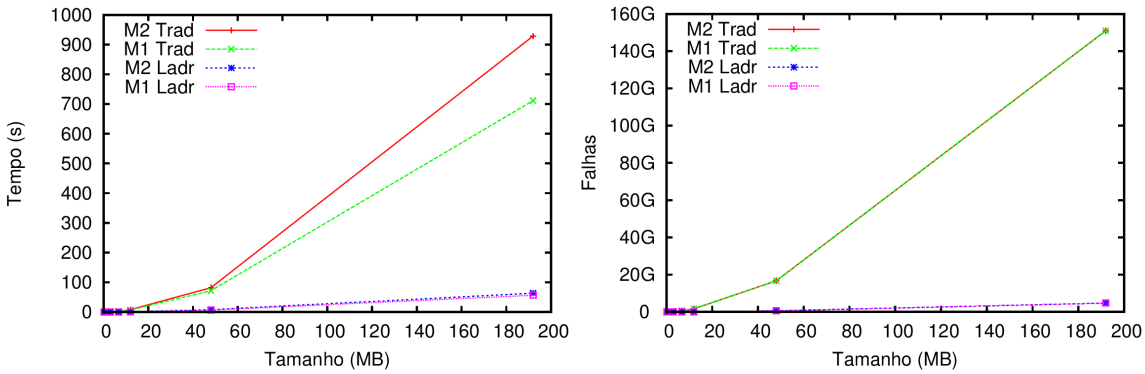
```
void Tiled_Mult(float C[][N], float a[][N], float b[][N], int
tile_size){
  float sum;
  for (int ii = 0; ii<N; ii+=tile_size)
    for (int jj = 0; jj<N; jj+=tile_size)
      for(int kk = 0; kk<N; kk+=tile_size)
        for (int i = ii; i < MIN(ii+tile_size,N); i++)
          for (int j = jj; j < MIN(jj+tile_size,N); j++) {
            sum = 0;
            for (int k = kk; k < MIN(kk+tile_size,N); k++)
              sum += a[i][k] * b[k][j];
            C[i][j]+= sum;
          }
}
```

(b)

Figura 1. (a) Multiplicação com 3 Laços Aninhados (b) Versão com Ladrilhos.

3. Impacto das Caches

Nesta seção iremos avaliar o impacto das caches com uma abordagem prática com códigos disponíveis para os leitores [Anônimo]. Continuando com o exemplo de multiplicação de matrizes, uma avaliação de desempenho em dois processadores é apresenta da figura 2(a). Os processadores têm três níveis de caches. As caches de primeiro (L1) e segundo (L2) nível tem o mesmo tamanho na maioria dos processadores Intel, sendo 32 Kbytes para L1 e 256 Kbytes para L2. A diferença de desempenho pode ser observada em função do tamanho da matriz, da frequência de relógio e do tamanho da cache L3 como ilustrado na figura 2(a). O processador M_1 é Intel i7, 4 núcleos, *clock* 3.4GHz e 8MB de cache L3. O processador M_2 é Intel Xeon, 8 núcleos, *clock* 2.4GHz e 20MB de cache L3. Como explicar que o processador M_2 com mais cache, dobro de núcleos tem um desempenho pior. O motivo é que o *clock* é menor e para ter desempenho é necessário fazer uso do processamento paralelo com 8 núcleos. Ou seja, mesmo custando o dobro do preço, o processador M_2 só será eficiente, se o programador explorar o paralelismo da arquitetura com múltiplos núcleos.



(a) (b)

Figura 2. (a) Tempo de execução para os processadores M_1 e M_2 para a versão tradicional e a versão com ladrilhos. (b) Falhas nas Caches L1 medidas com *perf*.

A análise de falhas nas caches é complexa mesmo para a multiplicação de matrizes. O uso de ferramentas para emular a execução e/ou monitorar é uma maneira simples

para entender o porque a versão com ladrilhos é 10 vezes mais rápida. A ferramenta Valgrind [Nethercote and Seward 2007] emula a execução do código e emite relatórios detalhados dos acessos as cache como ilustra a figura 3(a). Entretanto, para entradas grandes de dados, a emulação pode ser demorada. Os processadores atuais incluem contadores em hardware que fornecem informações sobre o número de acessos e falhas nas caches. A ferramenta *perf* [Weaver 2013] permite gerar relatórios com os dados da execução fazendo acesso aos dados internos dos processadores. A figura 3(b) ilustra algumas medidas realizadas com a ferramenta *perf*. Usando esta ferramenta fica claro entender com a figura 2(b) que o padrão de falhas influencia diretamente o tempo de execução, onde o perfil dos gráficos de execução e falhas é bem similar. Resumindo, mesmo com a mesma complexidade a nível de operações e um número menor de adições, o algoritmo tradicional de multiplicação de matrizes pode ser 10 vezes pior em desempenho devido as falhas em cache para matrizes grandes da era *big data*.

```
chrono time(s): 41.317193
I   refs:       7,886,395,077
D1  misses:     74,798,543
LLd misses:     264,479
LL refs:        74,800,423
LL misses:      266,266
```

(a)

```
0.916024      chrono time(s)
7.896.018.683 instructions
1.786.247     cache-references
371.888       cache-misses
75.147.376    L1-dcache-load-misses
979.907       L1-dcache-store-misses
```

(b)

Figura 3. Multiplicação de Matrizes 1024x1024 (a) Dados das emitidos pelo *Valgrind*; (b) Dados das emitidos pelo *Perf*

Na figura 3, os resultados foram para a execução de uma multiplicação de matrizes com dimensões $n = 1024$, ou seja, tamanho 1024x1024. Considerando que a complexidade em operações é $O(n^3)$, que para este exemplo corresponde a um bilhão e o número de instruções é proporcional, 7 a 8 instruções por operação básica do algoritmo, onde é necessário calcular índices, ler os dados, multiplicar e somar. Outro aspecto é o desempenho do *perf* que executou em menos de 1 segundo realizando as medidas em um processador com 3.4 GHz. Ou seja, o processador foi monitorado e ainda assim executou 2 instruções por ciclo. Entretanto, o *Valgrind* que emula a execução, demorou 41 segundos para executar, ou seja, não é viável para exemplos grandes. Com relação as medidas, ambas as ferramentas geraram valores bem próximos para número total de instruções e falhas na cache L1 (ou D1 no Valgrind).

Atualmente, os efeitos dos sistemas de memória e cache são bem mais complexos, sendo que aplicações independentes podem afetar o desempenho de outras aplicações, onde estimativas de como mitigar estes problemas são apresentadas em [Subramanian et al. 2015] para processadores com múltiplos núcleos.

4. Compiladores

Ao analisar o impacto das caches e o número de instruções executadas deve se ter uma atenção especial sobre a influência do compilador nas otimizações. Considere o exemplo de contagem de instruções ilustrado na figura 3. Este exemplo foi gerado com o compilador gcc com a opção de otimização -O3. Foram executadas 7,895 bilhões de instruções em 0,916 segundos na máquina M_1 a 3,4 Ghz, que implica em 2,4 instruções

por ciclo. Se o código for compilado sem otimizações, o resultado é de 3 instruções por ciclo. Observando esta métrica pode se pensar que o processador é mais eficiente sem a otimização. Entretanto, como destacado pelos professores Patterson & Hennessy, prêmio Turing de 2018, em seu livro clássico [Patterson and Hennessy 2007], para desempenho temos sempre que olhar o tempo de execução. Neste caso, sem otimização, o código tem 5 vezes mais instruções e executa em um tempo 4 vezes mais lento, mesmo tendo uma vazão maior de instruções por ciclo. Deve-se ter cuidado, pois as instruções X86 não tem o mesmo custo e as operações complexas podem ser mais eficientes. Uma vazão menor de instruções pode gerar um desempenho melhor. Apesar de parecer óbvio para o desempenho, muitos artigos recentes apresentam a métrica de eficiência energética em Giga operações por Watt, onde é válida a mesma observação que não se deve focar nesta métrica isolada, e sim no consumo total de energia para executar a tarefa, pois as instruções podem ser diferentes gerando distorções mesmo avaliando em um único processador. Se for comparar plataformas diferentes como CPU e GPU, devemos estar atentos a estes fatos para energia.

5. Mistérios das Memórias

Esta seção é dedicada a apresentar alguns mistérios referente às memórias. Os temas abordados fazem parte de aulas para introdução de sistemas lógicos e arquitetura de computadores para alunos do primeiro ano da Universidade de Zurich. Com apenas duas aulas teóricas disponíveis em [Anônimo], é possível incluir este tema nos cursos de programação, estrutura de dados e complexidade de algoritmos, motivando os alunos a buscarem mais informações referentes a estes conceitos e efeitos colaterais das memórias.

5.1. Perda de Desempenho com tarefas independentes em Multi-cores

Mesmo para telefones portáteis, os processadores possuem múltiplos núcleos e executam programas independentes que disputam concorrentemente o acesso ao sistema de caches e memória principal. Considere o cenário ilustrado na figura 4 onde dois processos A e B executam tarefas independentes em núcleos distintos de processamento. O processo A faz um acesso com padrão sequencial a um grande volume de dados. O processo B faz um acesso aleatório. Neste cenário irá ocorrer uma perda de desempenho para o processo B que será retardado pela execução concorrente do processo A [Woo and Lee 2007]. O problema exige uma visão mais ampla do funcionamento da controladora da memória DRAM, ainda que estes processos, tenham a mesma prioridade de execução perante o sistema operacional. O processo A irá ativar uma linha da DRAM e o controlador da memória dará prioridade aos seus pedidos, pois uma vez ativada a linha (que tem um custo em tempo e energia), os dados solicitados que estão próximos serão priorizados antes de ativar uma outra linha qualquer. Este evento é agravado pelo fato do processo B utilizar um padrão de acesso à memória aleatório, causando muitas falhas de cache L3. Ademais, outras complicações podem ocorrer, como um efeito colateral semelhante a ataques de negação de serviço, onde um processo consome todos os recursos obrigando aos demais processos aguardarem demasiadamente [Moscibroda and Mutlu 2007].

5.2. *RowHammer*

RowHammer ou "martelo de linha" é um efeito colateral que gera falhas devido a arquitetura interna das DRAMs. As memórias modernas são constituídas de linhas de

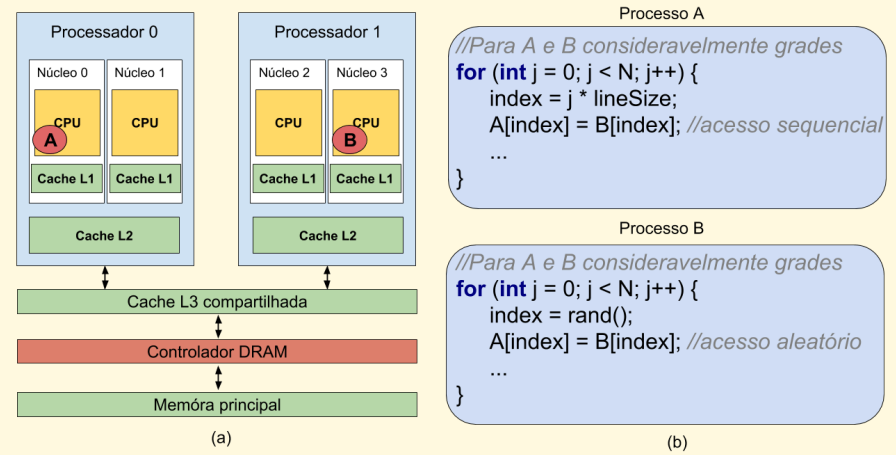


Figura 4. (a) Arquiterura Multi-cores (b) Processos independentes

células como ilustrado na figura 5 (a). Para acessar qualquer dado, primeiro a linha deve ser ativada e para isto é necessário alimentá-la com uma tensão elevada, como mostra a figura 5 (b). Ao término do acesso, a linha pode ser fechada aplicando uma tensão baixa (figura 5 (c)). A perturbação de "martelar" a linha ocorre quando alterna-se o acesso frequentemente. Neste momento ocorre o efeito colateral da inversão de bits (*bit flipping*) dos valores das células adjacentes à linha martelada (figura 5 (d)). Este tipo de falha permite que processos maliciosos alterem os conteúdos de partes arbitrárias da memória principal [Kim et al. 2014]. Ou seja, um simples ataque em software com leituras e escritas pode danificar os dados de outras regiões de memória.

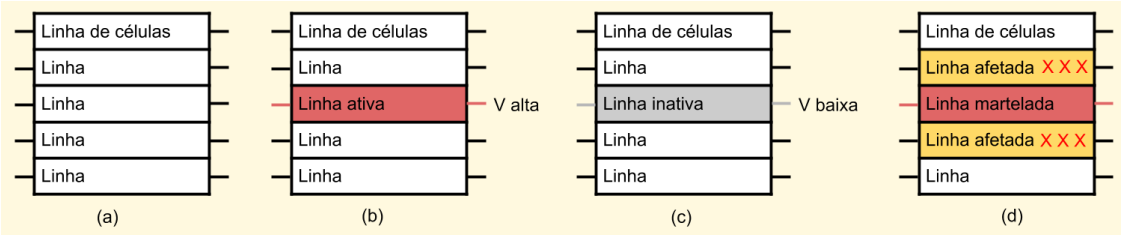


Figura 5. (a) Arquitetura das memórias. (b) Linha ativa. (c) Linha inativa. (d) Linha martelada

5.3. *Meltdown* e *Spectre*

Desde dos anos 90, os processadores buscam várias instruções a cada ciclo de relógio para obter alto desempenho. No exemplo da figura 3, 7,8 Giga instruções são executadas em 0,9 segundos com um *clock* de 3,4 Ghz, o que implica que foram executadas 2,5 instruções por ciclo. Entretanto, em média 20% das instruções são desvios condicionais. Suponha que um processador execute 3 instruções por ciclo que é equivalente a 0,33 ciclos por instrução (CPI). Se para resolver um desvio for necessário 2 ciclos e tivermos 20% de desvio, o desempenho medido em ciclos por instrução será $CPI = 0,33 + (T_{desvios} * T_{calculodesvio}) = 0,33 + (0,2 * 2) = 0,73$. Ou seja, o processador ficará duas vezes mais lento ! Para não prejudicar o desempenho, as técnicas de execução especulativa são empregadas. Suponha que a técnica tenha uma taxa de erro de apenas 5%, então o novo desempenho pode ser medido como

$CPI = 0,33 + (T_{erros} * T_{desvios} * T_{calculodesvio}) = 0,33 + (0,05 * 0,2 * 2) = 0,35$. Ou seja, uma perda de menos de 10% apenas. Por isto, a execução especulativa foi amplamente implementada nas últimas três décadas por todos os fabricantes de processadores. Para aumentar ainda mais o desempenho, durante a especulação, algumas verificações com proteção de memória são relaxadas.

No final de 2017, a divulgação da vulnerabilidade a dois novos ataques *Meltdown* e *Spectre* para todos os processadores fabricados desde de 1995 [Kocher et al. 2018, Lipp et al. 2018] teve um grande impacto mundial, pois ameaça a segurança de todos os sistemas computacionais. A base destes ataques é a execução especulativa, porém para compreender os ataques, que envolvem vários outros detalhes, os programadores precisam ter uma visão bem mais ampla das camadas de hardware. Além da especulação, estes ataques exploram os mecanismos de proteção de memória, as memórias caches, os preditores de desvio e os contadores de ciclos implementados em hardware disponíveis nos processadores.

O erro de projeto foi gerado ao implementar os mecanismos especulativos e as correções em caso de erro sem considerar a presença de memórias cache e contadores em hardware. Durante a especulação, os dados das instruções são armazenadas em registradores temporários que são descartados caso a especulação seja falsa, o processador perde alguns ciclos e retorna ao caminho de execução correto. Porém, estes dados podem deixar vestígios na cache. Todos os dados da memória para os registradores passam pela cache.

A base do ataque é executar um comando condicional (ex:*if (i < dado.Tamanho())* sabendo que ele será especulado e uma proteção de memória (tamanho de um vetor) não será verificada durante a especulação. Dentro do bloco do comando condicional, um acesso a uma área protegida irá ocorrer e uma cópia do dado sigiloso pode ficar na cache. Detectado que a condição é falsa, os registros são apagados, porém ficam os dados na cache. Então através de um outro trecho código, o acesso é feito a algumas variáveis, se elas estiveram na cache, detectado pelo tempo de resposta às leituras, estes valores são descobertos e dizem respeito as áreas protegidas que foram transferidas sem verificação durante a especulação. Para maiores detalhes em nível introdutório ou mais técnicos, sugerimos aos leitores as referências [Lipp et al. 2018, Kocher et al. 2018, Anônimo].

5.4. *DRAM Refresh*

As memórias DRAM são memórias dinâmicas onde cada bit é armazenado em um capacitor. Periodicamente, o capacitor deve ser realimentado para manter a carga que armazena. Esta ação é conhecida pelo termo *refresh*. Em um DRAM, tipicamente a cada 64 milisegundos, cada linha deve realizar uma operação de *refresh*. Este operação além de consumir energia, deixa a memória inacessível durante este evento, aumentando sua latência. Como o aumento dos tamanho das memórias, o tempo gasto com o *refresh* pode chegar até 46% para módulos de memória de 64 GB. Isto afeta não só a computação na nuvem como também a computação móvel prejudicando o desempenho dos *smartphones* e o consumo de energia, onde a memória DRAM tem um impacto grande. O tempo de *refresh* é determinado pelo pior caso, porém a grande maioria das células, 95% ou mais, pode reter os valores por 256 milisegundos ou mais. Isto permite que invasores com acesso físico ao hardware consigam extrair informações secretas, como chaves criptográficas, segundos ou minutos após a desenergização da máquina [Gruhn and Muller 2013]. Outro aspecto

seria explorar algoritmos que toleram pequenos erros nos dados, aumentando assim o tempo de *refresh* para de 64ms para 256ms, melhorando o desempenho e reduzindo o consumo de energia.

6. Novas Arquiteturas

Os sistemas computacionais tradicionais tem se tornado cada vez mais poderosos e eficientes com o passar dos tempos. Entretanto as memórias não conseguiram acompanhar estes avanços, tornando-se o principal gargalo de desempenho e consumo energético [Zhang et al. 2013]. Um novo modelo de memória propõe que o processamento seja realizado mais próximo dos dados. Este modelo é conhecido com processamento em memória (PIM - Processor In Memory) transforma a memória em uma entidade ativa, capaz de realizar computação e processamento direto onde são armazenados os dados. Que é um contra ponto a arquitetura atual que tem mais de 50% da área do processador ocupado por caches, ou seja, o modelo atual poderia ser referenciado como MIP (Memory in Processor). Na nova proposta baseada nas arquiteturas PIM, dispositivos sem processadores, produzidos com memória PCM (memória de mudança de fase) foram capazes de executar algoritmos de aprendizado de máquina com desempenho até 200 vezes mais rápido e energeticamente eficiente quando comparado com GPU's atuais [Sebastian et al. 2017].

Para contornar os problemas de *refresh* das DRAM, as memórias não voláteis (NVM) evoluíram significativamente em tamanho e desempenho nos últimos anos. NAND é a principal tecnologia empregada nos discos de estados sólidos (SSD). Esta tecnologia de semicondutores tem-se aproximado dos limites físicos, contudo os fabricantes têm desenvolvido soluções multinível (*3D NAND*) [Cappelletti 2015]. Entretanto esta tecnologia não é suficiente ainda para solucionar os problemas relacionados à latência. Outra tecnologia promete revolucionar o mercado de memórias não voláteis é a *3D XPoint*. Esta arquitetura apresenta desempenho e latência três vezes mais eficiente que os SSD's tradicionais. Permite ainda que seja aplicadas em cenários distintos, podendo ser utilizadas como uma extensão de memória DRAM, armazenamento de dados e/ou memória persistente. Consequentemente, a arquitetura *3D XPoint* proporcionará oportunidades para explorar aplicações que necessitam de grande desempenho relacionado à memória [Hady et al. 2017].

Outras tendências são as arquiteturas heterogêneas já presentes nos sistemas atuais como as GPUs. Uma GPU possui memórias DDR com vazão próxima a 1 Tera Byte por segundo, quase 100 vezes mais alta que a RAM dos processadores. A GPU consegue esconder a latência da memória com milhões de threads concorrentes em comparação com poucas dezenas de threads dos processadores tradicionais. Possuem cache e memórias em bancos reconfiguráveis em tempo de execução para se ajustar ao padrão de dados. Possuem também milhões de registradores para manter as variáveis locais apenas nos registradores sem precisar de usar a cache ou a memória RAM.

Outro modelo que vem sendo empregado são os aceleradores com FPGA (Field Programmable Gate Arrays) que são circuitos reconfiguráveis após a fabricação. Os FPGAs podem se adaptar a uma aplicação específica sendo personalizados para executar diretamente em hardware as funções. Por exemplo, um FPGA moderno possui mais 2000 somadores/multiplicadores e 2000 bancos de memória local que pode ser interligados de

qualquer maneira para modelar algum algoritmo, executando até 800 Giga operações por segundo e consumindo apenas 20 Watts em comparação com os 200 Watts de uma CPU de alto desempenho. Assim como as GPU, os FPGA vem sendo usados para implementar as redes neurais profundas que possuem muitas aplicações atualmente. Conceitualmente, um FPGA é uma grande matriz de pequenas memórias (da ordem de 200 mil módulos) que podem ser interligadas para modelar qualquer aplicação onde cada módulo implementa uma função Booleana de 5-8 variáveis. Os FPGAs são 10 vezes mais efiecientes em operações por Watt que as GPUs e CPUs. Recentemente, a Intel comprou a maior fabricante de FPGA, a empresa Altera, em uma transação de 16 bilhões de doláres. Além disso, a Microsoft e Amazon já fazem uso dos FPGAs nas nuvens para prover soluções com eficiência energética. A maior barreira para os FPGA é a falta de ferramentas para a comunidade de programadores de linguagens C,C++ ou equivalentes.

Outra tendência para o uso de FPGAs é o modelo de fluxo de dados, onde não existe busca nem decodificação de instruções, os dados são processados localmente e percorrem o grafo que descreve o algoritmo. O modelo de fluxo de dados explora o paralelismo temporal com as pipeline e o paralelismo espacial com replicações.

7. Conclusões

Este artigo aborda os sistemas de memórias, sua evolução, sua estrutura, seu ensino e seus desafios. Com a tendência do *Big Data*, a tradicional Ciência da Computação ganha um novo ramo que é a Ciência de Dados e novos paradigmas vem sendo propostos com novos desafios de eficiência energética, desempenho e segurança. Entender o modelo atual e o funcionamento das memórias permite ampliar a visão sobre ensinar e projetar algoritmos.

Neste trabalho usamos um exemplo simples de multiplicação de matrizes para destacar o impacto das caches para projetar algoritmos e estrutura de dados com eficiência energética e desempenho. Destacamos também como podemos extrair métricas durante a execução de programas como o monitoramento das falhas em cache. Ilustramos também os temas abordados em duas aulas do curso de graduação da Universidade de Zurich (ETH), onde quatro mistérios envolvendo as memórias, seu desempenho, vulnerabilidades e consumo que podem ser incluídos em curso de programação ou arquitetura para motivar os estudantes a ter uma visão ampliada dos sistemas de computação atuais. Por fim, apontamos algumas direções das novas arquiteturas que junto com os processadores tradicionais formam e continuaram a formar os sistemas de computação heterogêneos onde desenvolvemos nossas aplicações.

Referências

Anônimo. Apontadores para aulas. https://github.com/anonimogit/Aulas.git.

Cappelletti, P. (2015). Non volatile memory evolution and revolution. In *Electron Devices Meeting (IEDM), 2015 IEEE International*, pages 10–1. IEEE.

Dally, W. J. (2015). Challenges for future computing systems. Hipeac Keynote.

Frigo, M., Leiserson, C. E., Prokop, H., and Ramachandran, S. (1999). Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science*.

Gruhn, M. and Muller, T. (2013). On the practicability of cold boot attacks. In *Proc. Availability, Reliability and Security (ARES)*.

Hady, F. T., Foong, A., Veal, B., and Williams, D. (2017). Platform storage performance with 3d xpoint technology. *Proceedings of the IEEE*, 105(9):1822–1833.

Kim, Y., Daly, R., Kim, J., Fallin, C., Lee, J. H., Lee, D., Wilkerson, C., Lai, K., and Mutlu, O. (2014). Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. In *SIGARCH Computer Architecture News*.

Kocher, P., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., and Yarom, Y. (2018). Spectre attacks: Exploiting speculative execution. *ArXiv e-prints*.

Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., and Hamburg, M. (2018). Meltdown. *ArXiv e-prints*.

Livingston, K., Landwehr, A., Monsalve, J., Zuckerman, S., Meister, B., and Gao, G. R. (2017). Energy avoiding matrix multiply. In *Languages and Compilers for Parallel Computing*. Springer International Publishing.

Moscibroda, T. and Mutlu, O. (2007). Memory performance attacks: Denial of memory service in multi-core systems. In *USENIX Security Symposium*.

Nethercote, N. and Seward, J. (2007). Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6).

Patterson, D. A. and Hennessy, J. L. (2007). Computer organization and design. *Morgan Kaufmann*.

Sebastian, A., Tuma, T., Papandreou, N., Le Gallo, M., Kull, L., Parnell, T., and Eleftheriou, E. (2017). Temporal correlation detection using computational phase-change memory. *Nature Communications*, 8(1):1115.

Subramanian, L., Seshadri, V., Ghosh, A., Khan, S., and Mutlu, O. (2015). The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory. In *MICRO*.

Šidlauskas, D. and Jensen, C. S. (2014). Spatial joins in main memory: Implementation matters. *Proc. VLDB Endow.*, 8(1):97–100.

Weaver, V. (2013). Linux perf event features and overhead. In *Int. Workshop on Performance Analysis of Workload Optimized Systems, FastPath*.

Wolf, M. E. and Lam, M. S. (1991). A data locality optimizing algorithm. In *Conference on Programming Language Design and Implementation*. ACM.

Woo, D. H. and Lee, H. (2007). Analyzing performance vulnerability due to resource denial of service attack on chip multiprocessors. In *Workshop on Chip Multiprocessor Memory Systems and Interconnects*.

Zhang, D. P., Jayasena, N., Lyashevsky, A., Greathouse, J., Meswani, M., Nutter, M., and Ignatowski, M. (2013). A new perspective on processing-in-memory architecture design. In *ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*.