# Impacto do Subsistema de Memória em Arquiteturas CPU e GPU

**Matheus S. Serpa, Eduardo H. M. Cruz, Francis B. Moreira, Matthias Diener, Philippe O. A. Navaux**

[1] Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970, Porto Alegre – RS – Brasil

{msserpa, ehmcruz, fbmoreira, mdiener, navaux}@inf.ufrgs.br

***Abstract.*** *The variety of architectures and types of parallel applications impose a big challenge on developers. The same application can have a good performance when running on one architecture, but a bad performance on another architecture. In most situations, adapting the source code to the architecture is enough to fix such problems, although it is time consuming. Therefore, it is important to have a deep understanding of how the target application behaves on different architectures in order to optimize them to run on a variety of architectures with a good performance. The related work in this area mostly focuses on a limited analysis encompassing execution time and energy. Others focus on the behavior of the memory subsystem in a specific architecture, then propose a new mechanism or architecture. In this paper, we perform a detailed investigation of the impact of the memory subsystem of different architectures, which is one of the most important aspects to be considered, on the same set of parallel applications. For this study, we performed experiments in the Ivy Bridge (CPU) and Kepler architectures (GPU) using applications from the SHOC benchmark suite. In this way, we were able to understand why an application performs well on one architecture and badly on others.*

***Resumo.*** *A variedade de arquiteturas e tipos de aplicações paralelas impõe um grande desafio à desenvolvedores. A mesma aplicação pode ter um bom desempenho quando executada em uma arquitetura, porém um mal desempenho em outra arquitetura. Na maioria das situações, a adaptação do código fonte para a arquitetura é suficiente para solucionar tais problemas, embora demande tempo. Portanto, é importante um estudo detalhado sobre como uma aplicação se comporta em diferentes arquiteturas a fim de otimizá-la para executar em uma variedade de arquiteturas com um bom desempenho. Os trabalhos relacionados nesta área, em sua maioria, focam em uma análise abrangendo o tempo de execução e energia. Outros trabalhos focam no comportamento do subsistema de memória em uma arquitetura específica, e propõe mecanismos ou alterações à arquitetura. Neste trabalho, é realizada uma investigação detalhada do impacto do subsistema de memória em diferentes arquiteturas, que é um dos aspectos mais importantes a serem considerados, utilizando o mesmo conjunto de aplicações. Neste estudo, foram realizados experimentos com as arquiteturas Ivy Bridge (CPU) e Kepler (GPU), e aplicações do conjunto de benchmarks SHOC. Desta forma, foi possível compreender o motivo de uma aplicação desempenhar bem em uma arquitetura e mal em outra.*

## 1. Introdução

A computação tem sido responsável por uma grande revolução científica. Através dos computadores, problemas que até então não podiam ser resolvidos, ou que demandavam muito tempo para serem solucionados, passaram a estar ao alcance da comunidade científica. A evolução das arquiteturas de computadores acarretou no aumento do poder computacional, ampliando a gama de problemas que poderiam ser tratadas computacionalmente. A introdução de circuitos integrados, *pipelines*, aumento da frequência de operação, execução fora de ordem e previsão de desvios constituem parte importante das tecnologias introduzidas até o final do século XX. Recentemente, tem crescido a preocupação com o gasto energético, com o objetivo de se atingir a computação em nível *exascale* de forma sustentável. Entretanto, as tecnologias até então mencionadas não possibilitam atingir tal objetivo, devido ao alto custo energético de se aumentar a frequência e estágios de *pipeline*, assim como a chegada nos limites de exploração do paralelismo a nível de instrução [Borkar and Chien 2011, Coteus et al. 2011].

A fim de se solucionar tais problemas, arquiteturas paralelas e heterogêneas foram introduzidas nos últimos anos. A principal característica de arquiteturas paralelas é a presença de vários núcleos de processamento operando concorrentemente, de forma que a aplicação deve ser programada separando-a em diversas tarefas que se comunicam entre si. Em relação à arquiteturas heterogêneas, sua principal característica é a presença de diferentes ambientes em um mesmo sistema, cada um com sua própria arquitetura especializada para um tipo de tarefa. A utilização de aceleradores é uma das principais formas adquiridas por arquiteturas heterogêneas, no qual um processador genérico é responsável principalmente pela gerência do sistema, e diversos aceleradores presentes no sistema realizam a computação de determinados tipos de tarefas.

A utilização de arquiteturas paralelas e heterogêneas impõe diversos desafios para se obter um alto desempenho [Mittal and Vetter 2015]. As aplicações precisam ser codificadas considerando as particularidades e restrições de cada ambiente, assim como considerando suas características arquiteturais distintas. Por exemplo, na hierarquia de memória, a presença de diversos níveis de memória cache, alguns compartilhados e outros privados, bem como se os bancos de memória encontram-se centralizados ou distribuídos, introduz tempos de acesso não uniformes, o que gera um grande impacto no desempenho [Cruz et al. 2016]. Isto é ainda mais crítico em arquiteturas heterogêneas, visto que cada acelerador pode possuir sua própria, e distinta, hierarquia de memória. Além disso, nas arquiteturas heterogêneas, o número de unidades funcionais pode variar entre os diferentes aceleradores, sendo que o próprio conjunto de instruções pode também não ser o mesmo. Neste contexto, é importante desenvolver técnicas para análise de desempenho e do comportamento de arquiteturas paralelas e heterogêneas, a fim de se propiciar um melhor suporte para desenvolvedores, para que os mesmos tenham melhores condições de otimizar suas aplicações.

Este trabalho tem como objetivo realizar uma análise detalhada do impacto do subsistema de memória em diferentes arquiteturas. A proposta consiste em fazer uso dos contadores de *hardware* para ter medidas precisas do impacto real de diferentes fatores que influenciam no acesso à memória. Foram analisados contadores que medem o uso de memória *cache*, memória primária, tráfego em interconexões, dentre outros. Desta forma, foi possível obter uma compreensão detalhada de como diferentes aspectos da hierarquia

de memória impactam no desempenho das aplicações. Tal estudo pode servir de base para que os desenvolvedores das aplicações paralelas possam otimizar suas aplicações.

O artigo está organizado da seguinte forma. A Seção 2 apresenta os trabalhos relacionados. A Seção 3 demonstra a importância de se analisar em detalhes o comportamento de aplicações em diferentes arquiteturas. A Seção 4 explica a metodologia empregada nos experimentos. A Seção 5 exibe os resultados. Para finalizar, a Seção 6 apresenta as conclusões e trabalhos futuros.

## 2. Trabalhos Relacionados

Em [Mei and Chu 2015], é realizada uma dissecção e análise das características do subsistema de memória em 3 arquiteturas de GPU distintas: Fermi, Kepler e Maxwell. Os autores usam um *benchmark* de perseguição de ponteiro e a latência observada para definir as dimensões de todas as memórias dentro de cada GPU. Assim, chegam às características básicas e conseguem identificar peculiaridades interessantes, tal como uma política de substituição de linhas diferente da *Least Recently Used* (LRU) na memória *cache* L2. O artigo conclui que o planejamento da arquitetura Kepler foi agressivo em sua banda de memória, que acabou por muitas vezes inutilizada, e que na arquitetura Maxwell mais recursos foram investidos em memória compartilhada, gerando um sistema mais eficiente e balanceado. No presente artigo, o foco é observar o efeito destas características em *benchmarks* reais, especialmente no que concerne o desempenho dos mesmos.

Em [Ausavarungnirun et al. 2015], os autores propõe um mecanismo para balancear acessos à memória. A principal observação do artigo é de que *warps* com vários *hits* na *cache* L2, mas que também geram *misses* na *cache* L2, tem como gargalo estes mesmos *misses*, apesar de todos os *hits*, que ficam inutilizados. *Hits* em *warps* com muitos *misses* também são inúteis, pois o tempo do pior acesso sempre define a execução do *warp*. Através de mudanças no subsistema de memória, priorizando acessos de *warps* com maioria de *hits* e redirecionando acessos de *warps* com maioria de *misses* diretamente para a memória principal, a técnica é capaz de ganhar em média 21% de desempenho, com execução 20% mais eficiente. O trabalho explora uma característica arquitetural inerente ao modelo de *stream processor*, mitigando o problema de divergência de acessos de memória em *warps* individuais. Já a pesquisa considerada neste artigo trata de problemas relacionados à pressão de várias *threads* de *benchmarks* escaláveis em um nível de sistema, embora sejam problemas relacionados quando considerado o nível de *stream processor*.

Outro mecanismo é proposto em [Jia et al. 2014] para balancear acessos à memória. Neste artigo, os autores partem do princípio de que as memórias *cache* usadas em arquiteturas GPU são as mesmas projetadas para arquiteturas de CPU, o que prejudica o funcionamento das mesmas. O uso massivo de *threads* através de *block-threading* em *warps* significa que *caches* normais disponibilizam poucos *bytes* por *thread*, e quando todas *threads* necessitam de *cache*, todos dados são jogados fora rápido demais para ocorrer reuso (*thrashing*). Quando *threads* de múltiplos *warps* compartilham a *cache*, existe contenção pela própria fila de requisições, que não consegue suportar a pressão de tantas *threads*. Como soluções, os autores propõe 2 mecanismos: reordenamento de requisições através de filas e *cache bypassing*. Através de filas que usam identificadores de bloco, é possível separar as requisições e priorizar todas requisições de 1 bloco, de modo a obter

uso da localidade espacial e temporal deste bloco. Para evitar *starving* e contenções, são desenvolvidas políticas adicionais para balancear os casos de uso, como priorizar uma fila cheia que recebe uma nova requisição. Como as *caches* também não são dimensionadas para tantas threads, alguns *warps* tem todos seus acessos redirecionados diretamente para memória principal, efetivamente evitando acessos à memória *cache*. Isto melhora o acesso de todas as requisições à memória *cache*, pois os atrasos de espera na fila dos acessos que vão para memória *cache* é reduzido, e os acessos que não tem acesso à memória *cache*, por não terem prioridade, provavelmente seriam *misses* na memória *cache*. Ao evitar a espera e o acesso inútil, esta requisição é resolvida mais rápido ao ser direcionada diretamente para memória principal, caracterizando assim o *cache bypassing*.

A análise dos trabalhos relacionados demonstra como um profundo conhecimento do comportamento das aplicações a nível arquitetural permite desenvolver técnicas para ganhar desempenho. Neste contexto, o foco do presente artigo é estudar os efeitos de diferentes subsistemas de memória de arquiteturas CPU e GPU em aplicações paralelas, e assim analisar o impacto de que cada característica do subsistema de memória tem nas aplicações.

## 3. Desempenho de Aplicações Paralelas em Diferentes Arquiteturas

Nesta seção, é demonstrado porque é importante se realizar uma análise detalhada do comportamento das aplicações em diferentes arquiteturas no âmbito do comportamento do acesso à memória. Primeiro, é detalhado de como funciona o subsistema de memória da arquitetura de GPU que será usada nos experimentos. Logo após, é apresentado o funcionamento da arquitetura de memória da arquitetura para CPU. Por último, é exibido um experimento mostrando o desempenho de cada aplicação em cada arquitetura.

### 3.1. Subsistema de Memória da Arquitetura da GPU (Kepler)

Inicialmente, é necessário entender o subsistema de memória da arquitetura alvo. Na Figura 1, pode-se observar a base da arquitetura. A Kepler possui 1,5 MB de memória *cache* L2 para acessos globais a *stream processors*. A memória *cache* L2 possui um *prefetcher* sequencial associado a ela, o qual busca endereços em uma janela de tamanho equivalente a ⅔ do total de memória *cache* L2. A política de substituição de *cache* não é a comumente usada *least recently used* (LRU), demonstrando aperiodicidade em *microbenchmarks* [Mei and Chu 2015]. A memória global GDDR5 tem um *throughput* máximo de 215 GB/s. Embora não detalhado na Figura 1, os *translation look-aside buffers* (TLB) de primeiro nível e segundo nível estão ambos compartilhados entre todos os *stream processors*.

Um *stream processor* é constituído de 192 núcleos *CUDA*, sendo que cada núcleo compartilha um *register file* de 256 KB, com 64 KB registradores de 4 B. Adicionalmente, também compartilham acesso a 3 memórias *cache* internas ao *stream processor*. A primeira delas, a memória *cache* L1, é usada especificamente para acessos à pilha e para conter o *register spill*, ou seja, os valores em registradores em excesso aos suportados pela arquitetura. A *cache* é organizada em linhas de 128 B, gerando acessos à memória *cache* L2 em casos de dados não presentes. É possível utilizar a memória *cache* L1 para armazenar qualquer tipo de dado ao utilizar uma *flag* específica para o compilador. Neste artigo, a operação normal da memória *cache* foi usada, isto é, apenas dados locais da pilha e *register spill*.

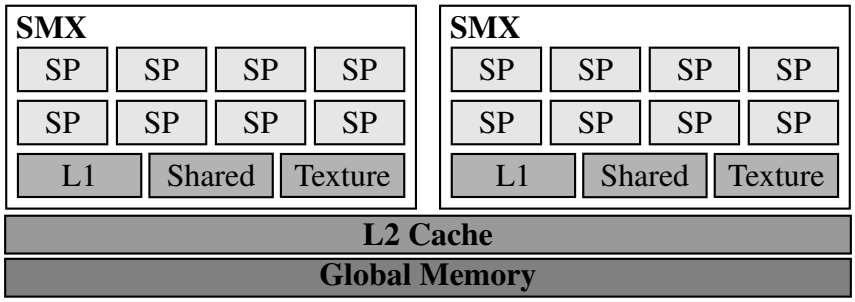



**Figura 1. Hierarquia de memória de uma GPU Kepler.**

A segunda *cache* interna ao *stream processor* é a memória *cache* compartilhada, a qual armazena dados locais alocados pelo programador com a diretiva *shared*, indicando que tais dados são compartilhados entre todas *threads*. O compilador configura automaticamente a divisão de espaço entre as *caches*, tendo três opções: 16 KB para *cache* L1 e 48 KB para *cache* compartilhada, 32 KB para cada, ou 48 KB para *cache* L1 e 16 KB para *cache* compartilhada. A terceira *cache* é a *cache* de leitura, a qual possui apenas dados de leitura. Originalmente ela era utilizada para texturas, mas na arquitetura Kepler qualquer dado pode ser alocado em seus 48 KB. A política usada é LRU. O compilador pode decidir automaticamente dados a serem armazenados nesta *cache* quando o programador usa a diretiva de C-99 `const restrict`. O programador também pode explicitamente usar esta *cache* através da intrínsica `lgd()`.

### 3.2. Subsistema de Memória da Arquitetura da CPU (Ivy Bridge)

A arquitetura Ivy Bridge [George et al. 2011] é uma arquitetura NUMA, do inglês *non-uniform memory access*. Em tais arquiteturas, a latência do acesso à memória irá variar dependendo de qual banco de memória será acessado [Cruz et al. 2016]. Cada processador na arquitetura contém um controlador de memória, formando desta forma um nó NUMA. Cada núcleo possui 2 núcleos lógicos, empregando uma tecnologia denominada *Hyper-Threading*. A conexão entre os diferentes processadores é realizada através do *QuickPath Interconnect* (QPI) [Ziakas et al. 2010]. A hierarquia de memória do Ivy-Bridge está ilustrada na Figura 2.

Para ilustrar como a hierarquia de memória influencia na latência do acesso à memória, a Figura 2 mostra um exemplo de arquitetura onde há diferentes possibilidades para o acesso à memória. As *threads* podem acessar a memória acertando a memória privada para cada núcleo, na *cache* L1 ou L2, obtendo assim o maior desempenho. As *threads* podem acessar a memória *cache* L3, compartilhada entre os núcleos de um mesmo processador. Caso não encontre os dados em alguma memória *cache* local, a arquitetura realiza a busca dos dados em uma memória *cache* remota, de outro processador, e, caso mesmo assim não encontre os dados, a memória principal é acionada.

### 3.3. Aplicações Paralelas em Diferentes Arquiteturas

Para melhor entender o impacto do subsistema de memória na eficiência da execução da GPU, fazem-se necessárias aplicações que demonstrem os gargalos da mesma. Na Figura 3, são demonstrados os tempos de execução para CPU e GPU de 6 *benchmarks* da suite SHOC (Scalable HeterOgeneous Computing) [Danalis et al. 2010]. Os *benchmarks Reduce* e *S3D* apresentam grande redução no tempo de execução quando

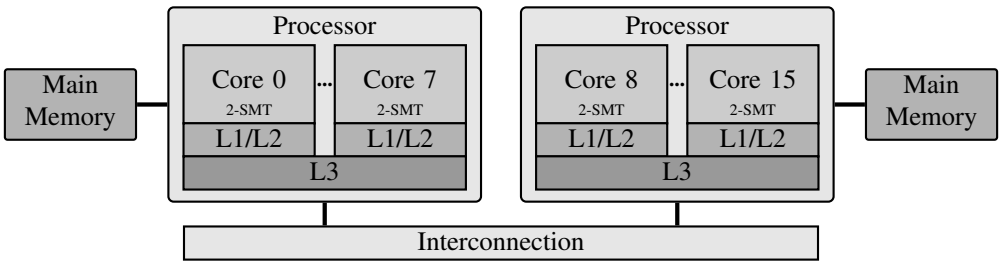



**Figura 2. Exemplo de subsistema de memória da arquitetura Ivy Bridge. Nesta figura, há 2 processadores, cada um constituindo um nó NUMA. Cada processador consiste de 8 núcleos de processamento, cada um representando 2 núcleos lógicos. Há 3 níveis de memória *cache*.**

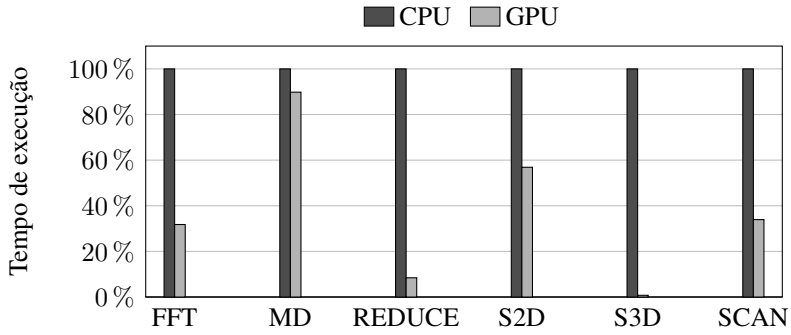



**Figura 3. Tempo de execução em diferentes arquiteturas, normalizado em relação à arquitetura CPU (quanto menor, melhor).**

comparados a uma CPU. Os *benchmarks S2D*, *FFT* e *Scan* apresentam redução aquém da esperada com o alto número de *threads* em uma GPU. Por fim, o *benchmark MD* apresenta pouca redução, indicando que o algoritmo não consegue usar o alto número de *threads* da GPU. Assim, torna-se interessante analisar o comportamento do subsistema de memória quando executando estas aplicações, para que possamos entender os gargalos em relação ao paralelismo e comunicação de cada uma delas. Tal análise ajuda não apenas o programador a entender as limitações inerentes à GPU na hora de programar, mas também é necessária para definir o foco das próximas arquiteturas.

## 4. Metodologia

Os experimentos foram realizados nos ambientes IvyBridge e Kepler. A IvyBridge possui dois processadores Intel Xeon E5-2640 v2, cada processador tem 8 *cores* físicos, permitindo execução de 32 *threads* com *Hyper-Threading*. A Kepler é uma GPU Tesla K20m com 2496 *CUDA Cores*. A Tabela 1 apresenta detalhes sobre cada ambiente.

O *benchmark* Scalable HeterOgeneous Computing (SHOC) foi utilizado pois implementa um conjunto de aplicações com diferentes características e dispõem de implementações para processadores (x86), placas gráficas (GPU) e coprocessadores Xeon Phi. Os experimentos apresentados na Seção 5 são a média de 30 execuções aleatórias. O desvio padrão apresentado é dado pela distribuição *t-student* com intervalo de confiança de $95\%$. Além de tempo de execução, investigamos outras métricas como utilização, largura de banda e taxa de acerto do subsistema de memória. As ferramentas Intel PCM [Intel 2012] e Intel VTune [Intel 2016] foram utilizadas na *IvyBridge*

**Tabela 1. Ambiente de Execução**

| Sistema | Parâmetro | Valor |
|---|---|---|
| *IvyBridge* | Processador | 2 × Xeon E5-2640 v2, 8 *cores*, 2 SMT-*cores* |
| | Microarquitetura | Ivy Bridge |
| | Caches/proc. | 8 × 32 KByte L1, 8 × 256 KByte L2, 20 MByte L3 |
| | Memória | 64 GByte DDR3-1600 |
| | Ambiente | Kernel Linux 3.16.0-70, GCC 4.8.4 |
| Kepler | Dispositivo | Tesla K20m |
| | Microarquitetura | Kepler GK110 |
| | CUDA Cores | 2496, 13 MPs × 192 Cores/MP |
| | Registradores | 13 × 256 KByte |
| | Cache | 13 × 64 KByte L1 / *shared*, 1280 KByte L2 |
| | | 13 × 48 KByte *texture (read-only)* |
| | Memória Global | 5 GByte GDDR5 |
| | CUDA | *Driver* 7.5, *Runtime* 7.5 |

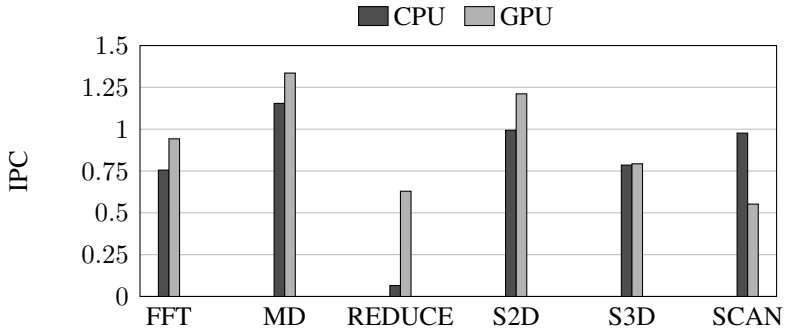



**Figura 4. Comparação de IPC entre CPU e GPU.**

e a nvprof [Nvidia 2016] na Kepler.

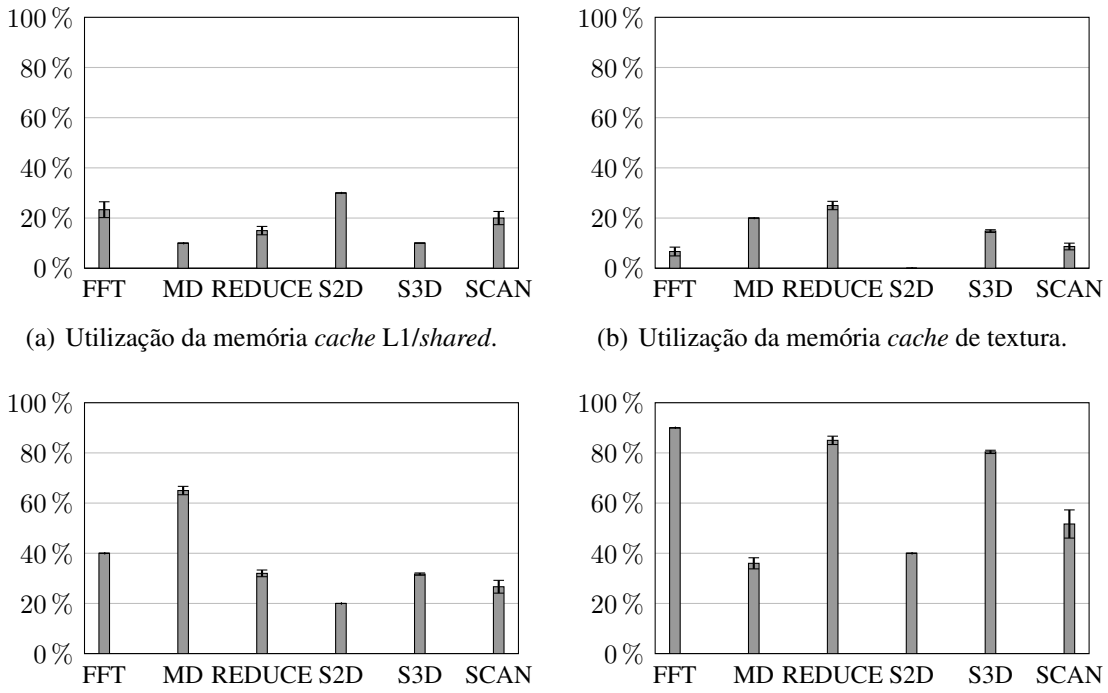
## 5. Experimentos

A Figura 4 apresenta o IPC (*Instructions per cycle*, ou, instruções por ciclo) que indica o número médio de instruções executadas por ciclo de *clock*. Os resultados mostram diferentes comportamentos por aplicação e ambiente de execução. FFT, MD e S2D tem melhores IPC na GPU sendo a diferença de desempenho entre CPU e GPU similar. S3D tem desempenho próximo na CPU e na GPU. Outros dois casos são Reduce, na qual o desempenho da GPU é $10\times$ maior que o da CPU, e Scan, onde a CPU é melhor.

Diferentes ganhos de desempenho motivam o estudo de características de cada aplicação e microarquitetura. Com isso, as subseções seguintes apresentam análises de desempenho das microarquiteturas Ivy Bridge (CPU) e Kepler (GPU), visando identificar características que impactam diretamente no desempenho de aplicações paralelas.

### 5.1. Utilização da memória (Kepler)

A arquitetura de uma GPU oferece diversos níveis e tipos de memória cada uma com suas peculiaridades. As memórias *L1/shared* e *texture (read-only)* são as mais




(a) Utilização da memória *cache* L1/*shared*.

(b) Utilização da memória *cache* de textura.

(c) Utilização da memória *cache* L2.

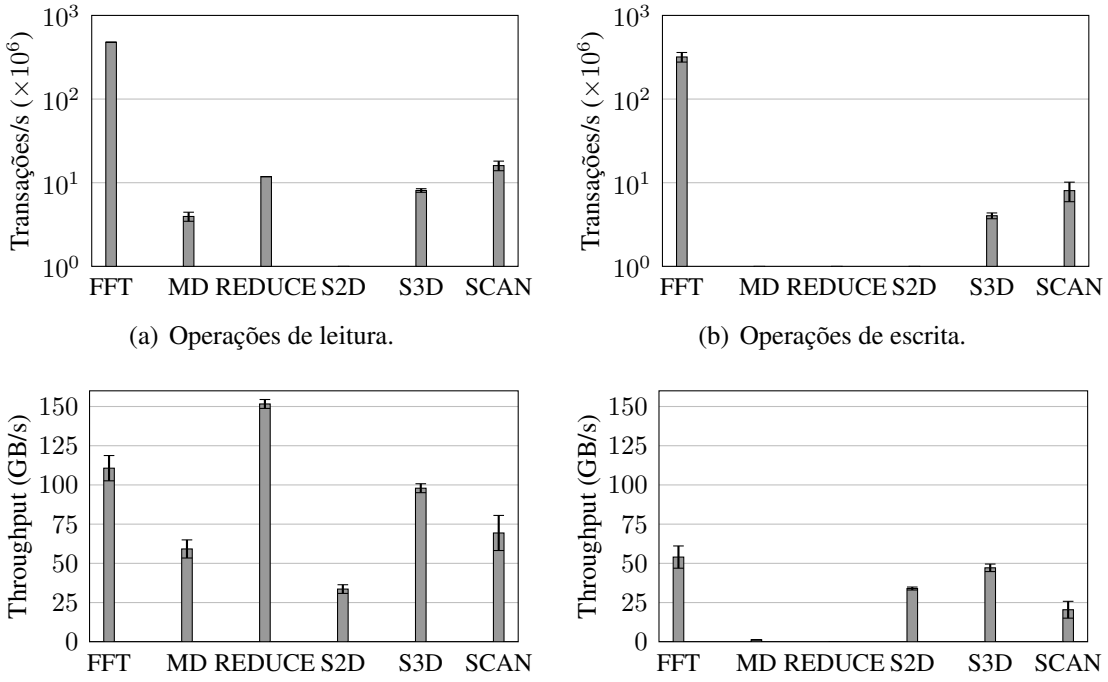
(d) Utilização da memória global.

**Figura 5. Utilização da hierarquia de memória na arquitetura Kepler.**

próximas das *threads* e tem as maiores taxas de transferência. As Figuras 5(a) e 5(b) apresentam a taxa de utilização desses recursos por aplicação. Em ambos os casos, a taxa de utilizacão é baixa, em média $18\%$ para memória *L1/shared* e $13\%$ para a *texture*. As aplicações com maior uso da *L1/shared* são S2D, FFT e Scan, o que indica maior uso de registradores e dados da pilha se comparados as outras aplicações. Além disso, as aplicações Reduce e MD são as que utilizam mais o recurso de *cache texture*. Nesse caso, ou muitas constantes são reutilizadas da pilha, ou o uso da *cache* se da por consequência do uso da memória *global texture*.

As memórias L2 e *global* (*device memory*) são maiores, mas tem taxas de transferência menores do que as *L1/shared* e *texture*. As Figuras 5(c) e 5(d) mostram que o uso dessas memórias é maior, sendo em média $36\%$ para L2 e $64\%$ para memória global. MD e FFT utilizam $65\%$ e $40\%$ da memória L2, respectivamente. O outro grupo de aplicações, formado por Reduce, S3D, Scan e S2D, tem comportamento semelhante, com taxas de utilização entre $20\%$ e $32\%$. A memória global é a mais lenta da hierarquia e tem uma alta taxa de utilização impactando diretamente no desempenho de aplicações paralelas.

Aplicações paralelas executadas em uma arquitetura de GPU fazem uso da memória *global* para leitura e escrita. As Figuras 6(a) e 6(c) apresentam, respectivamente, o número de operações de leitura e a taxa de transferência em operações de leitura. Em média, são feitas $55$ milhões de operações de leitura por segundo, sendo a maior taxa para FFT com $478$ milhões. Um segundo grupo, formado por MD, Reduce, S3D e Scan, tem valores entre $4$ e $16$ milhões. S2D tem o menor número de transações ($320$ *mil*) e o menor *throughput* ($3\ GB/s$). Reduce tem o maior *throughput*, mesmo não sendo a aplicação




(a) Operações de leitura.

(b) Operações de escrita.

(c) Taxa de transferência de leituras.

(d) Taxa de transferência de escritas.

**Figura 6. Operações na memória global na arquitetura Kepler.**

com mais transações.

Operações de escrita também predominam em um conjunto de aplicações paralelas. A taxa de transferência para operações de escrita e o número de operações de escrita são apresentadas nas Figuras 6(d) e 6(b). Em relação ao número de transações por segundo, FFT é a maior com $318$ milhões, após Scan e S3D com $8$ e $4$ milhões respectivamente. MD, Reduce e S2D tem valores inferiores aos demais. FFT e S3D tem um *throughput* semelhante, mesmo S3D tendo o número de transações $80\times$ menor. Esse comportamento ocorre pois a arquitetura Kepler permite transações de diferentes tamanhos (32, 64, 96 e 128B). Nesse caso, uma aplicação com *throughput* semelhante pode ter quantidade de transações e tamanho de cada transação diferentes.

## 5.2. Correção de Erros de Memória (Kepler)

Outro ponto que deve ser considerado em relação ao desempenho de uma aplicação paralela é a detecção de falhas durante a execução. Aplicações críticas como simulações médicas necessitam de integridade de dados, pois erros durante a simulação não são aceitáveis. Um método usado para detectar e corrigir erros é o *Error Checking and Correction* (ECC). Entretanto, esse método necessita de vários recursos da GPU, tendo como consequência a redução de desempenho.

É apresentado na Figura 7 o número de transações de ECC por aplicação. Aplicações com muitas transações de ECC como a FFT tem seu desempenho reduzido devido a grande quantidade de erros a serem corrigidos. Uma opção para aplicações que tem como requisito principal o desempenho é desativar o ECC ou aplicar técnicas de correção de erros em *software*.

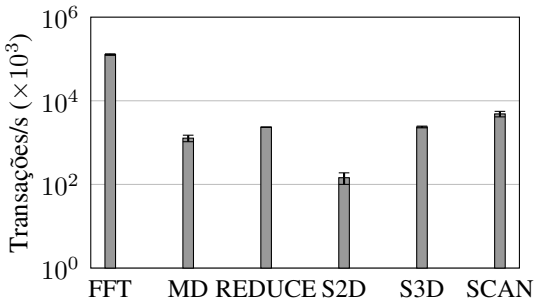



**Figura 7. Transações de ECC.**

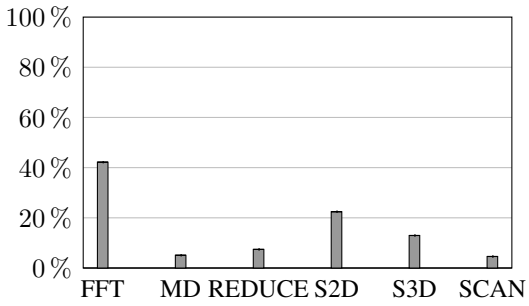



**Figura 8. Instruções de controle de fluxo.**

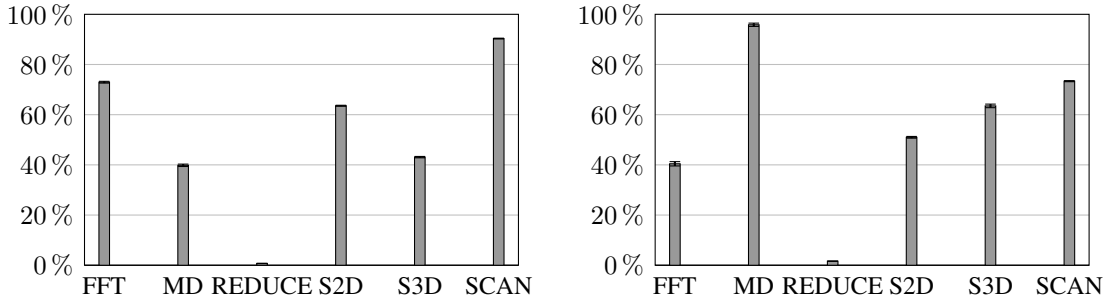
### 5.3. Instruções de Desvios (Kepler)

A questão dos desvios na GPU é citada pelo fabricante, que orienta desenvolvedores a fazer com que todas *threads* de um *warp* acessem a regiões de memória contíguas, para, com isso, atingir altos *throughputs* de acesso a memória. Desvios (*branching*) tem seu desempenho limitado em GPUs devido a essa característica da arquitetura [Cook 2012]. CPUs implementam um preditor de saltos que prediz se o desvio será tomado ou não antes do resultado da operação. As instruções continuam a serem buscadas e, se a predição foi correta, não existe penalidade no desempenho. No caso das GPUs, no primeiro momento, são executadas as *threads* da condição verdadeira e as outras são marcadas como inativas. Em um segundo momento, é feito o contrário para execução da condição falsa. A Figura 8 mostra o número de instruções de controle de fluxo executadas por cada aplicação. As aplicações com mais instruções desse tipo são FFT e S2D, sendo que cada situação de divergência entre as *threads* acarreta uma redução no desempenho, devido a necessidade de mais momentos de execução. Uma solução para esse problema é criar *kernels* menores e executar as instruções de controle de fluxo na CPU.

### 5.4. Memória (Ivy Bridge)

A Seção 5.1 apresentou uma discussão sobre o uso da hierarquia de memória da GPU. Os experimentos mostraram que a hierarquia da GPU tem um compartamento diferente do esperado em uma CPU. Isso ocorre devido às memórias *cache* da CPU e GPU terem características de projeto diferentes. CPUs tem grandes *caches* com objetivo de reduzir a longa latência da memória principal para latências menores em níveis de *cache* mais próximos ao processador. De outra forma, *caches* de GPUs visam o *throughput* de acesso a memória, permitindo a carga de vários operandos por unidade de tempo [Kirk and Wen-mei 2012].

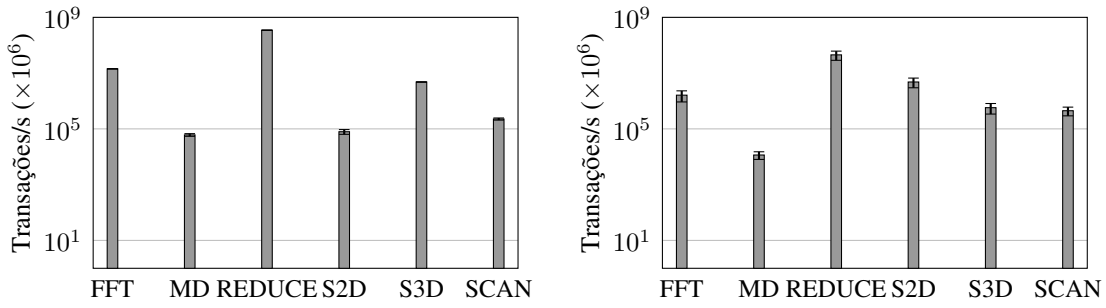
As taxas de acerto nas memórias *cache* L2 e L3 são apresentadas nas Figuras 9(a) e 9(b). A taxa de acerto média da L2 foi de $52\%$ e da L3 de $54\%$. Aplicações como FFT, MD e Scan tiram melhor proveito da *cache*, devido ao padrão de acesso aos dados. Por outro lado, existem aplicações que tem uma taxa de acerto baixa. Nesse caso, o desempenho da aplicação é afetado porque são feitos muitos acessos à memória principal. Esse comportamento pode ser confirmado nas Figuras 10(a) e 10(b), onde são apresentadas transações em DRAM por segundo e transações *Quickpath interconnect* (QPI) por segundo. Ambos acessos à DRAM e QPI afetam diretamente o IPC da aplicação. Um exemplo é a aplicação MD, que tem o maior IPC da CPU e as menores taxas de acessos a DRAM e QPI.




(a) Taxa de acerto na cache L2.

(b) Taxa de acerto na cache L3

**Figura 9. Taxa de acerto em memórias *cache* na Ivy Bridge.**




(a) Transações em DRAM.

(b) Transações pela QPI.

**Figura 10. Operações de memória na Ivy Bridge.**

## 6. Conclusão e Trabalhos Futuros

Este trabalho analisou o impacto do subsistema de memória das arquiteturas Ivy Bridge (CPU) e Kepler (GPU). Para isso, foram utilizados contadores de *hardware*, os quais auxiliaram na compreensão de aspectos da hierarquia de memória. Os resultados mostram que, em ambas arquiteturas, o maior limitador de desempenho são os acessos à memória principal.

Na arquitetura Kepler, a taxa de utilização da memória global é diretamente ligada ao desempenho da aplicação. Mesmo com a alta largura de banda da memória, aplicações como Reduce tem seu desempenho limitado. O melhor caso ocorre com pouca utilização da memória global e muita utilizacão das memórias *cache*. Esse é o caso da MD, com alta utilização de L2 e pouca utilização da memória global. O comportamento na arquitetura Ivy Bridge é semelhante. Aplicações com altas taxa de acerto na *cache* tem o melhor desempenho. Reduce tem taxa de acerto próxima a zero e isso acarreta demasiados acessos à memória primária, o que limita seu desempenho, devido à latência do alto número de transações resolvidas pela DRAM ou transferidas através da interconexão QPI.

Trabalhos futuros incluem análise da arquitetura Knights Landing do processador Xeon Phi, bem como um estudo da eficiência energética de diferentes arquiteturas.

## Agradecimentos

Os resultados relatados neste trabalho se dão em virtude do convênio entre a Hewlett Packard Enterprise (HPE) e a Universidade Federal do Rio Grande do Sul (UFRGS),

alcançados com recursos provenientes da contrapartida da isenção ou redução do IPI concedida pela Lei nº 8.248, de 1991 e suas atualizações posteriores. Parcialmente financiado pela CAPES, CNPq e RNP/UE no âmbito do projeto HPC4E (www.hpc4e.eu), n° 689772.

## Referências

[Ausavarungnirun et al. 2015] Ausavarungnirun, R., Ghose, S., Kayiran, O., Loh, G. H., Das, C. R., Kandemir, M. T., and Mutlu, O. (2015). Exploiting inter-warp heterogeneity to improve gpgpu performance. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 25–38. IEEE.

[Borkar and Chien 2011] Borkar, S. and Chien, A. A. (2011). The future of microprocessors. *Communications of the ACM*, 54(5):67–77.

[Cook 2012] Cook, S. (2012). *CUDA programming: a developer's guide to parallel computing with GPUs*. Newnes.

[Coteus et al. 2011] Coteus, P. W., Knickerbocker, J. U., Lam, C. H., and Vlasov, Y. A. (2011). Technologies for exascale systems. *IBM Journal of Research and Development*, 55(5):14–1.

[Cruz et al. 2016] Cruz, E. H., Diener, M., Alves, M. A., Pilla, L. L., and Navaux, P. O. (2016). Lapt: A locality-aware page table for thread and data mapping. *Parallel Computing (PARCO)*, 54:59 – 71.

[Danalis et al. 2010] Danalis, A., Marin, G., McCurdy, C., Meredith, J. S., Roth, P. C., Spafford, K., Tipparaju, V., and Vetter, J. S. (2010). The scalable heterogeneous computing (SHOC) benchmark suite. In *Proceedings of 3rd Workshop on General Purpose Processing on Graphics Processing Units, GPGPU 2010, Pittsburgh, Pennsylvania, USA, March 14, 2010*, pages 63–74.

[George et al. 2011] George, V., Piazza, T., and Jiang, H. (2011). Technology insight: Intel next generation microarchitecture codename ivy bridge. In *Intel Developer Forum*.

[Intel 2012] Intel (2012). Intel Performance Counter Monitor - A better way to measure CPU utilization.

[Intel 2016] Intel (2016). Intel VTune Amplifier XE 2016.

[Jia et al. 2014] Jia, W., Shaw, K. A., and Martonosi, M. (2014). Mrpb: Memory request prioritization for massively parallel processors. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 272–283. IEEE.

[Kirk and Wen-mei 2012] Kirk, D. B. and Wen-mei, W. H. (2012). *Programming massively parallel processors: a hands-on approach*. Newnes.

[Mei and Chu 2015] Mei, X. and Chu, X. (2015). Dissecting GPU memory hierarchy through microbenchmarking. *CoRR*, abs/1509.02308.

[Mittal and Vetter 2015] Mittal, S. and Vetter, J. S. (2015). A survey of cpu-gpu heterogeneous computing techniques. *ACM Computing Surveys (CSUR)*, 47(4):69:1–69:35.

[Nvidia 2016] Nvidia (2016). Developer Zone - CUDA Toolkit Documentation.

[Ziakas et al. 2010] Ziakas, D., Baum, A., Maddox, R. A., and Safranek, R. J. (2010). Intel QuickPath Interconnect - Architectural Features Supporting Scalable System Architectures. In *Symposium on High Performance Interconnects (HOTI)*, pages 1–6.