



**WSCAD 2019**

## Minicursos do WSCAD 2019

XX Simpósio em Sistemas Computacionais de Alto Desempenho  
Campo Grande – MS

16 a 18 de outubro de 2019

Porto Alegre  
Sociedade Brasileira de Computação – SBC  
2019

Minicursos do WSCAD 2019  
XX Simpósio em Sistemas Computacionais de Alto Desempenho  
Campo Grande – MS  
16 a 18 de outubro de 2019

**Organizadores**  
Ricardo Menotti  
Guilherme Galante

ISBN: 978-85-7669-486-1

Porto Alegre  
Sociedade Brasileira de Computação – SBC  
2019

Dados Internacionais de Catalogação na Publicação (CIP)  
(Divisão de Editora da UFMS, Campo Grande, MS, Brasil)

---

Simpósio em Sistemas Computacionais de Alto Desempenho  
(20. : 2019 : Campo Grande, MS).

Anais do Minicurso do Wscad, 16 a 18 de outubro de 2019/organizadores,  
Ricardo Menotti, Guilherme Galante – Porto Alegre: SBPC, 2019.

ISBN 978-85-7669-486-1

Disponível em: < <https://sol.sbc.org.br/index.php/indice> >

1. Computação de alto desempenho . 2. Programação paralela (computação). I.  
Menotti, Ricardo. II. Galante, Guilherme. III. Universidade Federal de Mato  
Grosso do Sul. IV. Título.

CDD (22) 004.35

---

Elaborada pela Bibliotecária Lilian Aguilar Teixeira CRB 1/2448

## Conteúdo

1. **Métricas e Números: Desmistificando a Programação de Alto Desempenho em GPU**  
*Ricardo Ferreira, Salles Viana Gomes de Magalhães, José A M Nacif*  
..... 5
2. **Introdução à Programação com OpenACC**  
*Evaldo B. Costa, Gabriel P. Silva*  
..... 35

## Capítulo

# 1

## Métricas e Números: Desmistificando a Programação de Alto Desempenho em GPU

Ricardo Ferreira<sup>1</sup>, Salles Viana Gomes<sup>1</sup> de Magalhães, José A M Nacif<sup>1</sup>

### *Resumo*

*Este minicurso apresenta os fundamentos e as características das GPUs de última geração (Turing, Volta, Pascal) para que os pesquisadores possam avaliar e compreender o desempenho e os gargalos no desenvolvimento de aplicações. Serão apresentadas as métricas e as técnicas com microbenchmarks para identificar e mitigar os gargalos, evitar equívocos nas interpretações quantitativas, além de uma comparação detalhada das arquiteturas e suas estruturas de memória. Uma melhoria no processo de medida e na experimentação possibilita um entendimento claro para obter alto desempenho, além de compreender os resultados reportados por outros trabalhos da área.*

### **1.1. Introdução**

Com um pouco mais de uma década [Cheng et al. 2014], as GPUs são aceleradores já incorporados na maioria dos supercomputadores, *datacenters*, computadores pessoais e celulares. Uma GPU pode ser programada de diversas formas, as mais comuns são através da API CUDA específica para GPUs da *Nvidia* e do framework do consórcio OpenCL, que é mais genérico. Este trabalho irá utilizar a API CUDA e desmistificar as arquiteturas recentes da *Nvidia*.

As GPUs introduziram um novo modelo de programação no qual milhões de *threads* podem ser disparados no paradigma SIMT (*Single Instruction Multiple Threads*). A Figura 1.1(a) ilustra uma chamada de função (ou *kernel*) que é realizada pela CPU (linha 8)). Os múltiplos *threads* serão executados na GPU. O código da CPU também faz o controle da alocação de memória da GPU (linhas 5-6) e da transferência de dados entre a CPU e a GPU (linhas 7 e 9). A CPU pode continuar executando em paralelo (linha 9) enquanto a GPU executa um ou mais *kernels*. O exemplo ilustra uma avaliação de um

---

<sup>1</sup>Apoio Financeiro: FAPEMIG, Nvidia, CNPq, Funarbe. O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001.

polinômio e o código do *kernel* é ilustrado na Figura 1.1(b). A Figura 1.1(c) ilustra uma versão em CPU do mesmo código.

<pre>int main() { ..... 1  h_pol  = (float *)malloc(nBytes); 2  h_x    = (float *)malloc(nBytes); ..... 3 // Alocando memória na GPU 4  float *d_x, *d_pol; 5  cudaMalloc((float**)&amp;d_x, nBytes); 6  cudaMalloc((float**)&amp;d_pol, nBytes); 7  cudaMemcpy(d_x, h_x, nBytes, cudaMemcpyHostToDevice); 8  kernel&lt;&lt;&lt;10000, 1000&gt;&gt;&gt;(d_x, d_pol, nElem); 9  cudaMemcpy(h_pol, d_pol, nBytes, cudaMemcpyDeviceToHost) .....; }</pre>	<pre><u>__global__</u> void kernel (float *x, float *pol, const int N) {   int i = blockIdx.x * blockDim.x + threadIdx.x;   if (i &lt; N) pol[i] = 3*x[i]*x[i]-7*x[i]+5; }</pre>
(a)	(b)
	<pre>for ( int i=0; i &lt; N; i++)   pol[i] = 3*x[i]*x[i]-7*x[i]+5;</pre>
	(c)

**Figura 1.1. (a) Código na CPU de chamada da GPU; (b) Código do *kernel* na GPU; (c) Código equivalente em CPU;**

Para simplificar a explicação, usaremos potências de 10 ao invés de potência de 2. Suponha que a CPU dispare 10 milhões de threads. Em CUDA, os *threads* são organizados em blocos, o que facilita a infraestrutura para execução na arquitetura. Existe um limite do número máximo de *threads* por bloco (1024 a 2048 nas arquiteturas atuais). Portanto, se quisermos disparar 3000 *threads*, precisaremos de 2 ou mais blocos. No exemplo da Figura 1.1(a) foram disparados 10 milhões de *threads* organizados em 10 mil blocos de mil *threads* cada. Na execução, a GPU irá enviar para cada multiprocessador um conjunto de *b* blocos. O número de blocos em execução em um multiprocessador depende das características do código e da arquitetura da GPU. Por exemplo na arquitetura Volta, *b* pode ter o valor entre  $0 \leq b \leq 16$ . A cada *clock*, a unidade de busca do multiprocessador buscará uma instrução para um grupo de 32 *threads*, denominado *warp*. Apesar de todos os *threads* do *warp* executarem sincronamente a mesma instrução<sup>2</sup>, cada *thread* terá um identificador único formado pelo número do *thread* (de 0 a 999, no nosso exemplo) e o número do bloco (0-9999). O identificador é usado para indexar a tarefa realizada pelo *thread*. No nosso exemplo, cada *thread* irá avaliar um ponto do polinômio, buscando na posição indexada pelo seu identificador calculado pela expressão  $\text{bloco} \cdot 10000 + \text{thread}$ . Suponha um *thread* qualquer de um bloco qualquer, por exemplo o *thread* 159 do bloco 314. Este *thread* irá gerar o identificador **0314159** e irá acessar esta posição de memória, onde os 4 dígitos mais significativos são o número do bloco e os três últimos dígitos são o número do *thread* dentro do bloco. Como veremos mais adiante, o acesso as posições consecutivas por *threads* com indexação consecutiva gera um padrão de acesso que a GPU consegue executar de forma eficiente.

Apesar de simples, podemos destacar alguns detalhes para começar a desmistificar como a GPU obtém desempenho com este primeiro exemplo:

1. A transferência de dados entre a GPU e CPU é limitada pelo barramento PCI (na faixa de 1-12 GB/s). Portanto, a GPU deve reusar o dado efetuando muitos cálculos para compensar o tempo de transferência (linhas 7 e 9 no código exemplo da Figura 1.1(a)). Supondo 10 Milhões de *threads* com um vetor de 10 Milhões de

<sup>2</sup>Supondo um cenário sem comandos condicionais

*float*, o tempo de transferência foi de 28 ms executando em uma GPU da geração Tesla.

2. A leitura dos dados na GPU (acesso à variável  $x$  no código Figura 1.1(b)) demora em torno de 800 ciclos. Portanto, o *thread* irá para uma fila de espera para aguardar os 800 ciclos após executar uma instrução de leitura. A arquitetura da GPU resolve este problema gerenciando milhares de *threads* ao mesmo tempo, onde a troca de contexto pode ser realizada em um ciclo de *clock*. Ou seja, centenas de *threads* serão disparados e irão para a fila de espera para esconder a latência da memória, até que o primeiro dado da memória chegue a GPU e todos executem em pipeline. Porém, existe um preço a ser pago, que é manter vivos nos registradores os valores para cada *thread* em estado de execução ou espera. Portanto, a GPU necessita de um grande número de registradores e um gerenciamento eficiente para troca de contexto. Uma GPU da arquitetura Volta V100 tem 80 multiprocessadores com 64K registradores cada, totalizando mais de 5 milhões de registradores que ocupam 15% da área em silício e corresponde a 21 Megabytes que é mais de três vezes o tamanho da cache L2 da GPU Volta, que tem 6 Megabtes de cache.
3. Considerando que um grande número de *threads* está em execução, mascarando a latência da memória, a GPU tem uma taxa de leitura (de sua memória principal) de pico que pode chegar à 200-800 GB/s, uma das suas maiores qualidades. Em nosso exemplo, com 10 milhões de *threads*, o tempo de execução total do *kernel* foi de 2 ms, o que equivale a uma taxa de leitura de apenas  $10000000 * 4 / 0,002 = 20\text{GB/s}$ . O fator 4 é devido ao fato de cada elemento ser um float de 32 bits ou 4 bytes. Podemos fazer duas observações. A primeira é que o tempo de transferência de dados entre a GPU e CPU para este exemplo foi 10x maior que o tempo de execução. Portanto é necessário realizar dezenas de cálculos com os dados para compensar o tempo gasto na transferência. A segunda observação é que para uma pequena quantidade de dados (10 milhões ou menos), o desempenho da GPU será bem inferior ao desempenho de pico. Portanto, a GPU terá alto desempenho se a quantidade de dados é significativamente maior, da ordem de pelo menos centenas de mega bytes.
4. Considerando 300 GB/s para ler o dado na arquitetura Pascal para uma GTX 1070, podemos perguntar como a GPU pode ter um desempenho de mais de 1 Tera operações por segundo? A resposta é simples: para cada dado lido a uma taxa 300 GB/s, a GPU precisa executar pelo menos 4 operações para gerar um desempenho de 1 Tera operações. No exemplo da Figura 1.1(b), como podemos calcular quantas operações são executadas por operação de memória? Primeiro podemos notar que para avaliar a expressão  $Pol[i] = 3 * x[i] * x[i] - 7 * x[i] + 5$  são realizadas 3 multiplicações e duas somas. Ou seja, 5 operações. A GPU irá ler apenas uma vez o elemento  $x[i]$  do vetor de entrada, armazenar em um registrador interno, efetuar os cálculos e gravar uma vez na memória no elemento  $Pol[i]$  do vetor de saída. Ou seja, executamos apenas 5 operações para 2 operações de memória. Considerando que o tempo de execução foi de 2ms, então foram executadas 50 milhões de operações, o desempenho foi de 25 Gflops/s. Neste exemplo, podemos avaliar o polinômio  $Pol[i] = 4 * x[i] * x[i] * x[i] + 3 * x[i] * x[i] - 7 * x[i] + 5$  que o tempo de execução não

irá ser alterado, mas o desempenho será de 45 Gflops/s pois agora são 9 operações por elemento. Se executarmos um cálculo mais complexo com dezenas de operações, o desempenho irá também ficar limitado pela latência das instruções além da latência da memória. Neste exemplo, a GPU usa as instruções FMA (multiplica e soma) e o polinômio pode ser executado com apenas três instruções: dois fma e uma multiplicação ( $Pol[i] = 3 * fma((x[i] * x[i]), fma(7 * x[i], 5))$ ).

Este capítulo tem o objetivo de mostrar como instrumentar e medir o desempenho de aplicações mapeadas em GPU para compreender as limitações de desempenho em função da memória, das instruções e de outras características das GPUs como as divergências de controle na execução de um sub-conjunto de *threads* (ou warps). Além das características básicas [Cheng et al. 2014], serão apresentados também detalhes das novas arquiteturas de GPU com as gerações Pascal, Volta e Turing da *Nvidia* [Serpa et al. 2019].

## 1.2. Modelos e Arquiteturas

Primeiro iremos apresentar as origens da arquitetura GPU e do modelo SIMT. Depois iremos descrever o modelo de execução da API CUDA com *threads* e blocos. Em seguida, iremos apresentar a estrutura interna com detalhes da arquitetura do sistema de memória. Para aprofundar no assunto, sugerimos a leitura dos cinco primeiros capítulos do livro [Cheng et al. 2014] cujos exemplos de código estão disponíveis no *link*<sup>3</sup>, adicionamos também um novo conjunto de exemplos no *link*<sup>4</sup>. Para motivar os estudantes e profissionais, mesmo sem acesso direto a uma máquina com uma GPU da *Nvidia*, os experimentos propostos para este minicurso podem ser executados na plataforma online Colab da Google. Maiores informações podem ser encontradas no Apêndice A.

### 1.2.1. Single Instruction Multiple Thread

A classificação de Flynn apresenta 4 categorias: SISD, SIMD, MISD e MIMD. Em uma análise simples, as GPUs seriam enquadradas na categoria SIMD (*Single Instruction Multiple Data*). Entretanto, as GPUs podem ser consideradas um misto em MIMD e SIMD. Para MIMD podemos justificar pois uma GPU tem vários multiprocessadores e a princípio cada multiprocessador pode estar executando um *kernel* diferente. Um mesmo multiprocessador tem vários escalonadores de *warps* e a princípio também poderia estar executando mais de um kernel. Para cada escalonador, vários *warps* estarão executando no modelo SIMT que é semelhante ao modelo SIMD, porém executa vários conjuntos de *threads* como iremos detalhar a seguir fazendo analogia com um processador RISC simples com estágios em pipeline.

A Figura 1.2 apresenta um *pipeline* de um processador no estilo MIPS com uma pequena modificação. Primeiro, o estágio de busca da instrução dispara uma única instrução para um conjunto de *threads*, que serão 4 *threads* no nosso exemplo. Para simplificar a explicação de *warp* (grupo de *threads* que executam em conjunto), iremos supor um *warp* de 4 *threads* e um total 12 *threads* em execução agrupados em três warps. No estágio de decodificação, cada *thread* tem seu próprio banco de registradores. Neste exemplo serão necessários doze bancos de registradores. A execução é em ordem. Ou seja, nenhuma ins-

<sup>3</sup>[https://media.wiley.com/product\\_ancillary/29/11187393/DOWNLOAD/CodeSamples.zip](https://media.wiley.com/product_ancillary/29/11187393/DOWNLOAD/CodeSamples.zip)

<sup>4</sup><https://github.com/cacauvicosa/wscad2019>

trução irá executar fora de ordem como ocorre nos processadores superescalares. Iremos supor também uma latência de 4 ciclos para qualquer instrução no estágio de execução. Porém, como estão em pipeline, a cada ciclo, uma nova instrução termina sua execução.

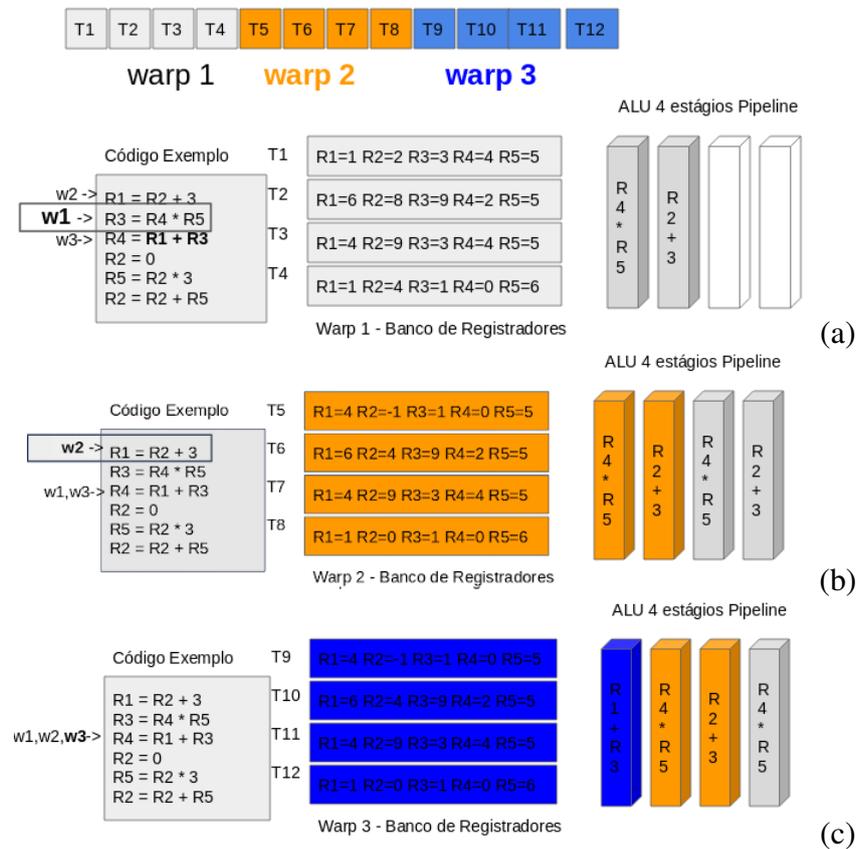


Figura 1.2. (a) Warp<sub>1</sub> com duas instruções em execução; (b) Warp<sub>2</sub> e warp<sub>1</sub> em execução; (c) O warp<sub>3</sub> entra em execução.

No trecho de código do exemplo, cada warp tem seu próprio apontador ou contador de programa  $w_1, w_2$  e  $w_3$ . Algumas instruções não tem dependências de dados, como as duas primeiras. Porém, a terceira instrução depende dos novos valores de  $R_1$  e  $R_3$  calculados pelas duas primeiras. Os warps podem executar em qualquer ordem. Vamos supor que inicialmente o warp<sub>1</sub> está com duas instruções começando a execução no pipeline de 4 estágios. O warp<sub>1</sub> não poderá continuar a execução, pois para calcular  $R_4$  é necessário o novo valor de  $R_1$  e  $R_3$  que ainda está no pipeline de execução como ilustrado na Figura 1.2(a), faltando 2 ciclos para cálculo de  $R_1$  e 3 ciclos para cálculo de  $R_3$ . O warp<sub>1</sub> irá para fila de espera e o próximo warp pode ser o warp<sub>2</sub> ou warp<sub>3</sub>. Suponha que seja o warp<sub>2</sub>, que só poderá executar também a primeira e segunda instrução e depois irá para fila de espera como ilustrado na Figura 1.2(b). Neste momento, o warp<sub>3</sub> deve ser escalonado, uma vez que é o único pronto para executar. O warp<sub>3</sub> já está na terceira instrução. A ordem que os warps executam não é garantida pelo modelo CUDA: cada execução pode ser em uma ordem diferente. Observe que no modelo SIMT, como já mencionado, cada warp tem seu próprio contador de programa e pode estar em uma instrução diferente. Porém múltiplos threads dentro do warp executam a mesma instrução (SIMT). No exemplo da Figura 1.2(a), o warp<sub>1</sub> estava executando a segunda instrução, o

$warp_2$  aguardando para executar a primeira e o  $warp_3$  aguardando para executar a terceira instrução.

Cada *thread* tem o seu próprio conjunto de registradores como ilustrado na Figura 1.2. Quando um *thread* vai para o estado de espera, o banco de registradores permanece vivo. É importante fundamentar as origens das GPUs, que foram introduzidas pela máquina HEP [Smith 1986] e pelo processador Niagara [Kongetira et al. 2005] com a troca de contexto de grão fino em um ciclo de *clock*, mascarando a latência com múltiplos *threads* e uso de diferentes bancos de registradores.

### 1.2.2. Multiprocessadores

Uma GPU tem de centenas a milhares de unidades de execução. A maioria das unidades são simples unidades lógico/aritméticas (ALU) capazes de executar operações com ponto flutuante ou inteiros. As unidades são agrupadas formando multiprocessadores chamados de SM (Stream Multiprocessors). Os primeiros modelos da *Nvidia* (suportando CUDA) como a GPU GTX280, lançada em 2008, possuíam 30 multiprocessadores com 8 unidades de execução cada, totalizando 240 núcleos de execução ou 240 *cores*, termo comumente utilizado. Vale ressaltar que a granularidade do termo núcleo ou *core* em GPU é bem diferente do termo núcleo nos processadores superescalares com múltiplos núcleos. Cada multiprocessador da GPU recebe um ou mais blocos para executar. Os blocos permanecem no multiprocessador até terminarem a execução, quando novos blocos são alocados para o multiprocessador. Como a GTX só tinha 8 unidades, a execução de uma instrução de um *warp* de 32 *threads* demorava em média 27 ciclos. A execução era semelhante ao ilustrado na Figura 1.2 porém com 8 unidades de execução, onde cada unidade tinha um *pipeline* de profundidade de 24 estágios. A cada ciclo, 8 *threads* do *warp* eram disparados em pipeline, demorando 24 ciclos de latência mais 3 ciclos para completar a execução do *warp*.

A evolução das GPUs teve uma tendência inicial em aumentar o número de unidades de execução por multiprocessadores, começando com 15 multiprocessadores de 32 unidades na arquitetura Fermi em 2010, totalizando 480 núcleos. A Fermi também introduziu unidades de ponto flutuante de precisão dupla (64 bits). Em 2012, a arquitetura Kepler foi lançada com um novo modelo de multiprocessador (new Streaming Multiprocessor Architecture ou SMX) com 192 unidades. A GPU GTX680 da geração Kepler possui 1536 núcleos em 8 multiprocessadores, ou seja, três vezes mais núcleos que a geração Fermi por multiprocessador. Esta inovação foi possível graças ao aumento em quase três vezes da eficiência energética dos transistores na implementação em silício da arquitetura Kepler. Vale ressaltar aqui que as GPUs além das inovações no nível de arquitetura, também tiveram ganhos de desempenho e eficiência resultante da evolução das tecnologias em silício. Mais unidades irão requer mais registradores: enquanto a Kepler possui um banco de 64K registradores por SMX a Fermi possui apenas 32K. Porém, se contabilizarmos a proporção de registradores por núcleos, vemos que a Fermi possui 1k registradores por núcleo enquanto a Kepler possui 1/3k registradores por núcleo.

Após a Kepler, começou um movimento contrário para redução na quantidade de núcleos por multiprocessador e aumento na quantidade de memória do multiprocessador por unidade de execução, além de aumentar a quantidade de multiprocessadores para

possibilitar paralelismo a nível de tarefa (MIMD). A próxima geração, a Maxwell, possui 128 unidades por multiprocessador. Outro grande salto foi o lançamento da arquitetura Pascal em 2016 com transistores na tecnologia de 16 nm FinFET em comparação com 40nm da geração da Kepler e 28nm da geração Maxwell. Ao reduzir para 64 unidades por multiprocessador, a Pascal aumentou a relação do número de registradores disponíveis por unidade de execução. Outra inovação foram as operações com precisão de 16 bits (Half float), permitindo duas operações de 16 bits sendo executadas em uma unidade de float 32 bits (precisão simples), dobrando a taxa de pico em operações com ponto flutuante, que tem impacto principalmente em aplicações com aprendizado de máquina.

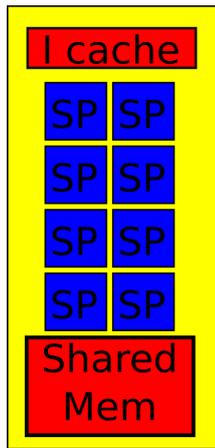
Posteriormente, em dezembro de 2017, a *Nvidia* lançou a arquitetura Volta na tecnologia 12nm FinFet com até 21 bilhões de transistores por *chip*, mais uma evolução na eficiência energética e a introdução das unidades multiplicadoras de matrizes 4x4 de 16 bits, os *tensor core* ou tensores. São unidades sistólicas que podem executar em *pipeline* uma multiplicação e uma soma de matrizes. A Figura 1.3 ilustra a evolução da arquitetura do multiprocessador da GPU da *Nvidia* começando com 8 unidades na arquitetura da GTX280 (Figura 1.3(a)) em 2008 até chegar na arquitetura Volta em 2018 (Figura 1.3(b)). A Figura 1.3(c) mostra a evolução das gerações *Nvidia* em quantidade de unidades e registradores por unidade. As memórias serão detalhadas na próxima seção.

Por fim, podemos destacar o poder de cálculo de uma GPU Volta V100. O multiprocessador é dividido em 4 partições. Cada partição tem 2 unidades tensor, 8 unidades de precisão dupla (64 bits), 16 unidades de precisão simples, 16 unidades de inteiro e uma unidade especial. Importante destacar que quando dizemos que uma V100 com 80 multiprocessadores tem 5120 núcleos, estamos contando apenas as unidades de precisão simples. A V100 ainda terá 2.560 unidades de precisão dupla e 640 unidades tensor. Estes recursos possibilitam que uma V100 seja capaz de executar 40960 operações multiplica/soma (FMA Float Multiply/add), ou 81920 operações de ponto flutuante de 16 bits nas unidades *tensor* por *clock*. Executando a 1,53 Ghz pode chegar a um desempenho de 125 Tera Flops/s. Já as unidades de cálculo de precisão simples e dupla podem gerar um desempenho de pico de 15,7 e 7,8 Tera Flops/s, respectivamente. Além disso, podemos ter 31,4 Tera Flops/s em operações de meia-precisão (16 bits) nas unidades de precisão simples.

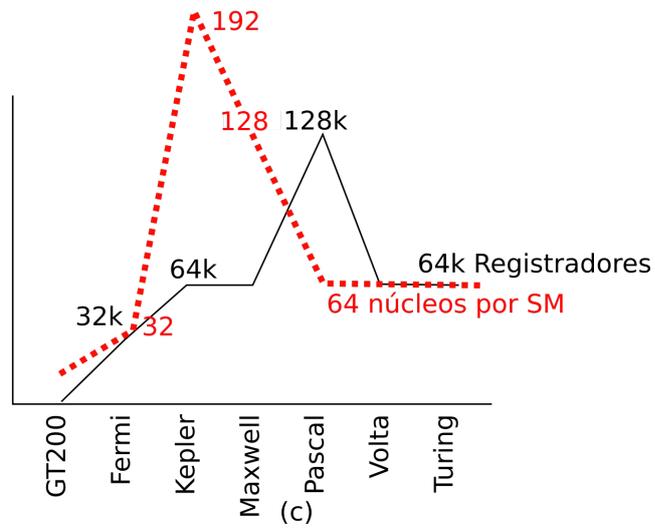
Dado este desempenho, fica a questão: como podemos modelar e implementar algoritmos para explorar todo o potencial das GPUs? Nas próximas seções iremos abordar as arquiteturas de memória e posteriormente, como instrumentar as implementações para extrair o máximo de desempenho.

### 1.2.3. Hierarquia de Memória

Como já mencionado, as GPUs usam milhões de registradores para manter as variáveis locais próximas das unidades de execução, esconder a latência caso existam dependências de dados e maximizar o uso das unidades. Além dos registradores, as GPUs possuem uma hierarquia de memória com semelhanças e diferenças em relação ao sistema de memória da CPU. Primeiro, o espaço de memória da GPU é isolado da CPU. Uma vantagem é não precisar garantir coerência de memória. Uma desvantagem é a limitação da transferência de dados pelo barramento PCI da CPU para GPU e vice-versa. A memória principal



(a)



(c)

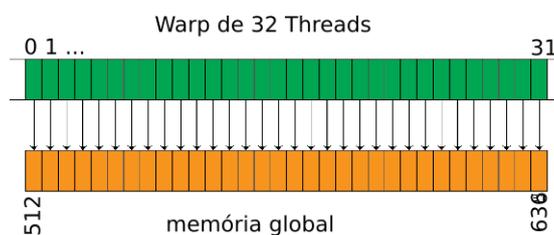


(b)

Figura 1.3. (a) Multiprocessador da GTX 280 (b) Multiprocessador da Arquitetura Volta; (c) Evolução dos recursos por unidade e multiprocessador nas Arquiteturas da Nvidia.

da GPU é denominada pelo termo memória global, pois é visível para todos os threads. Desde as primeiras gerações, a memória global da GPU possui uma vazão elevada de dados em comparação com a CPU, com taxas de vazão de 150-200 GB/s. As últimas GPUs têm vazão próxima de 1 Tera bytes por segundo. Apesar da vazão elevada, a latência da memória é alta, podendo chegar à 800 ciclos de *clock* em comparação com uma CPU que tem uma latência na faixa de 100 ciclos.

Para mascarar a latência, a GPU usa os milhares/milhões de *threads* e registradores. Quando um *thread* faz uma requisição de memória e a instrução seguinte precisa do dado solicitado, o *thread* vai para fila de espera. Em um cenário com milhares de *threads* com troca rápida de contexto, milhares de requisições serão disparadas e isso irá esconder a latência da memória à medida em que começam a ser atendidas. Assim que chegar a primeira requisição, as requisições seguintes já estarão no caminho e tudo funcionará com uma alta vazão em pipeline. Além disso, o acesso deve ter um padrão para maximizar a vazão. Cada transação com a memória envolve uma palavra de 128 bytes ou 32 palavras de 4 bytes (32 bits). Ou seja, se cada *thread* dentro de um *warp* solicitar uma palavra de 32 bits (float ou int), os 32 *threads* do *warp* estarão solicitando 128 bytes. Se *threads* solicitarem dados consecutivos ou aglutinados (*coalesced* é o termo em inglês), o acesso pode gerar a vazão máxima, pois apenas uma transação de 128 bytes de memória será necessária para atender todos os *threads* do *warp*. Entretanto, se cada *thread* de um mesmo *warp* solicitar dados em regiões espaçadas (com distância de dezenas ou centenas de bytes) como, por exemplo, percorrer uma coluna de uma matriz (variando a linha) a vazão será reduzida. Acesso a padrões aleatórios como, por exemplo, quando cada *thread* acessa um hash geram o pior desempenho. A Figura 1.4 ilustra o exemplo do polinômio da Figura 1.1(b) onde o padrão de acesso é ideal com *threads* fazendo acessos consecutivos e aglutinados.



**Figura 1.4. Padrão aglutinado de Acesso a Memória Global, todos os *threads* do *warp* acessam posições consecutivas da memória: *thread* 0 acessa posição 512, *thread* 1 posição 516, ... (de 4 em 4 bytes).**

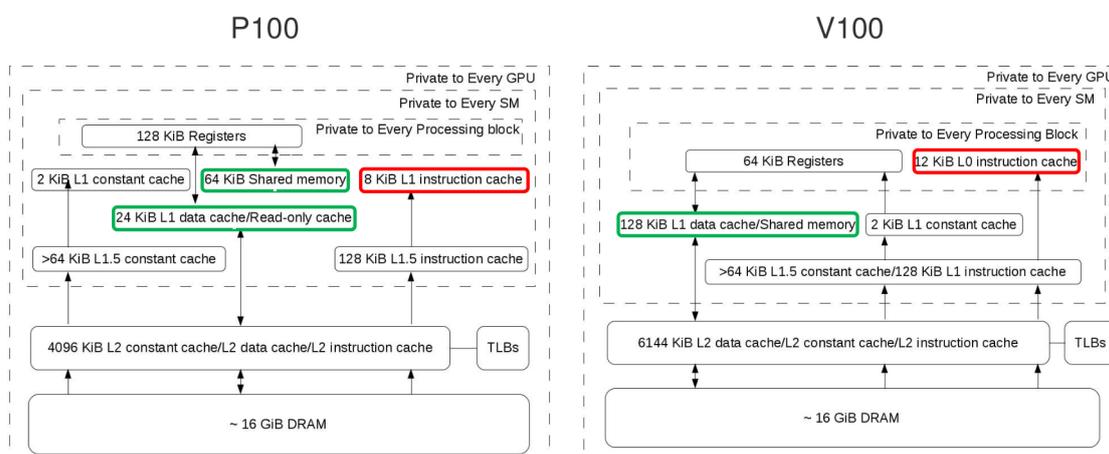
Apesar da proposta inicial das GPUs ter sido eliminar a cache para simplificar o projeto e maximizar área do *chip* para as unidades de execução, as arquiteturas da geração Fermi em diante incorporaram caches L1 e depois L2. As caches L1 variam de tamanho e cada multiprocessador tem sua própria cache privada. Um recurso interessante é controlar o tamanho da cache em tempo de execução. Algumas gerações permitem controlar o tamanho da cache e da memória compartilhada. A partir da arquitetura Kepler, um novo comando da API CUDA permite configurar dinamicamente o tamanho da L1 e da memória compartilhada (isso é ilustrado na Tabela 1.1). A última coluna ilustra um exemplo de tamanho com a arquitetura Kepler.

As novas gerações Volta e Turing adicionaram um novo recurso com a cache L0

**Tabela 1.1. Configurações de Tamanho para L1 e Memória Compartilhada com o comando `cudaDeviceSetCacheConfig ( cudaFuncCache cacheConfig )`**

Parâmetro	L1	Compartilhada	Kepler
<code>cudaFuncCachePreferNone</code>	Sem preferência	Sem preferência	
<code>cudaFuncCachePreferShared</code>	Menor	Maior	L1=16K,SM=48K
<code>cudaFuncCachePreferL1</code>	Maior	Menor	L1=48K,SM=16K
<code>cudaFuncCachePreferEqual</code>	Mesmo Tamanho	Mesmo Tamanho	L1=32K,SM=32K

para instruções devido ao aumento da codificação das instruções em binário. Em relação à memória compartilhada e cache, as gerações Pascal P100 e Volta V100 têm abordagens diferentes como ilustrado na Figura 1.5, onde a Pascal usa espaços separados e a Volta usa o mesmo espaço que pode variar o tamanho da L1 e da memória compartilhada, introduzido na arquitetura Kepler.



**Figura 1.5. Estrutura de Memória das arquiteturas Pascal P100 e Volta V100.**

Diferentemente das CPUs tradicionais, a GPU possui uma memória compartilhada (*shared memory* em inglês), que pode ser vista como uma memória cache controlada pelo programador (diferentemente da cache tradicional, que é controlada pelo hardware). Este recurso não é novidade e foi introduzido na década de 70 com o microprocessador Fairchild F8 com 64 bytes de memória local controlada por software (*scratchpad* é o termo em inglês). Nas GPUs, diferentemente da organização da cache (que é por linha, sendo de 128 bytes nas arquitetura da *Nvidia*) a memória compartilhada é organizada por bancos. Em todas as arquiteturas, a memória compartilhada possui 32 bancos. O acesso ideal é quando cada *thread* do mesmo *warp* faz acesso a um banco diferente. A Figura 1.6(a) ilustra um trecho de código sem conflito de acesso aos bancos, enquanto que a Figura 1.6(b) ilustra um exemplo com conflito, onde cada sub-grupo de 8 *threads* faz acesso ao mesmo banco de memória compartilhada. O acesso será serializado.

Um exemplo clássico de conflito é o acesso por coluna ilustrado na Figura 1.7(a), adaptado da referência [Cheng et al. 2014]. Neste exemplo, suponha a memória com cinco bancos. A Figura 1.7(a) destaca o acesso aos dados da coluna 0 (em cinza) que será concentrado no banco 0, gerando conflitos que fazem com que os acessos sejam serializados durante a execução. Este problema pode ser facilmente contornado com a

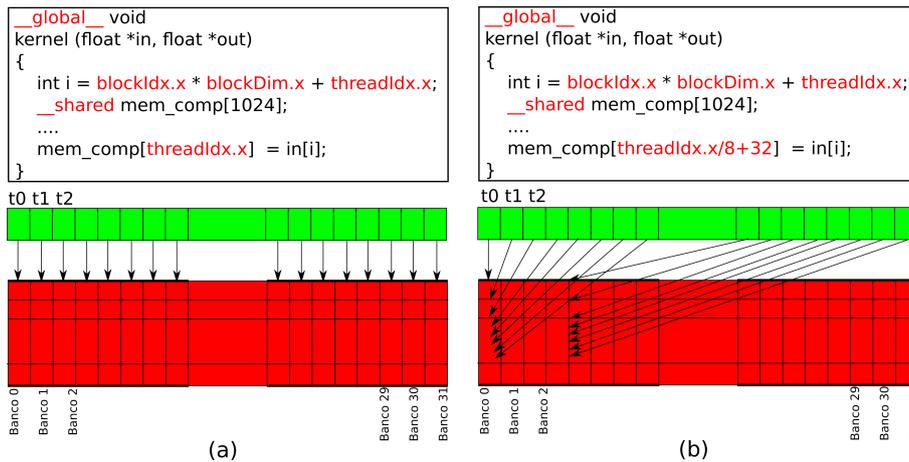


Figura 1.6. Trecho com visualização de um warp: (a) Padrão de acesso sem conflito na memória compartilhada; (b) Padrão de acesso com conflito.

técnica de "padding". Ao acrescentar uma coluna, fazendo um deslocamento dos dados como ilustrado na Figura 1.7(b), cada *thread* fará acesso a um banco diferente ao percorrer a matriz por coluna. Podemos observar que a coluna 0 em destaque pode ser acessada em paralelo, onde cada elemento está em um banco diferente. Vale ressaltar que o exemplo está simplificado para 5 bancos e pode ser diretamente estendido para 32 bancos.

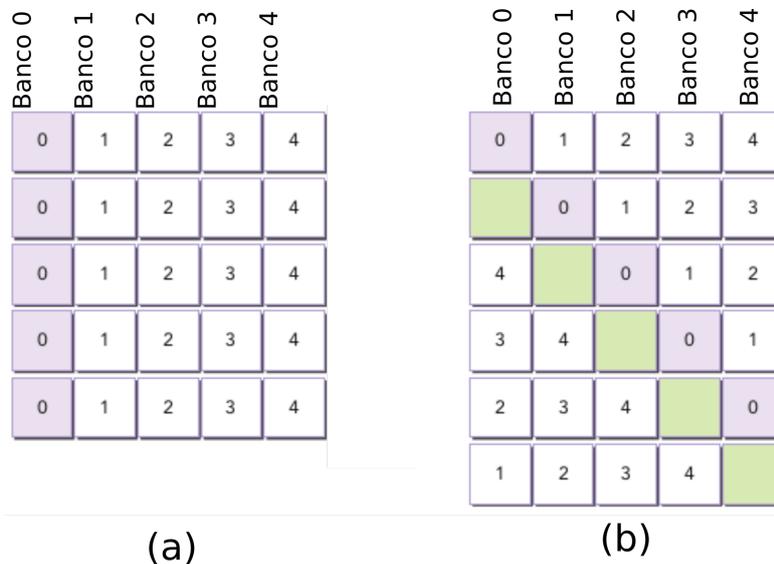


Figura 1.7. (a) Acesso com conflito na memória compartilhada para percorrer uma matriz por coluna (b) Acesso sem conflito aplicando a técnica de "padding" para percorrer por uma matriz por coluna. Figura adaptada da referência [Cheng et al. 2014].

### 1.3. Métricas

Nesta seção apresentamos inicialmente a ferramenta **nvprof**, as métricas que podem ser aferidas, como interpretar as medidas e qual a correlação delas com o código. Posteriormente, o papel do compilador e a análise do código PTX são introduzidos com exem-

plos simples e didáticos. Por exemplo, desmistificar a relação entre a ocupação e o desempenho, onde mais trabalho e poucos *threads* são capazes de gerar um desempenho maior [Volkov 2010].

### 1.3.1. Nvprof

A ferramenta **nvprof** irá executar uma aplicação realizando medidas detalhadas. Por ser uma ferramenta de linha de comando de simples configuração, pode ser bem útil para construção de *scripts* para avaliar uma aplicação. A Figura 1.8 ilustra um trecho de dados extraídos com **nvprof** para execução do exemplo de código da *Nvidia* para multiplicação de matrizes dividindo-a em blocos (*tiles*). Primeiro, a ferramenta imprime as características da GPU, neste exemplo é uma GPU de *laptop*, uma GPU 640M com 384 núcleos a 645 Mhz com um desempenho de pico de 247 Gflops. A aplicação imprime informações do tamanho das matrizes e o desempenho medido de 35,35 Gflops. O tempo de execução foi 3,7 milissegundos.

```
$ nvprof matrixMul
[Matrix Multiply Using CUDA] - Starting...
==27694== NVPROF is profiling process 27694, command: matrixMul
GPU Device 0: "GeForce GT 640M LE" with compute capability 3.0

MatrixA(320,320), MatrixB(640,320)
Computing result using CUDA Kernel...
done
Performance= 35.35 GFlop/s, Time= 3.708 msec, Size= 131072000 Ops, WorkgroupSize= 1024 t
Checking computed result for correctness: OK

Note: For peak performance, please refer to the matrixMulCUBLAS example.
==27694== Profiling application: matrixMul
==27694== Profiling result:
Time(%)   Time      Calls   Avg      Min      Max   Name
99.94%   1.11524s   301    3.7051ms  3.6928ms  3.7174ms  void matrixMulCUDA<int=32>(fl
int)
 0.04%   406.30us    2    203.15us  136.13us  270.18us  [CUDA memcpy HtoD]
 0.02%   248.29us    1    248.29us  248.29us  248.29us  [CUDA memcpy DtoH]
```

Figura 1.8. Trecho de execução com **nvprof** para a aplicação Multiplicação de Matrix disponibilizada pela *Nvidia*.

O exemplo multiplica uma matriz de 320x320 por uma matriz de 320x640. Serão executadas  $320 \times 320 \times 640 = 65.536.000$  multiplicações e adições, ou seja, aproximadamente 130 milhões de operações em 3,7 ms, que equivale a  $\frac{130M}{0.0037s} = 35$  Gflops. Agora iremos analisar as informações impressas pelo **nvprof**. Primeiro, 99,95% do tempo foi gasto com a multiplicação de matrizes. Porém, este código realiza 301 repetições (chamadas da função de multiplicação) como mostra a coluna *calls* do **nvprof**. Neste experimento, as repetições são usadas para aferir um tempo médio. Se considerarmos o tempo de uma chamada apenas, a multiplicação em 3,7ms seria responsável por 85% do tempo total de 4,35ms, onde 650us seriam necessários para transferir os dados entre da CPU e a GPU como apresentado (em azul) pelas chamadas *CUDA memcpy HtoD* (CPU para GPU) e *CUDA memcpy DtoH* (GPU para CPU).

O **nvprof** irá exibir o tempo de execução de cada kernel, das funções API CUDA e das transferências. A ferramenta também permite uma análise detalhada de várias métricas de memória e instruções como vazão, ocupação, número de operações, etc. O exemplo

abaixo executa uma aplicação e imprime a ocupação da GPU (fração de *warps* em execução pelo total de *warps* que podem executar) e o número médio de ciclos por instrução (*ipc*):

```
nvprof --metrics achieved_occupancy,ipc <app> <app args>
```

Além disso, a avaliação pode ser detalhada e específica para um *kernel* apenas:

```
nvprof --kernels<kernelspecifier> --analysis-metrics <app>
```

O **nvprof** possui muitos recursos que podem ser explorados. Vários exemplos de uso do **nvprof** são apresentados na referência [Cheng et al. 2014]. Um fato importante é evitar de fazer análises isoladas de uma métrica. Vale lembrar que é sempre bom ter o desempenho ou tempo de execução de uma versão da implementação como uma referência básica. Um exemplo ilustrado em [Cheng et al. 2014] é a transposição de matrizes onde a versão que faz a leitura da matriz original por coluna e escreve por linha apresenta uma vazão de leitura (*gld* = global load) na ordem de 600 GB/s aferida com o comando:

```
nvprof --devices 0 --metrics gld_throughput ./transpose
```

porém, o acesso por coluna não é eficiente que é verificado ao acionar a métrica *metrics gld\_efficiency* que mostra que apenas 6% de cada 128 bytes (uma transação de memória global) são efetivamente utilizados devido ao acesso por coluna.

Além do **nvprof**, a *Nvidia* disponibiliza várias outras ferramentas como o *Nvidia Visual Profile* com interface gráfica e análise. Porém é recomendado um bom entendimento da arquitetura e das métricas para avaliar as aplicações. A Figura 1.9 ilustra uma tela com a análise dos gargalos de uma aplicação, onde a ferramenta visual apresenta as frações para cache L2, memória e operações aritméticas. Para o exemplo podemos observar a recomendação que a aplicação está limitada pelo acesso aos dados na cache L2.

A Figura 1.10 ilustra outra tela da ferramenta Visual Profile, com um diagrama detalhado com a quantidade em milhões de instruções de memória executadas para os diversos níveis e tipos de memória da GPU.

Com as novas GPUs a partir da geração Pascal e da rede de interconexão de **nvlink** com alto desempenho, as ferramentas visuais também auxiliam na otimização de código com múltiplas GPUs como ilustrado na Figura 1.11. A taxa de pico pode chegar a 100 GB/s entre as GPUs, que é bem superior a taxa de 10 GB/s do barramento PCI.

### 1.3.2. Assembly PTX

A *Nvidia* disponibiliza uma pseudo linguagem assembly PTX (Parallel Thread Execution). O compilador **nvcc** traduz o código C++/CUDA em PTX, que depois é traduzido para binário para executar na GPU. Além de um formato intermediário, o PTX é legível possibilitando ao programador várias informações sobre o código gerado. Um detalhe importante é que o PTX usa um número arbitrário de registradores que depois será otimizado pelo compilador. Por exemplo, o comando:

### i Kernel Performance Is Bound By Memory Bandwidth

For device "Quadro K6000" the kernel's compute utilization is significantly lower than its memory utilization. These utilization levels indicate that the performance of the kernel is most likely being limited by the memory system. For this kernel the limiting factor in the memory system is the bandwidth of the L2 Cache memory.

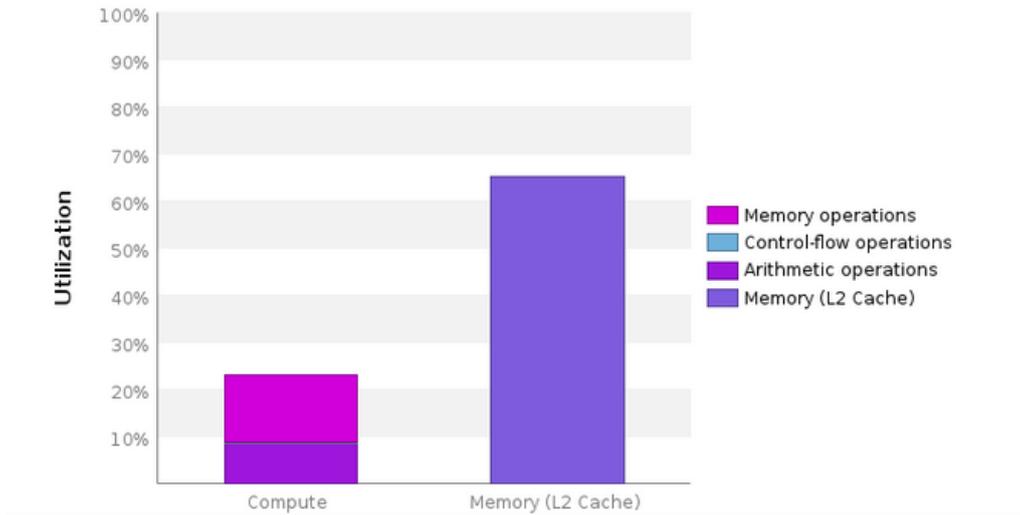


Figura 1.9. Análise de Dados de utilização da Cache L2, Memória e operações aritméticas

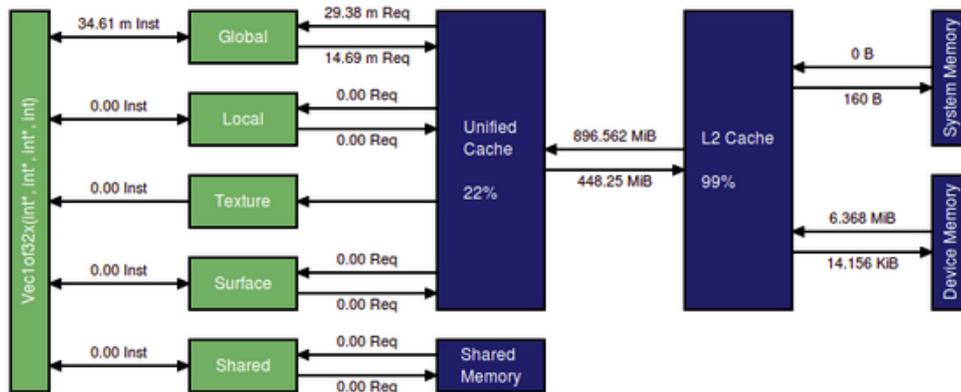


Figura 1.10. Diagrama com os diversos níveis de memória da GPU e o número de operações executadas.

```
.reg .u32 %r<51>;
```

declara 51 registradores de inteiros sem sinal com 32 bits cada. É possível identificar facilmente as operações com a memória global, instruções condicionais, etc. Por exemplo, o trecho a seguir:

```
ld.global.f32 %f2, [%rd8];
add.f32 %f3, %f2, %f1;
st.global.f32 [%rd10], %f3;
```

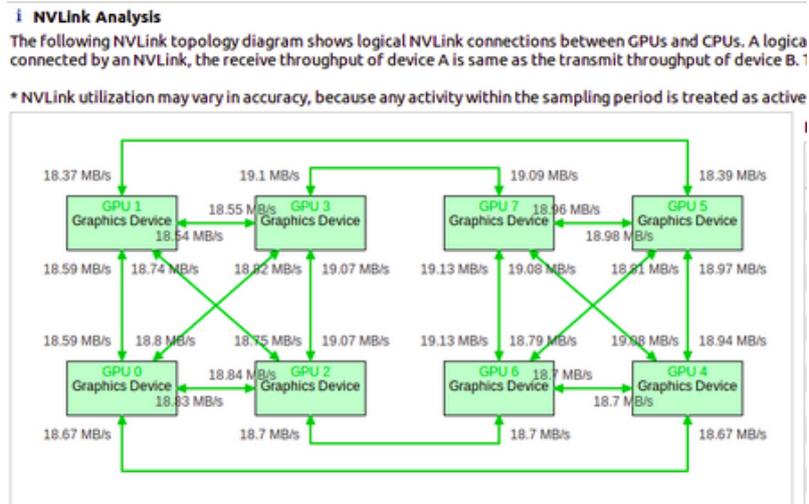


Figura 1.11. Taxa de troca de dados entre GPUs com uma rede de interconexão nvlínk.

executa uma leitura (*load*) na memória global para o registrador *f2* da posição apontada pelo registrador *rd8*. A instrução *add* executa a soma  $f3 = f1 + f2$ . Finalmente a instrução *st* (*store*) grava o valor de *f3* na memória global no endereço apontado pelo registrador *rd10*. Apesar de simples, o trecho pode demorar muitos ciclos para executar. A operação de leitura pode demorar até 800 ciclos, portanto o *thread* irá para fila de espera, caso o dado não esteja na cache. Mesmo na cache, como veremos na Seção 1.4.1, a leitura pode demorar alguns ciclos. A instrução de soma pode demorar vários ciclos também (como será detalhado na Seção 1.4.2) e finalmente a instrução de escrita do resultado na memória. Ou seja, apesar de pequeno, o trecho ilustrar dependência de dados e de operações de memória.

Ao compilar com o parâmetro `-ptx`, o `nvcc` irá gerar um arquivo com a extensão `PTX` onde podemos visualizar o código gerado. Além disso, o `PTX` pode ser usado para inserir código e/ou instrumentar trechos de código para realizar medidas como iremos ilustrar a seguir.

Primeiro iremos ilustrar um exemplo com assembly `PTX` para otimizar uma operação simples na execução da expressão da aplicação `SAPXY`. O exemplo foi proposto em [Jia et al. 2019] para motivar a importância de otimizações mesmo nas últimas gerações de GPU. O exemplo ilustrado na Figura 1.12 mostra que podemos ter um ganho significativo de desempenho na arquitetura Turing ao agrupar a leitura de dados de 4 em 4 valores com as instruções vetoriais `load.global.v4` e `store.global.v4`. O código é simples e executa  $y[i] = x[i] * \alpha + y[i]$ . Cada *thread* irá ler 4 elementos dos vetores *x* e *y*, executar a operação localmente nos registradores *a, b, c* e *d* no exemplo com uma única operação `FMA` (multiplica e soma) por elemento e finalmente gravar o resultado agrupado com o comando `store.global.v4`. Podemos observar que o código mescla instruções em `PTX` e em `C/C++`. A comunicação é feita pelos parâmetros `%0`, `%1` e `%2` que são respectivamente os ponteiros para os vetores *x[i]* e *y[i]* e a variável *alpha*.

A Figura 1.13 mostra que a versão em `PTX` com as instruções explícitas para ler/escrever agrupando de 4 em 4 elementos por *thread* geram um desempenho 2x me-

```

__global__ void improved_Saxpy( float *d_y, const float *d_x,
                               const float alpha, const uint32_t arraySize)

// every thread process 4 elements at a time
uint32_t tid = (threadIdx.x+blockIdx.x*blockDim.x)*4;
// the elements that all threads on GPU can process at a time
uint32_t dim = blockDim.x*blockDim.x*4;

for(uint32_t i = tid; i < arraySize; i += dim)
asm volatile ("{\t\n"
// registers to store input operands
".reg .f32 a1,b1,c1,d1;\n\t" ← 4 registradores para vetor x
".reg .f32 a2,b2,c2,d2;\n\t" ← 4 registradores para vetor y

// loading with vectorized, 128-bit instructions
"ld.global.v4.f32 {a1,b1,c1,d1}, [%0];\n\t" ← leitura agrupada de 4 valores
"ld.global.v4.f32 {a2,b2,c2,d2}, [%1];\n\t"

// core math operations
"fma.rn.f32 a2,a1,%2,a2;\n\t" ← a2 = a1*%2+a2 onde
"fma.rn.f32 b2,b1,%2,b2;\n\t" ← a1 elemento vetor x
"fma.rn.f32 c2,c1,%2,c2;\n\t" ← a2 elemento vetor y e
"fma.rn.f32 d2,d1,%2,d2;\n\t" ← %2 o parametro a.

// storing results with a vectorized, 128-bit write instruction
"st.global.v4.f32 [%1], {a2,b2,c2,d2};\n\t" ← grava agrupado no vetor y

```

Figura 1.12. Trecho de código extraído da referência [Jia et al. 2019] com destaques ilustrando um código misto C/C++ e Assembly PTX para a aplicação Saxpy.

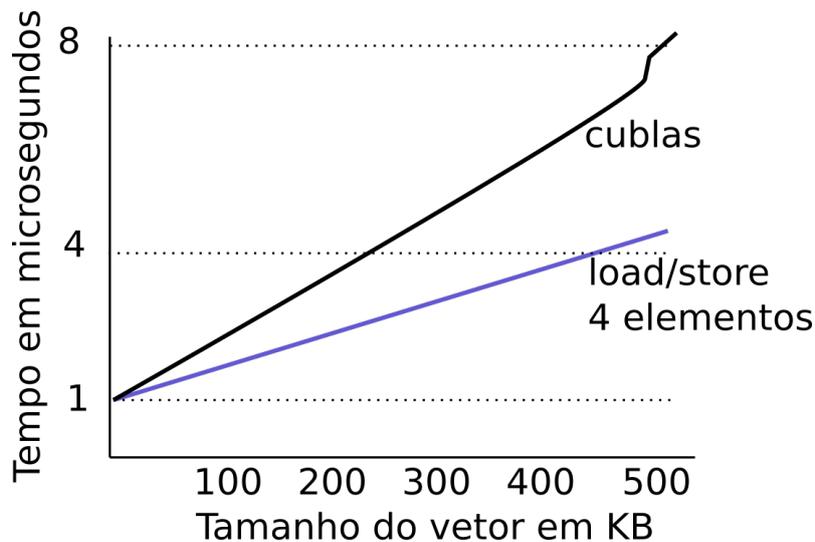


Figura 1.13. Tempo de execução para a aplicação Saxpy comparando a biblioteca otimizada Cublas com um trecho com o código misto PTX/C que explora a vetorização com 4 elementos por threads por vez. O resultado foi extraído e adaptado da referência [Jia et al. 2019].

Ihor em comparação com a versão otimizada da biblioteca CUBLAS de álgebra linear da Nvidia na arquitetura Turing. Resumindo, este exemplo demonstra uma oportunidade de otimização acessível aos programadores que possuem um conhecimento de PTX e um entendimento no nível de arquitetura das gerações de GPU. Maiores detalhes de como explorar estes recursos estão ilustrados em [Jia et al. 2019].

Outro recurso interessante é medir a latência de execução de uma instrução ou

de um trecho de código para cada *thread* em execução. Esta medida é interessante para comprovar a latência real assim como para estimar o número de *threads* necessários para esconder a latência com a sobreposição concorrente das execuções. Este recurso é muito utilizado em microbenchmark [Arafa et al. 2019] que são exemplos de testes para avaliação isolada de instruções, pequenos trechos ou recursos das arquiteturas.

Para medir o tempo de um trecho podemos coletar o *clock* da GPU antes e depois do trecho, calcular a diferença e teremos o número de ciclos de *clock* gastos por cada *thread*. A Figura 1.14(a) ilustra um exemplo simples com dois vetores A e B de float com o código instrumentado para medir o tempo em ciclos da operação *add*. As linhas 5 e 7 fazem a leitura do *clock* antes e depois da execução do *add* e armazenam nas variáveis locais *c1* e *c2*. O vetor C irá retornar a latência em ciclos de *clock* para cada *thread*.

```

__global__ void kernel(float *A, float *B, unsigned int *C, const int N)
{
    1 float a;
    2 int idx = blockIdx.x * blockDim.x + threadIdx.x;
    3 unsigned int c1,c2;
    4 a = A[idx]; // load

    5 asm("mov.u32 %0,%%clock;" : "=r"(c1)); // comeco
    6 asm("add.f32 %0,%1,%2;" : "=f"(a): "f"(a),"f"(a)); // a=a+a , add
    7 asm("mov.u32 %0,%%clock;" : "=r"(c2)); // fim

    8 if ( idx < N) { B[idx] = a; C[idx] = c2-c1; } // b[i]=2a[i]
}

```

(a)

```

__global__ void kernel(float *A, float *B, unsigned int *C, const int N)
{
    1 float a;
    2 int idx = blockIdx.x * blockDim.x + threadIdx.x;
    3 unsigned int c1,c2;
    4 a = idx; // evitar o tempo do load

    5 asm("mov.u32 %0,%%clock;" : "=r"(c1)); // comeco
    6 asm("add.f32 %0,%1,%2;" : "=f"(a): "f"(a),"f"(a)); // a=a+a , add
    7 asm("mov.u32 %0,%%clock;" : "=r"(c2)); // fim

    8 if ( idx < N) { B[idx] = A[idx]+A[idx]; C[idx] = c2-c1; }
}

```

(b)

**Figura 1.14. Trecho de código com instrumentação para medida de tempo da instrução de soma de ponto flutuante de 32 bits: (a) Versão ingênua; (b) Versão sem a latência da memória.**

O valor esperado para a latência do *add* é em torno de 10-20 ciclos. Porém o valor medido é da ordem de 600 a 800 ciclos. Este fato é justificado pois ao disparar a leitura do valor na linha 4, a instrução *load* é não bloqueante. A próxima instrução é a leitura do *clock* para o registrador *c1*, que não depende do *load* e irá executar logo após a requisição de *load* ser disparada. Na linha 6, antes de executar, o *thread* irá aguardar o *load* e só retornará depois que o dado já estiver no registrador para calcular a expressão  $a + a$  com a instrução *add*. Portanto a medida do *clock* na linha 7 irá incluir a latência do *load* e do *add*. Para contornar este problema e fazer a medida apenas da latência do *add*, podemos

usar a versão ilustrada na Figura 1.14(b), onde usamos o número do *thread* para o valor de *a*, medimos a latência do *add* e depois executamos a operação  $2 * A[i]$  para gravar no vetor.

No código binário, além das instruções as GPUs da *Nvidia* utilizam bits para armazenar código de controle. São 23 bits nas gerações *Volta* e *Turing* [Jia et al. 2019]. As palavras de controle foram introduzidas na geração *Kepler* para substituir as técnicas de escalonamento dinâmico empregadas nos processadores, uma vez que as GPUs fazem a execução em ordem com escalonamento estático. Estas informações irão auxiliar a unidade de controle de execução (escalonador de *warps*) da GPU, sem complicar o hardware, deixando o trabalho para o compilador. O código binário da *Turing* e da *Volta* possui 128 bits incluindo as informações de controle. Estas informações são divididas em campos. O campo de reuso é usado para evitar conflito no banco de registradores, salvando valores para uso nas próximas instruções. Campos de barreiras de espera, escrita e leitura são usados para auxiliar o escalonador com relação a latência das instruções que serão executadas para planejar a troca de contexto dos *warps*. O campo de barreiras de dependências de leitura servem para evitar conflitos de escritas após leituras. O campo parada (*stall*) indica quantos ciclos o escalonador deve esperar para retornar para executar a próxima instrução do *warp* corrente. Finalmente o campo *yield*, com um bit apenas, serve para dizer se o escalonador deve manter o *warp* em execução ou realizar a troca de contexto.

Para auxiliar no ensino e pesquisa com assembly PTX, a ferramenta *Compiler Explorer* de compilação disponível no navegador pode ser usada [Godbolt 2019]. Como ilustra a Figura 1.15 do lado esquerdo temos o trecho de código com cores e do lado direito o assembly PTX gerado também colorido para mostrar o mapeamento do código de alto nível no código assembly.

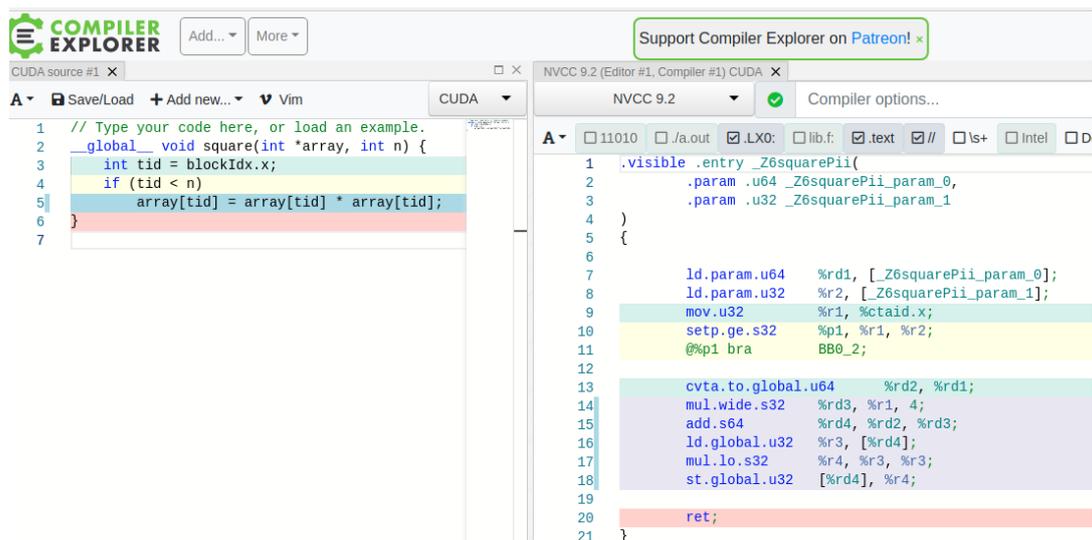


Figura 1.15. Tela do navegador com a Ferramenta Compiler Explorer disponível para ensino e pesquisa de compiladores que incluem versões do NVCC 9 e 10.

## 1.4. Latência e Vazão

A Seção 1.4.1 apresenta detalhes para compreender as restrições das classes de aplicações com desempenho limitado pela memória e/ou processamento [Serpa et al. 2019, Shekofteh et al. 2019]. Estudos recentes [Jia et al. 2019, Mei and Chu 2016] são apresentados na Seção 1.4.1 e mostram a latência dos vários níveis de memória das gerações recentes (Turing, Volta, Pascal, Maxwell). Com relação as operações, a Seção 1.4.2 destaca o nível de instrução e a grande variação na latência do conjunto de instruções das gerações aferidas com precisão [Arafa et al. 2019].

### 1.4.1. Memórias

**Tabela 1.2. Latência para diversas situações com e sem falhas de acesso nas caches L1, L2 e nas TLBs.**

Padrão	$P_1$	$P_2$	$P_3$	$P_4$	$P+5$	$P_6$
L1	acerto	acerto	acerto	falha	falha	falha
L1 TLB	acerto	falha	falha	acerto	falha	falha
L2 TLB	-	acerto	falha	-	acerto	falha
Latência						
Maxwell 980 L1	82	-	-	385	2439	2740
Sem L1	214	225	289	383	2461	2750
Kepler 780	198	204	257	339	702	968
Fermi 560 L1	96	384	468	635	1239	-
Sem L1	351	378	462	619	1225	-

Em [Jia et al. 2019, Jia et al. 2018, Mei and Chu 2016] são apresentados estudos detalhados das arquiteturas Turing T4, Volta, Maxwell, Kepler e Fermi com *microbenchmarks*, avaliando as latências dos diversos níveis de memória. O trabalho apresentado em [Mei and Chu 2016] introduziu uma análise detalhada da latência nos diversos níveis de cache. A Tabela 1.2, adaptada de [Mei and Chu 2016], mostra as latências para diversas situações  $p_1, \dots, p_6$  que incluem acertos (hit) e falhas de acessos (miss) na cache de dados L1 e L2 bem como na cache de endereços (TLB) de nível 1 e 2. Podemos observar que existe uma grande variação na latência dos acessos.

**Tabela 1.3. Tabela adaptada de [Jia et al. 2019] para a latência aferida com *microbenchmarks* para as arquiteturas Turing, Volta, Pascal e Kepler.**

Arquitetura Placa	Turing T4	Volta V100	Pascal P100	Pascal P4	Maxwell M60	Kepler K80
<i>clock</i> Ghz	1.5	1.4	1.3	1.5	1.2	0.9
Multiprocessadores	40	80	56	40	16	13
Latência Hit L1	32	28	82	82	82	35
Tamanho linha L1	32	32	32	32	32	128
Latência Hit L2	188	193	234	216	207	200

A Tabela 1.3 mostra algumas medidas de latências e outras características ilustrando pequenas e grandes diferenças entre as últimas gerações de GPU com dados extraídos com *microbenchmarks* nos experimentos realizados em [Jia et al. 2019]. Por exemplo, a latência do acerto (hit) é em torno de 30 ciclos para a Turing T4 e para Volta V100. Já para a Pascal e Maxwell, a latência da L1 é de 82 ciclos. Entretanto, a latência da K80, uma GPU bem mais antiga é de 35 ciclos. Uma mudança da K80 em relação às GPUs mais novas é o tamanho da linha de cache que passou de 128 bytes para 32 bytes. Com relação à latência da cache L2, a variação é pequena, assim como para maioria das outras medidas apresentadas pelos experimentos que estão detalhados em [Jia et al. 2019].

```

__global__ void l2_bw(float *dsink, uint32_t *posArray)
{
    // block and thread index
    UINT tid = threadIdx.x;
    UINT bid = blockIdx.x;

    // accumulator; side effect to prevent code elimination
    float sink = 0;

    // load data from l2 cache and accumulate
    for (UINT i = 0; i < L2_SIZE; i += THREADS_NUM) {
        DTYPE* ptr = posArray+i;

        // every warp loads all data in l2 cache
        for (UINT j=0; j < THREADS; j += 32) {
            UINT offset = (tid+j)%THREADS;
            asm volatile ("{\t\n"
                ".reg .f32 data;\n\t"
                ".ld.global.cg.f32 data, [%1];\n\t"
                "add.f32 %0, data, %0;\n\t"
                "}" : "+f"(sink) : "l"(ptr+offset) : "memory"
            );
        }
    }
    // side effect: store the result
    dsink[tid] = sink;
}

```

(a)

Vazão em GB/s da Cache L2 para as arquiteturas

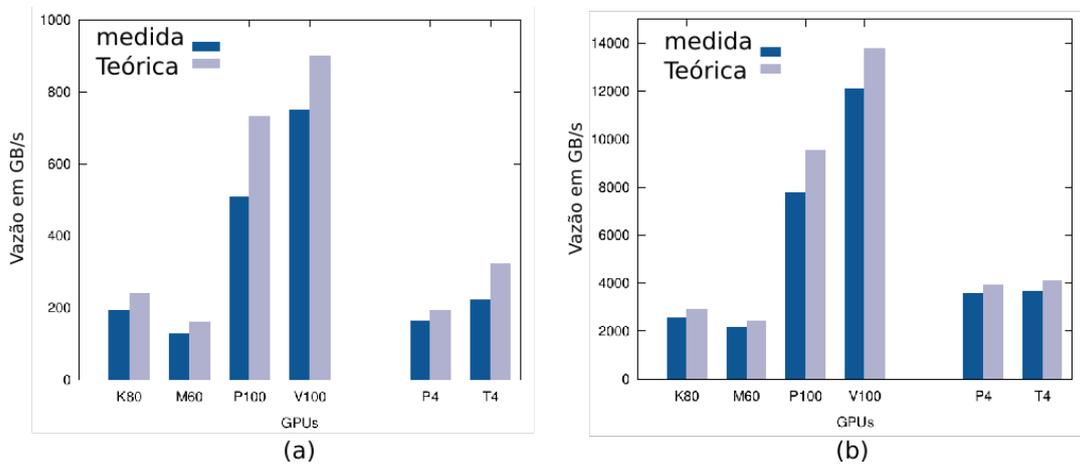
Turing	Volta	P100	P4	Maxwell	Kepler K80
1270	2150	1600	979	446	339

(b)

Figura 1.16. Trecho de código com instrumentação para medida da vazão da cache L2 em diversas arquiteturas proposto em [Jia et al. 2019] (a) Microbenchmark (b) Vazão em GB/s.

A Figura 1.16 ilustra um exemplo de microbenchmark proposto em [Jia et al. 2019]

para avaliação da latência no nível L2 de cache. O código inclui trechos mistos em C/C++ e assembly PTX. A instrução `ld.global.cg.f32` que irá realizar a leitura de memória. Note que é necessário adicionar o vetor `sink` para evitar que o compilador remova código e gere erros nas medições. Este detalhe é importante na instrumentação dos códigos. Além deste exemplo, a referência [Jia et al. 2019] apresenta uma avaliação detalhada das caches L1, caches de constantes, memória global e compartilhada, acesso concorrente na memória compartilhada, operações atômicas, cache de instruções, banco de registradores, latência de instruções e desempenho dos novos operadores tensores. Por exemplo, a Figura 1.17 mostra a diferença entre a vazão teórica e a vazão medida para memória global e compartilhada aferida em [Jia et al. 2019]. Podemos observar que as gerações Pascal e Volta aumentaram a vazão da memória global da faixa de 200/300 GB/s para 600/800 GB/s. Com relação a memória compartilhada, também existe um ganho significativo chegando a taxa de 12 Tera bytes por segundo na geração Volta para a GPU V100.



**Figura 1.17. (a) Vazão em GB/s da Memória Global (b) Vazão em GB/s da Memória Compartilhada. Experimentos realizados em [Jia et al. 2019].**

A Tabela 1.4 foi extraída da referência [Jia et al. 2019] com as latências para acesso com operações atômicas com e sem conflitos de acesso (competição pela mesma posição) com várias configurações de *threads* para diversas arquiteturas: Turing T4, Volta V100, Pascal P100 e P4, Maxwell M60 e Kepler K80.

**Tabela 1.4. Latência das operações atômicas em cenários com acesso em presença de conflitos com 2 a 32 threads nas memórias global e compartilhada para diversas gerações de GPU. A Tabela foi extraída da referência [Jia et al. 2019].**

Contention	Shared memory						Global memory					
	T4	V100	P100	P4	M60	K80	T4	V100	P100	P4	M60	K80
none	8	6	15	16	17	93	76	36	26	30	24	29
2 threads	10	7	17	18	19	214	72	31	31	50	26	69
4 threads	14	11	19	25	25	460	73	32	48	50	41	96
8 threads	22	18	30	30	31	952	81	41	48	51	41	152
16 threads	37	24	46	46	47	1,936	97	58	50	51	46	264
32 threads	69	66	78	78	79	4,257	116	76	50	51	46	488

Podemos observar que a operação atômica tem uma pequena latência para as arquiteturas mais novas no cenário sem conflito em comparação com a arquitetura Kepler K80 para memória compartilhada. À medida que 2 ou mais *threads* fazem a operação atômica na mesma posição, a latência vai aumentando mas não chega a ser 32x maior no pior caso, exceto para a K80 onde a latência supera os 4000 ciclos de *clock*. Em relação à memória principal, a arquitetura T4 apresenta uma latência até maior que a K80, mas à medida que os conflitos com mais *threads* aumentam, a latência da T4 fica menor que a latência da K80.

Trabalhos recentes fazem avaliação de memória com simulação sem perder a precisão e utilizando *microbenchmarks*, como por exemplo o modelo proposto para o sistema de memória da arquitetura Volta implementado no simulador GP-SIM apresentado em [Khairy et al. 2018]. O trabalho proposto em [Chen et al. 2019] apresenta um modelo sistólico com deslocamento e somas parciais que utiliza o banco de registradores como cache para acelerar aplicações com cálculo de *stencil* e convoluções que explora os mecanismos de cache L1, L2 e banco de registradores para gerar uma aceleração de 2,5x em relação a biblioteca NPP.

#### 1.4.2. Instruções

Nesta seção iremos apresentar alguns resultados de trabalhos recentes na instrumentação de códigos com *microbenchmarks* [Jia et al. 2019, Jia et al. 2018, Arafa et al. 2019]. O uso de *microbenchmarks* já foi proposto desde as primeiras gerações [Wong et al. 2010], mas nesta seção iremos destacar apenas as últimas gerações.

A Tabela 1.5 foi extraída da referência [Arafa et al. 2019] e foi resumida para ilustrar a variação das latências para as diversas arquiteturas e tipos de instruções. A Tabela mostra o número de ciclos com e sem otimização (otim. com a opção -O3/não otim. com a opção -O0) para a latência na execução das instruções de apenas três classes (inteiros, precisão simples ou float, precisão dupla ou double) e três operações add, mult e div. Primeiro, podemos observar que praticamente não existe diferença na latência entre as operações com inteiros e com precisão simples. Com relação à precisão dupla há apenas uma diferença nas arquiteturas Titan X e RTX, que possuem poucas unidades de precisão dupla (o que gera uma latência 4-5x maior). Já a operação de divisão tem uma grande latência. Instruções lógicas e aritméticas, com precisão de 16 bits, instruções especiais (raiz quadrada, cosseno, etc.) dentre outras estão todas detalhadas na referência [Arafa et al. 2019].

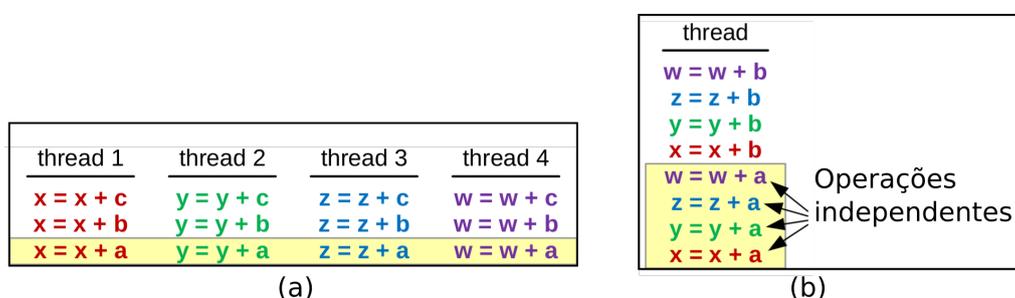
#### 1.4.3. Mais Trabalho com Redução dos Threads

As latências podem ser mascaradas com milhares de *threads* ou com a redução dos *threads* mais aumento da carga de trabalho por *thread*. Por exemplo, no lugar de cada *thread* avaliar uma entrada do vetor para o *kernel* de polinômio ilustrado na Figura 1.1, podemos avaliar 2 ou 4 entradas e melhorar o desempenho. Esta ideia foi introduzida por Volkov em [Volkov 2010]. A Figura 1.18 ilustra a ideia básica, onde ao invés de ter um código com pouco paralelismo a nível de instrução (ILP) da Figura 1.18(a) devemos buscar uma implementação com mais paralelismo como ilustrado no trecho da Figura 1.18(b).

Ao executar mais trabalho nos threads, reduzimos a proporção de código e variá-

**Tabela 1.5. Tabela adaptada de [Arafa et al. 2019] para a latência das instruções aferida com *microbenchmarks* para as arquiteturas Turing, Volta, Pascal e Kepler.**

Instrução	K80	TitanX	P100	V100	RTX
add int	9/16	6/15	6/15	4/15	4/15
mult int	9/16	13/87	13/85	4/15	4/15
div int	134/791	141/1020	144/1039	125/815	111/785
add float	9/16	6/15	6/15	4/15	4/15
mult float	9/16	6/15	6/15	4/15	4/15
div float	151/621	135/725	167/671	123/638	152/546
add double	10/16	48/52	8/15	8/15	40/48
mult double	10/16	48/52	8/15	8/15	40/54
div double	445/1588	709/1821	545/1399	159/945	540/1202



**Figura 1.18. (a) Código sem paralelismo a nível de instrução (b) Código com paralelismo e mais trabalho por *thread*. Figura adaptada da referência [Volkov 2010].**

veis de controle por *thread* em relação às operações com dados. Exemplos de controle são as variáveis de índice, os controles de laços *for*, etc. A Tabela 1.6 mostra o impacto no uso de registradores e no desempenho quando o algoritmo de multiplicação de matrizes com diversas versões onde são avaliados 1, 2, 4, 8 e 36 pontos por *thread*.

**Tabela 1.6. Experimentos introduzidos em [Volkov 2010] na arquitetura Fermi para aumentar o desempenho com a redução dos *threads* e mais trabalho com ILP por *thread*.**

Pontos por Thread	Registradores	Desempenho	Ocupação	Blocos por Multiprocessador
1	21	241 GFlops/s	67%	1
2	28	341 GFlops/s	67%	1
4	41	427 GFlops/s	50%	3
8	63	485 GFlops/s	33%	4
36	63	838 GFlops/s	33%	2

O resultado final é um ganho de 4x em desempenho com a técnica graças ao melhor uso dos registradores por *thread*, mesmo com baixa ocupação de *warps*. O aumento das oportunidades de paralelismo a nível de instrução reduz as trocas de contexto necessárias para mascarar as latências das operações. A Figura 1.19 ilustra a modificação na

implementação da multiplicação de matrizes para executar o cálculo de dois elementos por *thread*, destacadas em vermelho. Podemos notar que parte do código não é duplicada e apenas 28 registradores são usados em comparação com a versão com um elemento por *thread* que faz uso de 21 registradores. Ou seja, o desempenho melhora de 241 para 341 GFlops, sem muito impacto no aumento do número de registradores necessários, que poderia dobrar, mas só requer um aumento de 30%.

```
float Csub[2] = {0,0}; //array is allocated in registers
for (int a = aBegin, b = bBegin; a <= aEnd;
     a += aStep, b += bStep)
{
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
    AS(ty, tx) = A[a + wA * ty + tx];
    BS(ty, tx) = B[b + wB * ty + tx];
    AS(ty+16, tx) = A[a + wA * (ty+16) + tx];
    BS(ty+16, tx) = B[b + wB * (ty+16) + tx];
    __syncthreads();
    #pragma unroll
    for (int k = 0; k < BLOCK_SIZE; ++k)
    {
        Csub[0] += AS(ty, k) * BS(k, tx);
        Csub[1] += AS(ty+16, k) * BS(k, tx);
    }
    __syncthreads();
}
int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
C[c + wB * ty + tx] = Csub[0];
C[c + wB * (ty+16) + tx] = Csub[1];
```

Figura 1.19. Código com dois elementos calculados por *thread* para a multiplicação de matrizes. Figura adaptada de [Volkov 2010].

## 1.5. Novos Operadores e Computação Aproximada

Desde as primeiras gerações, as GPUs oferecem várias opções de funções aritméticas com precisão variada e desempenho variado. Por exemplo, a função `__powf(x,i)` e a função `powf(x,i)` para calcular  $x^i$ . Na versão CUDA 5.0, a função `__powf(x,i)` é implementada com 17 instruções PTX em comparação com a função `powf(x,i)` que requer 344, que impacta em um tempo de execução 24x mais lento. Em termos de precisão, a diferença é na sétima casa decimal [Cheng et al. 2014].

Com a introdução de operadores de grão grosso capazes de executar uma multiplicação e soma de uma matriz 4x4 com números na representação de meia-precisão (16 bits) em um único *clock* (em pipeline), as novas GPUs a partir da geração Volta trouxeram um impacto significativo no desempenho para aplicações de aprendizado de máquina. Além disso, a GPU Turing pode operar com números de 4 ou 8 bits, obtendo um desempenho acima de 100 Tera operações por segundo, podendo chegar em alguns casos a até 500 Tera operações por segundo. Em [Jia et al. 2019], a comparação entre a Turing e a geração Pascal mostra que para 16 bits, a Turing tem um desempenho de 41 Tera Flops (contra 6 Tera Flops da Pascal. Para números de 8 bits, a Turing tem um desempenho de 75 Tera Flops contra 24 Tera Flops da Pascal, além das possibilidades de executar operações com 4 bits e 1 bit de forma vetorizada, que não são possíveis na Pascal, gerando um desempenho de 114 e 552 Tera Flops, respectivamente.

O trabalho [Lew et al. 2019] apresenta o uso de simuladores de GPU para avaliar o desempenho com cargas de trabalho de aprendizado de máquinas, que podem auxiliar no desenvolvimento de novas arquiteturas para aumentar ainda mais o desempenho no processamento de redes neurais profundas. Em [Raihan et al. 2019], o desempenho dos operadores tensores das gerações Volta e Turing é avaliado para aplicações de aprendizado de máquina com *microbenchmarks* e simulação. Um modelo para os operadores tensores é proposto e validado com 99.6% de precisão em comparação com a execução na GPU Volta.

Os operadores tensores também podem ser usados para executar outras operações como mostra o trabalho [Dakkak et al. 2019] que propõe a implementação eficiente das operações de redução e varredura (*scan*) usando os operadores tensores que alcança de 89% à 98% do desempenho de pico para cópia na memória e é ordens de magnitude (100 para redução e 3 para varredura) mais rápido em comparação com o estado da arte para segmentos pequenos, comuns em aplicações de aprendizado de máquina. A implementação apresenta uma economia de energia de até 22% para redução e 16% para a varredura.

## 1.6. Considerações Finais

Este minicurso tem como proposta desmistificar as arquiteturas de GPU, suas métricas e técnicas que foram bem sucedidas para gerar código com desempenho. O conhecimento em detalhe da arquitetura e do assembly PTX podem auxiliar os programadores a extrair o máximo de desempenho. Compreender bem a hierarquia de memória, suas latências e vazão é importante para modelar as estruturas de dados que serão utilizadas na implementação. O uso de *microbenchmarks* e das ferramentas de profile como **nvprof** auxiliam no entendimento dos gargalos. Este texto também incluem várias referências recentes [Jia et al. 2019, Jia et al. 2018, Arafa et al. 2019] que fazem uma análise detalhada das GPUs.

Finalmente, como já mencionado na introdução, para motivar os estudantes e profissionais, mesmo sem acesso direto a uma máquina com uma GPU da *Nvidia*, os experimentos propostos para este minicurso poderão ser executados na plataforma online Colab da Google. Maiores informações podem ser encontradas no Apêndice A.

## Referências

- [Arafa et al. 2019] Arafa, Y., Badawy, A.-H., Chennupati, G., Santhi, N., and Eidenbenz, S. (2019). Instructions' latencies characterization for nvidia gpgpus. *arXiv preprint arXiv:1905.08778*.
- [Chen et al. 2019] Chen, P., Wahib, M., Takizawa, S., Takano, R., and Matsuoka, S. (2019). A versatile software systolic execution model for gpu memory-bound kernels. *arXiv preprint arXiv:1907.06154*.
- [Cheng et al. 2014] Cheng, J., Grossman, M., and McKercher, T. (2014). *Professional Cuda C Programming*. John Wiley & Sons.
- [Dakkak et al. 2019] Dakkak, A., Li, C., Xiong, J., Gelado, I., and Hwu, W.-m. (2019). Accelerating reduction and scan using tensor core units. In *Proceedings of the ACM*

*International Conference on Supercomputing*, pages 46–57. ACM.

- [Godbolt 2019] Godbolt, M. (2019). Compiler explorer. <https://godbolt.org/>.
- [Jia et al. 2019] Jia, Z., Maggioni, M., Smith, J., and Scarpazza, D. P. (2019). Dissecting the nvidia turing t4 gpu via microbenchmarking. *arXiv preprint arXiv:1903.07486*.
- [Jia et al. 2018] Jia, Z., Maggioni, M., Staiger, B., and Scarpazza, D. P. (2018). Dissecting the nvidia volta gpu architecture via microbenchmarking. *arXiv preprint arXiv:1804.06826*.
- [Khairy et al. 2018] Khairy, M., Akshay, J., Aamodt, T., and Rogers, T. G. (2018). Exploring modern gpu memory system design challenges through accurate modeling. *arXiv preprint arXiv:1810.07269*.
- [Kongetira et al. 2005] Kongetira, P., Aingaran, K., and Olukotun, K. (2005). Niagara: A 32-way multithreaded sparc processor. *IEEE micro*, 25(2):21–29.
- [Lew et al. 2019] Lew, J., Shah, D. A., Pati, S., Cattell, S., Zhang, M., Sandhupatla, A., Ng, C., Goli, N., Sinclair, M. D., Rogers, T. G., et al. (2019). Analyzing machine learning workloads using a detailed gpu simulator. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 151–152. IEEE.
- [Mei and Chu 2016] Mei, X. and Chu, X. (2016). Dissecting gpu memory hierarchy through microbenchmarking. *IEEE Transactions on Parallel and Distributed Systems*, 28(1):72–86.
- [Raihan et al. 2019] Raihan, M. A., Goli, N., and Aamodt, T. M. (2019). Modeling deep learning accelerator enabled gpus. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 79–92. IEEE.
- [Serpa et al. 2019] Serpa, M. S., Moreira, F. B., Navaux, P. O., Cruz, E. H., Diener, M., Griebler, D., and Fernandes, L. G. (2019). Memory performance and bottlenecks in multicore and gpu architectures. In *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 233–236. IEEE.
- [Shekofteh et al. 2019] Shekofteh, S., Noori, H., Naghibzadeh, M., Yazdi, H. S., Fröning, H., et al. (2019). Metric selection for gpu kernel classification. *ACM Transactions on Architecture and Code Optimization (TACO)*, 15(4):68.
- [Smith 1986] Smith, B. J. (1986). A pipelined, shared resource mimd computer. In *Advanced computer architecture*, pages 39–41. IEEE Computer Society Press.
- [Volkov 2010] Volkov, V. (2010). Better performance at lower occupancy. In *Proceedings of the GPU technology conference, GTC*, volume 10, page 16. San Jose, CA.
- [Wong et al. 2010] Wong, H., Papadopoulou, M.-M., Sadooghi-Alvandi, M., and Moshovos, A. (2010). Demystifying gpu microarchitecture through microbenchmarking. In *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, pages 235–246. IEEE.

## A. Colab

Este minicurso apresenta também um roteiro de exemplos e exercícios que podem ser executados na plataforma **colab** da Google. Este roteiro foi desenvolvido com a colaboração dos estudantes da Universidade Federal de Viçosa: Marcelo Menezes, Michael Canesche e Westerley Carvalho.

O **colab** ou *Colaboratory* é um ambiente com um **Jupyter notebook** que não requer inicializações e executa na nuvem. Com o **colab**, você pode escrever e executar código, documentar, fazer análise, ter acesso a recursos computacionais diretamente de seu navegador sem custos adicionais. Na versão atual, o **colab** disponibiliza uma CPU *XEON dual-core* com 46 MB de cache L3, 360 GB de área em disco, GPU K80 ou T4 e acesso ao acelerador TPU da Google.

O **colab** foi desenvolvido para executar códigos em Python integrado com ambientes como TensorFlow, PyTorch, Keras, etc. Entretanto, como é um computador na nuvem com Linux tem muita flexibilidade e pode ser configurado para executar código C/C++ e CUDA. Nesta Seção iremos descrever brevemente os principais passos para executar códigos CUDA no **colab**. Além do exemplo do **colab** para este minicurso, estamos desenvolvendo outros exemplos para iniciantes ou com temas específicos como stencil, convolução, etc. Todo o material pode ser acessado a partir do link <sup>5</sup> do minicurso no Github, incluindo um exemplo bem simples para iniciantes de como usar o **colab** com GPU.

### A.1. Configurando o Colab

A abrir o link <https://colab.research.google.com> no seu navegador, você primeiro deve configurar a GPU. Nas opções ir para o menu *runtime* → *change runtime type* → *Hardware accelerator* escolher GPU. Você pode ter acesso a uma GPU K80 ou um GPU Turing T4. Para verificar executar o comando

```
!nvidia-smi
```

Note que é um comando de uma ferramenta da Nvidia com o caracter '!' na frente. Todos os programas que você instalar ou já estiverem instalados podem ser executados com um '!' na frente, como se fossem comandos de linha do Linux.

O **colab** tem células de texto e de código. As de texto servem para documentar, dar instruções, etc. A célula de código irá executar código python. Porém, o desenvolvedor Andrei Nechaev disponibilizou um plugin <sup>6</sup> para executar códigos CUDA. Ao criar uma nova célula basta iniciar com `%%cu` seguido do seu código CUDA. O código será compilado e executado. Adaptamos este plugin para iniciar a célula com `%%gpu`. Além disso, o plugin deve ser carregado no ambiente. A seguir os comandos para configurar:

```
!pip install git+git://github.com/canesche/nvcc4jupyter.git
%load_ext nvcc_plugin
```

---

<sup>5</sup>[github.com/cacauvicosa/wscad2019](https://github.com/cacauvicosa/wscad2019)

<sup>6</sup><https://github.com/andreinechaev/nvcc4jupyter>

feito isto, você pode abrir uma nova célula e editar, compilar e executar código em CUDA. A seguir um exemplo simples:

```
%%gpu
#include <stdio.h>
#include <stdlib.h>

__global__ void add(int *a, int *b, int *c) {
    *c = *a + *b;
}

int main() {
    int a, b, c; // host copies of variables a,b&c
    int *d_a, *d_b, *d_c; // device copies of variables a,b&c
    // Allocate space for device copies of a, b, c
    int size = sizeof(int);
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);
    c = 0; // Setup input values
    a = 4;
    b = 8;
    // Copy inputs to device
    cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

    add<<<1,1>>>(d_a, d_b, d_c); // Launch add() {\em kernel} on GPU

    // Copy result back to host
    cudaError err=cudaMemcpy(&c, d_c,size,cudaMemcpyDeviceToHost);
    if(err!=cudaSuccess) {
        printf("CUDA error copying to Host: %s\n",
            cudaGetErrorString(err));
    }
    printf("result is %d\n",c);
    // Cleanup
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);
    return 0;
}
```

## A.2. Executando com nvprof

Além da execução, introduzimos mais opções. Começando com a diretiva `%%nvprof`, o código CUDA da célula será executado com o **nvprof** e podemos verificar o número de

chamadas de cada *kernel* e de cada API, bem como o tempo de execução. Se você estiver em uma sessão com uma GPU K80, poderá também usar as métricas do **nvprof**.

### A.3. Usando Github, Compilando e Executando

Uma segunda opção é compilar e executar com comando de linha. Primeiro, você pode configurar o *path* com os comandos:

```
import os
os.environ['PATH'] += ':/usr/local/cuda/bin'
```

Depois pode copiar da sua conta *Google Drive* ou baixar do *github* ou outro sítio da internet. No caso dos exemplos deste minicurso, você pode baixar com o comando:

```
%cd /content
!git clone https://github.com/cacauvicosa/wscad2019.git
```

onde o comando *cd* é para se posicionar no pasta raíz, seguido do comando para *download* do repositório com os exemplos e documentação do minicurso. Note que para navegar nas pastas usamos com comando *cd* com o caracter *%* na frente, não usar com '!'. O próximo passo é ir para pasta desejada, compilar e executar um exemplo, como a seguir:

```
%cd /content/wscad2019
!nvcc -arch=sm_35 1_Introducao/poly.cu
!nvprof ./a.out
```

onde o código *poly.cu* foi compilado com *nvcc* e depois a execução com *profile* usando o **nvprof**.

Finalmente, podemos mesclar as duas abordagens, para mostrar a parte do código que o usuário pode fazer pequenas modificações e deixar o restante no arquivo, usando uma abordagem mista como ilustra o exemplo a seguir:

```
%%gpu
#define POWER 26

__global__ void poli(float* poli, const int N) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < N) {
        float x = poli[idx];

        poli[idx] = 5 + x * ( 7 - x * (9 + x *
            (5 + x * (5 + x)))) - 1.0f/x + 3.0f/(x*x) + x/5.0f;
    }
}

#include "/content/wscad2019/1_Introducao/poly_test.cu"
```

onde o usuário pode definir o tamanho do vetor de entrada e visualizar o código do *kernel* que avalia um polinômio. O usuário pode modificar a expressão e observar o tempo de execução. A maior parte do código ficou "escondida" no arquivo *poly\_test.cu*, deixando a interface mais leve e a atenção apenas na parte que está sendo avaliada.

## Capítulo

# 2

## Introdução à Programação com OpenACC

Evaldo B. Costa – DCC/UFRJ – ebcosta@dcc.ufrj.br

Gabriel P. Silva – DCC/UFRJ – gabriel@dcc.ufrj.br

### Resumo

O OpenACC (programação para aceleradores) é um modelo de programação para computação paralela desenvolvido com o objetivo de simplificar a programação paralela, oferecendo alto desempenho e portabilidade entre vários tipos de arquiteturas: multicore, manycore e GPUs. Este minicurso tem por objetivo apresentar este novo modelo de programação e suas facilidades de uso. A proposta deste minicurso é oferecer uma introdução à programação com OpenACC através de uma abordagem expositiva como: uma visão geral dos conceitos, diretivas e cláusulas; acrescidas da utilização de exemplos com os diversos tipos de arquiteturas alvo.

### 1.1. Introdução

As arquiteturas paralelas tem alcançado um alto grau de paralelismo com a utilização de um número cada vez maior de processadores, como podem ser encontrados nas arquiteturas multicore, manycore e GPUs (Figura 1.1). Esses tipos de arquitetura, com uso de processamento paralelo maciço, são usados extensivamente em aplicações nas áreas de geofísica, sequenciamento genético, simulações de modelos matemáticos e previsão do tempo, entre outras [Silva 2018] [Costa et al. 2018].

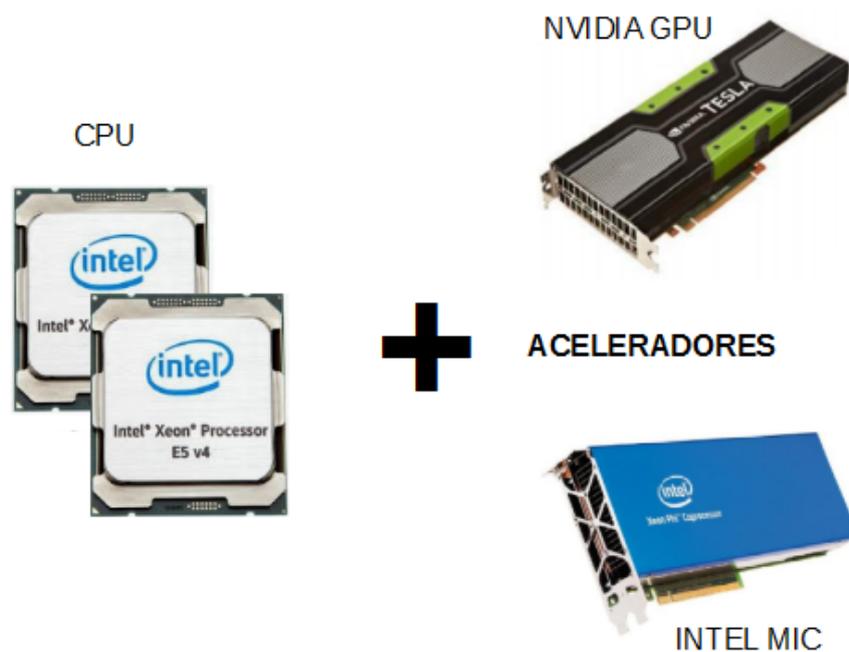
Entre os tipos de arquiteturas existentes destacam-se as GPUs e manycore. Desenvolvida pela NVIDIA nos anos 90 a GPU (Graphical Processing Unit) se baseia em um grande número de núcleos para processamento paralelo maciço com foco na eficiência energética e para aplicações com demandas que melhorem o throughput. Inicialmente desenvolvidos com o objetivo de atender a área de jogos, rapidamente mostrou-se muito eficiente em outras áreas [A. and M. 2012].

Em 2010 a Intel iniciou os primeiros estudos para o desenvolvimento da arquitetura MIC (Many Integrated Core). A arquitetura do Intel MIC (Many Integrated Core) oferece paralelismo maciço e vetorização com foco em computação de alto desempenho (HPC), que utiliza grandes demandas de dados para processamento. Atualmente essas arquiteturas

estão presentes em todos os setores, como a engenharia, medicina, economia e finanças [Rahman 2013].

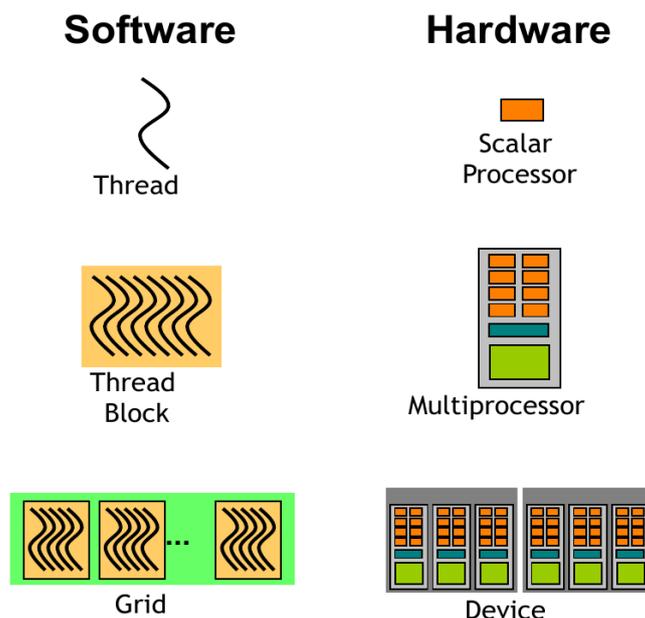
Na programação dessas arquiteturas são utilizados modelos de programação como CUDA, OpenCL e OpenACC. Neste minicurso usaremos o modelo de programação OpenACC, que surgiu em 2011 utilizando diretivas de compilação de alto nível e foi desenvolvido para o uso em aceleradores como os produzidos pela NVIDIA e Intel [Larkin 2018].

O OpenACC foi desenvolvido pelos principais fabricantes de hardware e software como a Cray, CAPS, NVIDIA e PGI, com o objetivo de simplificar a programação paralela, oferecendo alto desempenho e tornando possível a portabilidade do código independente de qual arquitetura foi desenvolvida inicialmente [OpenACC-Standard.org 2015] [Chen 2017].



**Figura 1.1:** Tipos de arquiteturas: multicore, GPU e manycore

As *threads* são executados por processadores escalares. Os *thread blocks* são executados em multiprocessadores e não podem migrar. Diversos *thread blocks* podem residir em um multiprocessador, limitados pelo total de recursos disponíveis no multiprocessador, tais como memória compartilhada e banco de registradores. Um *kernel* é lançado como uma grade de *thread blocks*. Veja na Figura 1.2.



**Figura 1.2:** Modelo de Execução [Abbott 2017]

Este minicurso introdutório à programação OpenACC consiste de três partes: na primeira parte serão abordados os conceitos iniciais, como definição do modelo, vantagens e desvantagens e onde melhor se aplica o seu uso; na parte seguinte estudaremos quais as principais diretivas usadas; finalmente na última parte veremos quais compiladores suportam o OpenACC e como compilar um programa, sendo que também examinaremos um exemplo com análise dos resultados.

## 1.2. Referencial Teórico

### 1.2.1. Avaliação de Desempenho

Antes de iniciarmos o estudo do OpenACC é importante destacar alguns conceitos sobre como avaliar o desempenho de um programa paralelo.

Uma das questões que surgem ao elaborar um programa paralelo é saber se o mesmo apresenta um desempenho adequado quando executado em um ambiente paralelo. Deste modo, para avaliar o desempenho de um sistema de processamento paralelo, as métricas mais importantes para serem consideradas são: *speedup* (também conhecido como ganho de desempenho ou aceleração), eficiência e escalabilidade.

São diversos fatores que influenciam essas métricas, tais como o custo de sincronização e comunicação, a distribuição das tarefas entre os processadores e o percentual do tempo de execução do programa que é passível de paralelização.

#### 1.2.1.1. Speedup

O *speedup*, ou aceleração, mede a razão entre o tempo gasto para execução de um algoritmo ou aplicação em um único processador ( $T(1)$ ) e o tempo gasto na execução com  $P$

processadores ( $T(P)$ ), como visto na Equação 1:

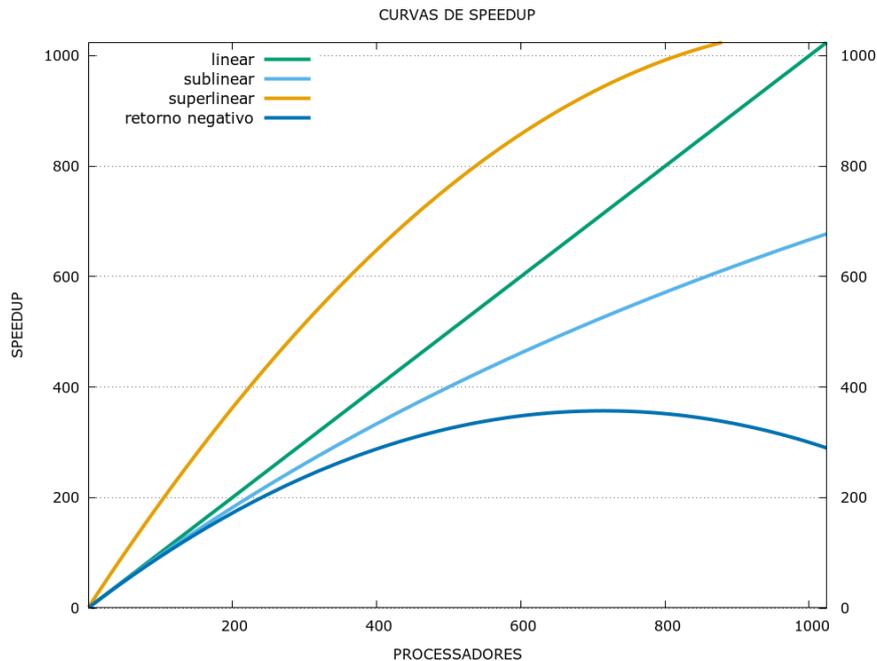
$$S(P) = \frac{T(1)}{T(P)} \quad (1)$$

Em condições ideais, quando o *speedup* é sempre igual a  $P$ , onde  $P$  é o número de processadores em uso, temos o chamado *speedup* linear. Mas, em geral, o *speedup* é menor do que  $P$ , devido principalmente à sobrecarga de comunicação entre os diferentes fluxos de execução do programa, perdas por sincronização e decomposição de tarefas mal feita. Quando isso acontece, chamamos o *speedup* de sublinear. Essa situação pode se deteriorar até o ponto em que a adição de mais processadores diminui, em vez de aumentar, o ganho obtido, caracterizando o que se chama de “retorno negativo”.

Em algumas situações especiais, *speedups* superiores a  $P$  podem ser obtidos (denominados “*speedups* superlineares”). Exemplos destas situações são aplicações onde o volume de dados manipulados é grande o suficiente para exceder o tamanho da memória cache de um único processador. Nesse caso, ao dividir a aplicação entre  $P$  processadores, o volume de dados manipulado por cada processador é aproximadamente dividido por  $P$ , sendo este volume agora pequeno o suficiente para poder ser armazenado integralmente, ou com baixo nível de interferência destrutiva, na memória cache de cada processador. Dentro deste quadro, e respeitadas as condições mencionadas anteriormente, é razoável se esperar um *speedup* superior a  $P$  no processo de paralelização, já que o desempenho com um único processador fica muito prejudicado pela baixa taxa de acerto nos acessos à memória cache.

Uma situação análoga ocorre quando são feitas buscas em grandes bases de dados, tais como a busca de dados genômicos realizadas por diversas implementações paralela do programa BLAST. Neste caso, a quantidade de memória RAM acumulada de cada um dos nós em um *cluster* permite que a base de dados se mova do disco inteiramente para a memória, reduzindo portanto dramaticamente o tempo requerido, por exemplo, para o mpiBLAST percorrer toda a base de dados [Correa and Silva 2012].

O *speedup* superlinear pode ocorrer também, sob determinadas condições, quando da execução de algoritmos de *backtracking* e *branch and bound* paralelos [Silva 2018]. A Figura 1.3 ilustra diferentes curvas de *speedups*.



**Figura 1.3:** Curvas de *speedup*

### 1.2.1.2. Eficiência

A **eficiência** é uma medida de quão efetiva é a adição de novos processadores para ajudar na resolução de um problema. Seu valor é obtido pela razão entre o *speedup* ( $S(P)$ ) e o número de processadores ( $P$ ) utilizados para obter este *speedup*, conforme podemos observar na equação 2:

$$E(P) = \frac{S(P)}{P} \quad (2)$$

Novamente, como o *speedup* em geral é menor do que  $P$  por conta da sobrecarga do processamento paralelo, a eficiência tipicamente assume um valor menor do que 1, mas, preferencialmente, próximo de 1. Para se obter *speedup* muito próximo de  $P$  e, consequentemente, eficiência muito próxima de 1, as seguintes condições devem normalmente ser satisfeitas:

- no código a ser paralelizado, o percentual de código não paralelizável (que continuará sendo executado de forma sequencial) é muito pequeno;
- no processo de paralelização, a distribuição de carga de trabalho entre os  $P$  processadores deve ser homogênea;
- os processadores trabalham nos trechos de código executados em paralelo de forma bastante independente, requerendo muito pouca comunicação ou sincronização entre eles.

### 1.2.1.3. Escalabilidade

Um sistema é dito escalável quando sua eficiência se mantém constante à medida que o número de processadores  $P$  aplicado à solução do problema cresce. Se o tamanho do problema é mantido constante e o número de processadores aumenta, o *overhead* de comunicação tende a crescer e a eficiência a diminuir. Na prática, uma análise da escalabilidade deve considerar a possibilidade de se aumentar proporcionalmente o tamanho do problema a ser resolvido à medida que  $P$  cresce de forma a contrabalançar o natural aumento do *overhead* de comunicação quando  $P$  cresce.

Considere como exemplo um problema de tamanho  $S$ . Usando  $P$  processadores esse problema leva um tempo  $T$  para ser executado. O sistema é dito escalável se um problema de tamanho  $2S$  executado em  $2P$  processadores leva o mesmo tempo  $T$ . Escalabilidade é frequentemente uma propriedade mais desejável que o *speedup*.

É importante ter essas métricas em mente, para sabermos se os programas paralelos que iremos desenvolver em OpenACC estão realmente tendo o desempenho esperado.

## 1.3. Programação em OpenACC

O OpenACC é um modelo de programação aberta para computação paralela desenvolvido com o objetivo de simplificar a programação paralela, oferecendo alto desempenho e portabilidade entre diversos tipos de arquiteturas: *multicore*, *manycore* e GPUs.

O OpenACC é compatível com os modelos de programação OpenMP e MPI, ambas as abordagens podem ser combinadas com o OpenACC. Em geral, as diretivas do OpenACC são muito semelhantes às do OpenMP. Em relação ao CUDA, OpenACC é totalmente compatível tornando a necessidade de alteração do código a menor possível.

O modelo de programação OpenACC possui algumas características que o tornam fácil e simples de utilizar, como:

- Independente de fabricante;
- Oculta a complexidade do *hardware* dos programadores;
- Requer poucas modificações ao código fonte;
- Mais fácil de programar e depurar que o CUDA;
- Possui algumas facilidades que o CUDA não oferece;
- Mesmo código por ser usado em *multicore*, *manycore* e GPUs;
- Similar ao OpenMP (familiaridade);
- Fácil transição para o OpenMP 4.5 (futuro).

Antes de iniciar o processo de programação utilizando o OpenACC é recomendado que se faça uma análise do código a ser paralelizado seguindo o ciclo de desenvolvimento e análise de código conforme descrito:

1. Analisar o código para determinar quais as regiões mais prováveis que podem ser paralelizadas ou otimizadas.
2. Paralelizar o código iniciando com as regiões com o maior tempo de execução.
3. Otimizar o código para melhorar o tempo de execução observado a partir das alterações executadas.

Na Figura 1.4 é apresentado o ciclo de desenvolvimento e análise do código.



**Figura 1.4:** Ciclo de desenvolvimento e análise do código

O OpenACC pode ser utilizado em códigos programados em linguagens de programação C/C++ ou Fortran, deve-se atentar a sintaxe correta para tipo de linguagem de programação correta conforme a Figura 1.5.



**Figura 1.5:** Sintaxe do OpenACC em C/C++ e Fortran

#### 1.4. Diretivas e Cláusulas

O modelo de programação usado no OpenACC é baseado em diretivas e cláusulas. As diretivas são comandos de instrução passadas pelo programador ao compilador. Cláusulas são os parâmetros adicionais atribuídos às instruções usadas nas diretivas.

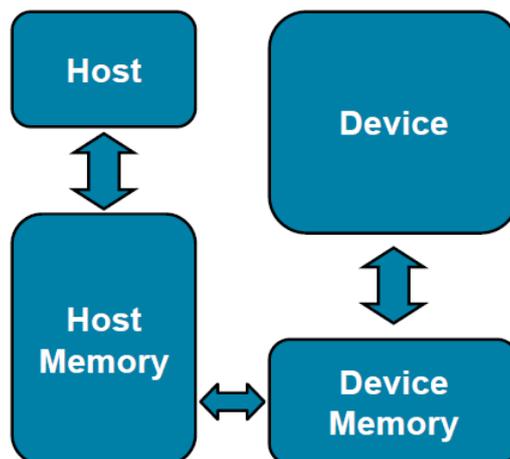
As diretivas do OpenACC são muito parecidas com as diretivas do OpenMP. As diretivas são escritas na forma de **pragma**. Existem vantagens em usar diretivas, uma delas é o fato de o código precisar de pequenas modificações, as mudanças podem ser feitas de

forma incremental, um **pragma** de cada vez. Com o uso desse processo torna-se especialmente útil para fins de depuração, já que fazer uma única alteração permite identificar rapidamente um erro.

O uso do OpenACC pode ser desativado em tempo de compilação. Quando o suporte OpenACC está desabilitado na compilação, o **pragma** referente ao OpenACC é considerado um comentário, sendo ignorado pelo compilador. Com isso, um único código pode ser usado para compilar tanto um código com suporte a acelerador quanto uma versão sequencial.

#### 1.4.1. Movimentação de dados

Um grande fator de impacto de desempenho no processamento paralelo é a movimentação de dados, principalmente quando se faz o processamento dos dados em lugares diferentes. Quando se usa processamento em aceleradores nem sempre é possível carregar todos os dados para o acelerador, isso ocorre em geral porque a memória da CPU é maior a dos aceleradores, embora os aceleradores tenham maior largura de banda (Figura 1.6).



**Figura 1.6:** Modelo básico de movimentação de dados entre host e o acelerador [Chen 2017]

A movimentação de dados entre o *host* e o acelerador é feita através do barramento, que é lento em comparação com largura de banda de memória. Por sua vez o acelerador não pode executar o processamento dos dados até que eles estejam na sua memória local.

Para realizar a movimentação de dados entre o *host* e o acelerador durante a execução do programa é necessário o uso das cláusulas de dados. As cláusulas de movimentação de dados podem ser usadas nas diretivas **kernels** ou **parallel** e tem como principais características:

- Define a região do código na qual os dados permanecem no acelerador;
- Define quais dados são compartilhados entre todos os *kernels* de uma região paralela;
- Realiza transferências de dados explícitas.

```
#pragma acc data [clause]
```

Cláusula	Descrição
<b>copy</b>	Cria espaço para as variáveis listadas no dispositivo, inicia as variáveis copiando dados para o dispositivo no início da região, copia os resultados de volta para o <i>host</i> no final da região e finalmente libera o espaço no dispositivo quando terminar.
<b>copyin</b>	Cria espaço para as variáveis listadas no dispositivo, inicia a variável copiando os dados para o dispositivo no início da região e libera o espaço no dispositivo quando terminar, sem copiar os dados de volta para o <i>host</i> .
<b>copyout</b>	Cria espaço para as variáveis listadas no dispositivo, mas não as inicia. No final da região, copia os resultados de volta para o <i>host</i> e libera o espaço no dispositivo.
<b>create</b>	cria espaço para as variáveis listadas e as libera no final da região, mas não copia nenhum dos dados de/para o dispositivo.
<b>present</b>	As variáveis listadas já estão presentes no dispositivo, portanto, nenhuma outra ação precisa ser executada. Isso é usado com mais frequência quando existe uma região de dados em uma rotina de maior nível.
<b>deviceptr</b>	As variáveis listadas usam a memória do dispositivo que foi gerenciada fora do OpenACC, portanto as variáveis devem ser usadas no dispositivo sem qualquer conversão de endereço. Esta cláusula é geralmente usada quando o OpenACC é misturado com outro modelo de programação.

**Tabela 1.1:** Cláusulas da Diretiva Data

## 1.5. Diretiva Parallel

A paralelização usando o construtor **parallel** identifica uma região de código que será paralelizada, quando executada em conjunto com a diretiva **loop**, o compilador gerará uma versão paralela do laço para o acelerador. Desta forma o compilador executa o paralelismo do código de forma direta.

```
#pragma acc parallell [clause-list]
```

Neste modo de programação, os laços que serão paralelizados precisam ser definidos no código, o compilador não consegue identificar os laços que precisam ser paralelizados, se a diretiva **loop** não for especificada não será feita a sua paralelização.

### 1.5.1. Cláusulas da Diretiva Parallel

Algumas cláusulas são usadas para melhorar o desempenho da região a ser paralelizada, essas cláusulas permitem ter um maior controle e processamento dos laços.

Cláusula	Descrição
<b>private</b>	A cláusula privada especifica que cada iteração do laço tem a sua própria cópia das variáveis listadas
<b>reduction</b>	A redução é executada com as operações suportadas: + * max min
<b>async</b>	Retira as barreiras implícitas no final da região paralela

**Tabela 1.2:** Cláusulas da Diretiva Parallel

A cláusula **private** da construção **parallel** vai privatizar as variáveis listadas para cada *gang* na região paralela.

A cláusula **reduction** funciona de forma similar à cláusula **private** de forma que é gerada uma cópia privada das variáveis, mas existe a redução ao final da região paralela de todas as cópias privadas em um único resultado final, que é retornado ao sair da região paralela. As operações de redução possíveis são soma, multiplicação, máximo e mínimo.

A construção **parallel loop** possui uma cláusula com o mesmo nome **reduction** e com funcionamento similar. O formato para a cláusula de redução é como a seguir:

```
#pragma acc parallel reduction(operator:variable)
```

### 1.6. Diretiva Parallel Loop

Para fazer o paralelismo de vários laços, é necessário que cada laço seja acompanhado de uma diretiva **parallel loop**.

A diretiva **parallel loop** é uma afirmação do programador de que é seguro e desejável paralelizar o laço afetado. Isso depende se o programador identificou corretamente o paralelismo no código e removeu qualquer coisa no código que poderia tornar perigosa a sua paralelização. Se o programador afirmar incorretamente que o laço pode ser paralelo, e ele não pode, a aplicação resultante pode produzir resultados incorretos.

O programador identifica o paralelismo sem dizer ao compilador como explorar esse paralelismo. Isso significa que o código OpenACC pode ser portado para outros dispositivos além do dispositivo para o qual o código foi primeiramente desenvolvido, porque detalhes sobre como paralelizar o código são deixados para o compilador decidir, ao invés de ser especificado explicitamente no código fonte.

Cada região **parallel loop** pode ter diferentes laços e cada laço pode ser paralelizado e otimizado de forma independente entre eles. Porém alguns cuidados devem ser tomados.

```

#pragma acc parallel loop
for (int i = 0 ; i < N; i++)
    a[i] = 0;
#pragma acc parallel loop
for (int j = 0 ; j < M; j++)
    b[j] = 0;

```

### Exemplo 1.1: Diretiva Parallel Loop

Por exemplo, essa é a maneira recomendada de paralelizar vários laços. Tentando paralelizar múltiplos laços dentro da mesma região paralela pode ocasionar problemas de desempenho ou resultados inesperados.

#### 1.6.1. Cláusulas da Diretiva Parallel Loop

Como o OpenACC serve como uma linguagem para aceleradores genéricos existem três níveis de paralelismo que podem ser usados no OpenACC. Eles especificam o nível de paralelismo contidos na rotina, são chamados de **gang**, **worker** e **vector**. Uma *gang* é composta por um ou vários *workers*. Todos os *workers* de uma *gang* podem compartilhar os mesmos recursos, como memória cache ou processador.

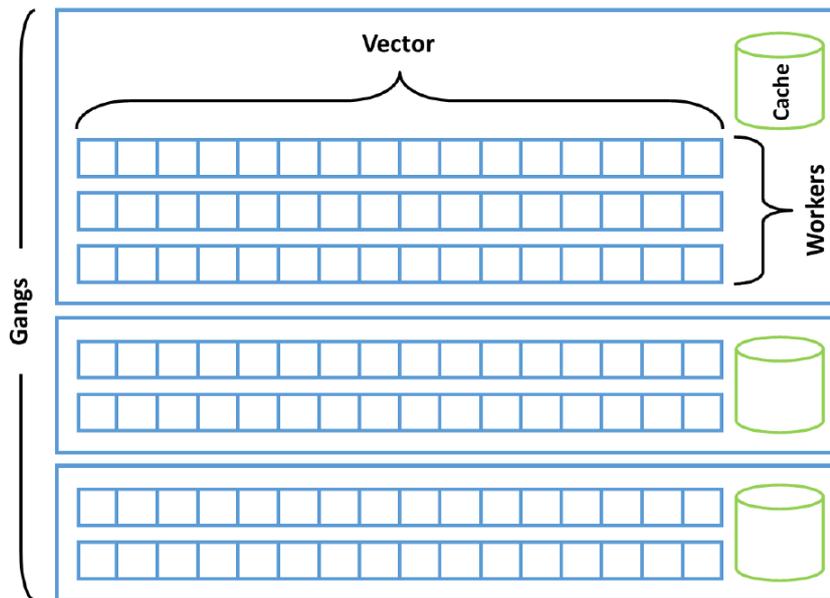
Cláusula	Descrição
<b>gang</b>	Particiona o laço entre as <i>gangs</i>
<b>worker</b>	Particiona o laço entre os <i>workers</i>
<b>vector</b>	Vetoriza o laço
<b>seq</b>	Não particiona o laço, que é executado sequencialmente

**Tabela 1.3:** Cláusulas da Diretiva Parallel Loop

Os níveis de paralelismo usado no OpenACC podem ser comparados ao níveis de execução usados na programação CUDA. Podendo assim admitir a relação entre eles: **gang** = **block**, **worker** = **warp** e **vector** = **threads**.

Essas diretivas também podem ser combinadas em um laço específico. Por exemplo, um laço **gang vector** pode ser particionado entre *gangs*, cada uma delas com 1 *worker* implicitamente, e depois vetorizado.

A especificação OpenACC reforça que o laço mais externo deve ser um laço de uma *gang*, o laço paralelo mais interno deve ser um laço *vector* e um laço *worker* pode aparecer no meio. Um laço sequencial (**seq**) pode aparecer em qualquer nível.



**Figura 1.7:** Gangs x Workers x Vector [OpenACC-Standard.org 2015]

O uso dos níveis de paralelismo são aplicados na diretiva **parallel loop** para gerar maior ganho na execução do laço. Também podem ser usadas da diretiva **kernels**.

```
#pragma acc parallel loop gang
for(i = 0 ; i < size; i++)
  #pragma acc loop worker
  for(j = 0 ; j < size; j++)
    #pragma acc loop vector
    for(k = 0 ; k < size; k++)
      c[i][j]+=a[i][k]*b[k][j];
```

**Exemplo 1.2:** Cláusulas da Diretiva Parallel Loop

## 1.7. Diretiva kernels

Quando é utilizada a diretiva **kernels**, isto significa que é informado ao compilador que existem regiões do código que podem ser paralelizadas e que o compilador será o responsável por identificar quais são as regiões e qual a estratégia que será utilizada. A estratégia definida pelo compilador pode ser extrair o máximo de paralelismo do código ou executar somente o mínimo de paralelismo.

Com uso da diretiva **kernels**, o compilador analisará o código e apenas paralelizará quando tiver certeza de que é seguro fazê-lo. Em alguns casos, o compilador pode não ter informações suficientes para determinar se é seguro paralelizar um laço, neste caso essa paralelização não será feita. Os passos do processo de compilação do código usando a

diretiva **kernels** são:

- Analisar o código para identificar regiões de paralelismo
- Se encontrado, identificar quais dados devem ser transferidos
- Criar um *kernel*
- Movimentar os dados para dispositivo

```
#pragma acc kernels [clause-list]
```

### 1.8. Diretiva **kernels** vs Diretiva **parallel**

As diretivas **kernels** e **parallel** no OpenACC, tem características distintas para cada tipo de aplicação e são usadas de acordo com a necessidade de execução do código.

Existem códigos que são fáceis de alterar e obtém melhor desempenho usando usando a diretiva **parallel**, porém existem códigos que possuem grande dificuldade de alteração não sendo possível usar diretiva **parallel**, neste caso é usado a diretiva **kernels**, pois as alterações são as mínimas possíveis.

No exemplo abaixo usamos um trecho de um código que possui dois laços comparando o processo de paralelização utilizando as diretivas de **kernels** e **parallel**.

<b>kernels</b>	<b>parallel</b>
<pre>#pragma acc kernels {   for (i=0; i&lt;n; i++)     a[i] = 3.0f*(float)(i+1);   for (i=0; i&lt;n; i++)     b[i] = 2.0f*a[i]; }</pre>	<pre>#pragma acc parallel {   #pragma acc loop   for (i=0; i&lt;n; i++)     a[i] = 3.0f*(float)(i+1);   #pragma acc loop   for (i=0; i&lt;n; i++)     b[i] = 2.0f*a[i]; }</pre>

Nota-se que no processo de execução usando a diretiva **kernels** são gerados dois *kernels*, um para cada laço. Existe um barreira entre os laços, deste modo o segundo laço só será iniciando quando o primeiro laço terminar.

No processo de execução usando a diretiva **parallel** é gerado um único *kernel*. Não existe barreira entre os laços, assim podem ser executados de forma independente.

## 1.9. Operações Atômicas

Quando uma ou mais iterações de um laço precisam acessar um elemento na memória ao mesmo tempo, condições de corrida podem ocorrer. Por exemplo, se uma iteração do laço está modificando o valor contido em uma variável e outra está tentando ler a mesma variável em paralelo, diferentes resultados podem ocorrer dependendo de qual iteração ocorra primeiro.

Em programas seriais, os laços sequenciais garantem que a variável será modificada e lida em uma ordem previsível, mas os programas paralelos não garantem que uma iteração específica de um laço irá ocorrer antes da outra. Em casos simples, como encontrar uma soma, valor máximo ou mínimo, uma operação de redução irá garantir que o programa será executado corretamente.

Para operações mais complexas, a diretiva **atomic** garantirá que não haverá duas *threads* executando a operação nela contida simultaneamente. O uso da diretiva **atomic** é às vezes uma parte necessária do processo de paralelização para garantir a correção do código.

A diretiva **atomic** aceita uma das quatro cláusulas seguintes para declarar o tipo de operação contida na região:

- A operação **read** assegura que duas iterações de um laço não farão leituras da região ao mesmo tempo;
- A operação **write** garantirá que não haja duas iterações realizando escrita na região ao mesmo tempo;
- Uma operação **update** é uma operação de leitura e de escrita combinadas;
- Finalmente, uma operação **capture** executa uma atualização, mas salva o valor calculado nessa região para ser utilizada no código seguinte à região.

Se nenhuma cláusula for definida, uma operação **update** é assumida.

Um histograma é uma técnica comum para contar quantas vezes os valores ocorrem em um conjunto de entrada de acordo com o seu valor. O código do exemplo abaixo percorre uma série de números inteiros de um intervalo conhecido e conta o total de ocorrências de cada número nesse intervalo. Como cada número no intervalo pode ocorrer várias vezes, precisamos garantir que cada elemento no vetor de histograma seja atualizado atomicamente. O código abaixo demonstra usando a diretiva **atomic** para gerar um histograma.

```
#pragma acc parallel loop
    for(int i=0;i<HN;i++)
        h[i]=0;
#pragma acc parallel loop
    for(int i=0;i<N;i++) {
#pragma acc atomic update
        h[a[i]]+=1; }
```

**Exemplo 1.3:** Diretiva Atomic

Observe que as atualizações no vetor do histograma  $h$  são executadas atomicamente. Como estamos incrementando o valor do elemento de um vetor, uma operação **update** é usada para ler o valor, modificá-lo e gravá-lo novamente.

## 1.10. Compilação

Antes de compilar qualquer código é importante saber quais dispositivos aceleradores estão configurados para uso no sistema. Existem alguns comandos que fornecem informações de modelos e características desses dispositivos.

Para aceleradores da NVIDIA existem os comandos *nvidia-smi* e *nvidia-settings*, estes comandos fornecem de informações de configuração como: modelo, cpus, cuda core, memória. Com o compilador da PGI também existe um comando chamado *pgaccelinfo* que fornece as principais características dos aceleradores instalados no sistema.

```
# nvidia-smi -q | grep "Product Name"
    Product Name                : Quadro K420
    Product Name                : Tesla K80
    Product Name                : Tesla K80

# pgaccelinfo | grep "Device Name"
    Device Name:                Tesla K80
    Device Name:                Tesla K80
    Device Name:                Quadro K420
```

Para compilar os códigos feitos em OpenACC, gerando código para execução em GPUs, é necessário o uso de compiladores que suportem OpenACC. A PGI (Portland Group), tem um versão disponibilizada para uso público sem a necessidade de licença, também existe uma versão com uso de licença que permite ter acesso à equipe de suporte da PGI.

A Cray também tem seu compilador o CCE 8.6.5, para usá-lo é necessário a aquisição de uma licença, não existe um licença para uso público.

Nos exemplos apresentados neste minicurso usaremos o compilador da PGI disponibilizado para a comunidade, o compilador pode ser baixado do sítio da PGI através no endereço <https://www.pgroup.com/products/community.htm>.

Para compilar códigos em C usaremos o comando *pgcc*, e para compilação de códigos em C++ usar o comando *pgc++*. Algumas parâmetros básicos devem ser usados durante a execução dos comandos *pgcc* e *pgc++*. Esses parâmetros definem em qual dispositivo o código é executado de acordo com a arquitetura. Relação dos principais parâmetros usados no comando *pgcc*:

Parâmetro	Descrição
-fast	faz a otimização do código
-acc	habilita o uso de diretivas OpenACC
-Minfo=accel	informações sobre quais partes do código foram aceleradas
-Minfo=opt	informações sobre todas as otimizações de código
-Minfo=all	informações sobre todas as saídas de código
-ta=host	compila o código em modo serial
-ta=multicore	compila o código usando <i>threads</i> em CPU
-ta=nvidia	compila o código usando NVIDIA

**Tabela 1.4:** Parâmetros de Compilação pgcc

Alguns exemplos de uso do compilador PGI e seus parâmetros básicos.

```
$ pgcc -acc -ta=nvidia -Minfo=accel main.c
$ pgc++ -acc -ta=nvidia -Minfo=accel main.cpp
```

Uma outra alternativa é o uso do compilador GNU GCC a partir da versão 6 o compilador tem suporte ao OpenACC. Como o GNU GCC é um compilador *opensource* não requer licença para uso. Para usar o GNU GCC especificar o parâmetro *-fopenacc*.

```
$ gcc -fopenacc main.c
```

## 1.11. Exemplos

Após a introdução do conceito e das principais diretivas usadas no OpenACC, veremos alguns exemplos da aplicações dessas diretivas, como elas se comportam e quais as melhores opções de uso das diretivas para gerar maior ganho.

Os códigos foram executados em um servidor com dois processadores Intel Xeon E5-2609 (2,40 GHz, 4 núcleos cada, cache de 10 MB), com 128 GB de memória compartilhada, discos locais de alta velocidade SSD (Solid-State Drive) e um acelerador NVIDIA Tesla K80 (24 GB de memória, 4992 CUDA cores). O sistema operacional usado foi a versão 7.3 da distribuição Centos Linux de 64 bits. Todos os códigos foram compilados usando o PGI Community Edition Version 19.4.

### 1.11.1. Cálculo de Pi

Iniciaremos com o exemplo básico do cálculo do número Pi. O valor de Pi é definido pela relação entre o perímetro de uma circunferência e seu diâmetro, para a maioria dos cálculos simples é comum usar a aproximação do valor de Pi para 3,1415. Em computação existem algoritmos que podem ser utilizados para o cálculo aproximado do Pi, como: Gauss-Legendre e Monte Carlo. A seguir é apresentada uma implementação simples para

o cálculo sequencial (serial) de Pi.

```
#include <stdio.h>
#define N 1000000000

int main(void) {
    double pi = 0.0f; long i;
    for (i=0; i<N; i++) {
        double t=(double) ((i+0.5)/N);
        pi += 4.0/(1.0+t*t);
    }
    printf("pi=%f\n", pi/N);
    return 0;
}
```

#### Exemplo 1.4: Cálculo de Pi Sequencial

A execução do cálculo de Pi dentro do laço na versão sequencial será feito por uma *thread*, independente de quantidade de processadores que existam no sistema.

Podemos paralelizar o código em OpenMP adicionando a linha **#pragma omp parallel for reduction(+: pi)** antes do laço, desta forma pode-se utilizar mais de uma *thread* para o cálculo.

```
#pragma omp parallel for reduction(+: pi)
for (i=0; i<N; i++)
{
    double t=(double) ((i+0.5)/N);
    pi += 4.0/(1.0+t*t);
}
```

#### Exemplo 1.5: Cálculo de Pi com OpenMP

Para que o cálculo seja executado pelo acelerador usaremos a diretiva **parallel** do OpenACC. Da mesma forma que foi usado no OpenMP, adicionaremos a linha **#pragma acc parallel loop reduction(+: pi)** antes do laço.

```
#pragma acc parallel loop reduction(+: pi)
for (i=0; i<N; i++)
{
    double t=(double) ((i+0.5)/N);
    pi += 4.0/(1.0+t*t);
}
```

#### Exemplo 1.6: Cálculo de Pi com OpenACC

Para avaliar o tempos de execução do cálculo de Pi, foram executados os códigos sequencial, com OpenMP e com OpenACC. Para o cálculo em OpenMP foram utilizadas 16 *threads*, a quantidade máxima de processadores no servidor.

O tempo de execução sequencial do cálculo de Pi foi de 5,6 segundos, o tempo com OpenMP foi de 0,60 segundos e com OpenACC foi de 0,15 segundos.

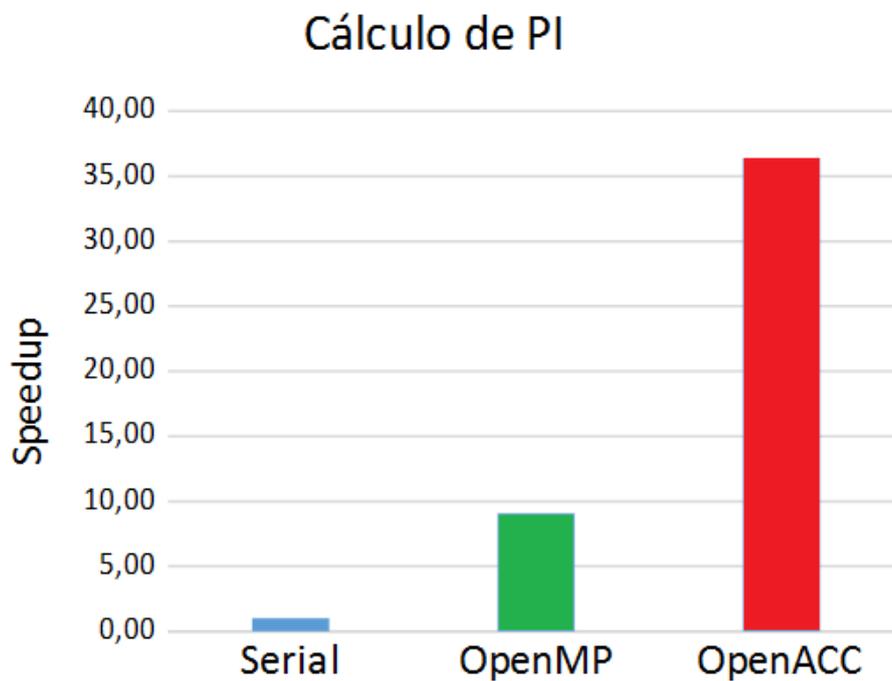
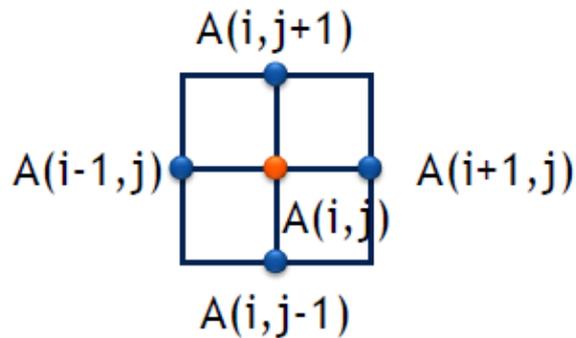


Figura 1.8: Seepdup - Cálculo do valor de Pi

O *speedup* encontrado usando programação OpenMP foi de 9,10 em comparação a execução em sequencial, enquanto que usando OpenACC o *speedup* foi de 36,40 em relação ao executado em sequencial. Os resultados são apresentados na Figura 1.8

### 1.11.2. Método Jacobi

O Método de Jacobi é um procedimento iterativo para a resolução de sistemas lineares. Converte iterativamente para o valor correto, calculando novos valores em cada ponto a partir da média dos pontos vizinhos. Neste exemplo faremos o cálculo da temperatura na placa usando a equação de Laplace:  $\nabla^2 f(x,y) = 0$ .



$$A_{k+1}(i,j) = \frac{A_k(i-1,j) + A_k(i+1,j) + A_k(i,j-1) + A_k(i,j+1)}{4}$$

A seguir é apresentada uma implementação sequencial (serial) para o cálculo da temperatura da placa usando o método Jacobi.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#define COLUMNS    1000
#define ROWS       1000

#define MAX_TEMP_ERROR 0.01

double Anew[ROWS+2][COLUMNS+2];
double A[ROWS+2][COLUMNS+2];

void initialize();
void track_progress(int iter);

int main(int argc, char *argv[]) {

    int i, j;
    int max_iterations=1000;
    int iteration=1;
    double dt=100;

    initialize();

    while ( dt > MAX_TEMP_ERROR && iteration
           <= max_iterations ) {

        for(i = 1; i <= ROWS; i++) {
```

```

        for(j = 1; j <= COLUMNS; j++) {
            Anew[i][j] = 0.25 * (A[i+1][j] +
                A[i-1][j] + A[i][j+1] + A[i][j-1]);
        }
    }

    dt = 0.0;

    for(i = 1; i <= ROWS; i++){
        for(j = 1; j <= COLUMNS; j++){
            dt = fmax( fabs(Anew[i][j]-A[i][j]), dt);
            A[i][j] = Anew[i][j];
        }
    }

    iteration++;
}

printf("\n Erro maximo na iteracao %d era %f\n",
        iteration-1, dt);
}

void initialize(){

    int i,j;

    for(i = 0; i <= ROWS+1; i++){
        for (j = 0; j <= COLUMNS+1; j++){
            A[i][j] = 0.0;
        }
    }

    for(i = 0; i <= ROWS+1; i++) {
        A[i][0] = 0.0;
        A[i][COLUMNS+1] = (100.0/ROWS)*i;
    }

    for(j = 0; j <= COLUMNS+1; j++) {
        A[0][j] = 0.0;
        A[ROWS+1][j] = (100.0/COLUMNS)*j;
    }
}
}

```

### Exemplo 1.7: Método de Jacobi Sequencial

O primeiro laço dentro do *while* de convergência calcula o novo valor para cada elemento

com base nos valores atuais de seus vizinhos. Armazenando em uma matriz temporária, garantindo que todos os valores sejam calculados usando o estado atual de **A** antes que **A** seja atualizado. Como resultado, cada iteração do laço é completamente independente uma da outra.

Esse laço também calcula um máximo valor de erro. O valor do erro é a diferença entre o novo valor e o antigo. Se a quantidade máxima de alteração entre duas iterações estiver dentro de alguma tolerância, o problema será considerado convergido e o laço externo será encerrado. O segundo laço simplesmente atualiza o valor de **A** com os valores calculados em **Anew**.

A execução do cálculo da temperatura dentro dos laços serão feitos por uma *thread*, independente de quantidade de *threads* que existam no sistema.

Podemos paralelizar o código em OpenMP adicionando as linhas **#pragma omp parallel for** antes do primeiro laço e a linha **#pragma omp parallel for reduction(max:dt)** antes do segundo laço, desta forma pode-se utilizar mais de uma *thread* para o cálculo.

```
#pragma omp parallel for
for(i = 1; i <= ROWS; i++) {
    for(j = 1; j <= COLUMNS; j++) {
        Anew[i][j] = 0.25 * (A[i+1][j] +
            A[i-1][j] + A[i][j+1] + A[i][j-1]);
    }
}

dt = 0.0;

#pragma omp parallel for reduction(max:dt)
for(i = 1; i <= ROWS; i++){
    for(j = 1; j <= COLUMNS; j++){
        dt = fmax( fabs(Anew[i][j]-A[i][j]), dt);
        A[i][j] = Anew[i][j];
    }
}
```

### Exemplo 1.8: Método de Jacobi Sequencial com OpenMP

Para que o cálculo seja executado pelo acelerador usaremos a diretiva **parallel** do OpenACC. Da mesma forma que foi usado no OpenMP, adicionar a linha **#pragma acc parallel loop**, e a linha **#pragma acc parallel loop reduction(max:dt)** no segundo laço.

```
#pragma acc parallel loop
for(i = 1; i <= ROWS; i++) {
    for(j = 1; j <= COLUMNS; j++) {
```

```

        Anew[i][j] = 0.25 * (A[i+1][j] +
        A[i-1][j] + A[i][j+1] + A[i][j-1]);
    }
}

dt = 0.0;

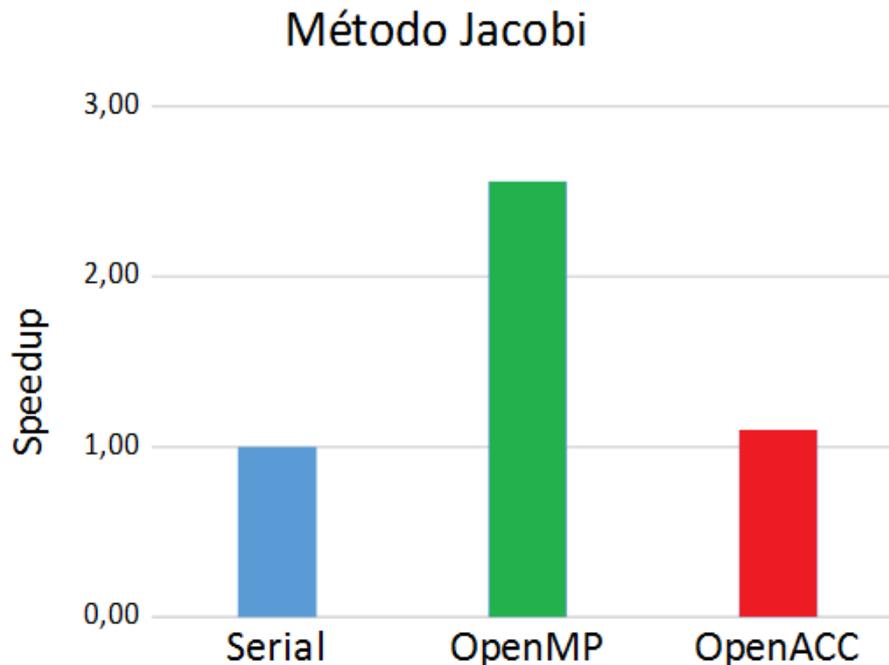
#pragma acc parallel loop reduction(max:dt)
for(i = 1; i <= ROWS; i++){
    for(j = 1; j <= COLUMNS; j++){
        dt = fmax( fabs(Anew[i][j]-A[i][j]), dt);
        A[i][j] = Anew[i][j];
    }
}

```

**Exemplo 1.9:** Método de Jacobi Sequencial com OpenACC - Versão 1

Para avaliar os tempos de execução dos cálculos da temperatura, foram executadas as versões sequencial, com OpenMP e com OpenACC. Para o cálculo em OpenMP foram utilizadas 16 *threads*, quantidade máxima de processadores no servidor.

Foram observados os seguintes tempos de execução: o código sequencial teve um tempo total de 11 segundos, sendo que com OpenMP o tempo de execução foi de 4,30 segundos e com OpenACC o valor medido foi de 10 segundos.



**Figura 1.9:** Seepdup - Método Jacobi

O *speedup* encontrado na execução do código com OpenMP foi de 2,56 em comparação

à execução em sequencial, enquanto que usando OpenACC o *speedup* foi de 1,10. Os resultados são apresentados na Figura 1.9.

Como visto, o resultado do *speedup* encontrado usando OpenACC foi quase idêntico aos resultados encontrados na execução do código sequencial. Isso ocorre porque a matriz de cálculo não está armazenada no acelerador. Toda vez que o acelerador executa uma operação as informações são gravadas na matriz que está na memória do *host*.

Para resolver este problema é necessário fazer a cópia da matriz para o acelerador de modo que não seja mais necessário gravar as informações no *host* toda vez que for realizada uma operação pelo acelerador.

Usaremos a diretiva **data** do OpenACC. Adicionar a linha **#pragma acc data copy(A) create (Anew)** antes dos dois laços para fazer a cópia da matriz para a memória do acelerador.

```
#pragma acc data copy(A) create(Anew)
while ( dt > MAX_TEMP_ERROR && iteration <=
max_iterations ) {

    #pragma acc parallel loop
    for(i = 1; i <= ROWS; i++) {
        for(j = 1; j <= COLUMNS; j++) {
            Anew[i][j] = 0.25 * (A[i+1][j] +
            A[i-1][j] + A[i][j+1] + A[i][j-1]);
        }
    }

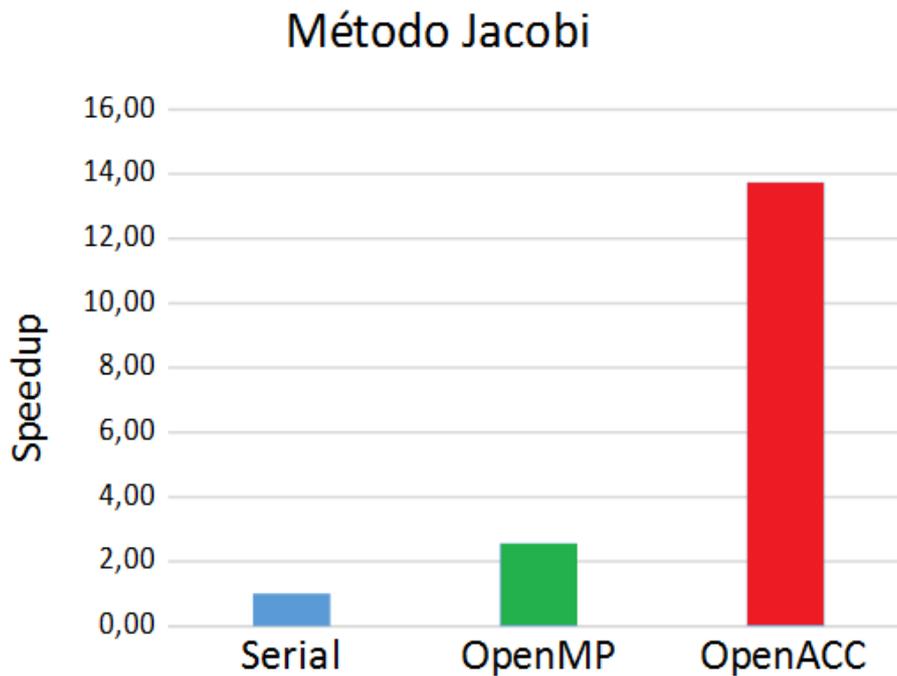
    dt = 0.0;

    #pragma acc parallel loop reduction(max:dt)
    for(i = 1; i <= ROWS; i++){
        for(j = 1; j <= COLUMNS; j++){
            dt = fmax( fabs(Anew[i][j]-A[i][j]), dt);
            A[i][j] = Anew[i][j];
        }
    }
}
```

#### Exemplo 1.10: Método de Jacobi Sequencial com OpenACC - Versão 2

O total de tempo para a execução sequencial foi de 11 segundos, sendo que o tempo total com OpenMP se manteve em 4,30 segundos e o tempo de execução com OpenACC foi reduzido para 0,80 segundos.

O *speedup* para a execução com OpenMP continuou em 2,56 em comparação a execução sequencial, enquanto que usando OpenACC o *speedup* aumentou para 13,75. Os resultados são apresentados na Figura 1.10.



**Figura 1.10:** Seepdup - Método Jacobi

## 1.12. Conclusão

O modelo de programação OpenACC foi desenvolvido pelos principais fabricantes de hardware e software do mercado, com o objetivo de simplificar a programação paralela, tornando possível a portabilidade do código.

Com o uso do OpenACC é possível atingir alto níveis de paralelismo usando arquitetura baseada em aceleradores. Entre suas principais características se destacam:

- O OpenACC é fácil de usar;
- Usa uma abordagem baseada em diretivas de compilação;
- Em alguns casos são feitas pequenas alterações no código;
- O código pode ser implementado em qualquer acelerador.

## Referências

- [A. and M. 2012] A., Z. L. F. and M., M. (2012). *Arquitetura e programação de GPU nvidia*. UNICAMP.
- [Abbott 2017] Abbott, S. (2017). *Advanced OpenACC*. NVIDIA Corporation.
- [Chen 2017] Chen, S. (2017). *Introduction to OpenACC*. Research Computing Services Information Services and Technology Boston University.
- [Correa and Silva 2012] Correa, J. C. and Silva, G. P. (2012). Analysis and performance evaluation of parallel BLAST. *I. J. Comput. Appl.*, 20(2):112–122.

- [Costa et al. 2018] Costa, E. B., Silva, G. P., and Teixeira, M. (2018). Avaliação de desempenho do montador daligner em arquiteturas manycore. In *WSCAD 2018 - WCH* (), São Paulo - SP, Brazil.
- [Larkin 2018] Larkin, J. (2018). *Introduction to OpenACC*. NVIDIA.
- [OpenACC-Standard.org 2015] OpenACC-Standard.org (2015). *OpenACC Programming and Best Practices Guide*. OpenACC-Standard.org.
- [Rahman 2013] Rahman, R. (2013). *Intel Xeon Phi Coprocessor Architecture and Tools The Guide for Application Developers*. Apress Open.
- [Silva 2018] Silva, G. P. (2018). *Programação Paralela com MPI Um Curso Introductório*. Amazon.