Universidade Federal do Piauí
Centro de Ciências da Natureza
Programa de Pós-Graduação em Ciência da Computação

METAL: Uma Plataforma *Manycore* de Propósito Geral Adaptada ao Modelo de Programação OpenCL

Ramon Santos Nepomuceno

Teresina-PI, Setembro de 2016

Ramon Santos Nepomuceno

METAL: Uma Plataforma *Manycore* de Propósito Geral Adaptada ao Modelo de Programação OpenCL

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da UFPI (área de concentração: Sistemas de Computação), como parte dos requisitos necessários para a obtenção do Título de Mestre em Ciência da Computação.

Universidade Federal do Piauí – UFPI

Centro de Ciências da Natureza

Programa de Pós-Graduação em Ciência da Computação

Orientador: Prof. Dr. Ivan Saraiva Silva

Coorientador: Prof. Dr. Gustavo Girão Barreto da Silva

Teresina-PI

Setembro de 2016

Ramon Santos Nepomuceno

METAL: Uma Plataforma *Manycore* de Propósito Geral Adaptada ao Modelo de Programação OpenCL

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da UFPI (área de concentração: Sistemas de Computação), como parte dos requisitos necessários para a obtenção do Título de Mestre em Ciência da Computação.

Trabalho aprovado. Teresina-PI, 30 de setembro de 2016:

Prof. Dr. Ivan Saraiva Silva
Orientador

Prof. Dr. Gustavo Girão Barreto da Silva
Co-Orientador

Prof. Dr. Kelson Rômulo Teixeira Alves
Professor

Prof. Dr. Raimundo Santos Moura
Professor

Prof[a]. Dr. Mônica Magalhães Pereira
Professor

Teresina-PI
Setembro de 2016

À minha mãe Raimunda,
por sempre ter acreditado em mim.

Agradecimentos

Agradeço a toda a minha família por ter me dado todo o suporte que precisei. Minha tia Lucilene, meu padrinho Lourival e minha prima Vivia por terem me acolhido. Minha madrinha Conceição e meu padrinho Filho que, estando perto ou longe, sempre olharam por mim.

A minha irmã Amanda, meu tio Vicente e minha mãe Raimunda por fazerem parte do meu núcleo familiar.

A todos meus companheiros de UFPI e do Cesla (Pedro, Matheus, Jefferson, Tiago, Laysson, Jônatas, Jonathas, Natasha, Eugênio, Juninho, Patrocínio, Chaguinha) pelas incontáveis horas de convívio e cumplicidade.

Aos meus queridos amigos Magno, Hellen, Mateus e Sales.

Agradeço ao meu orientador, Ivan Saraiva Silva, por toda a ajuda, amizade, liderança e por sempre exigir nada menos que o meu máximo. A todos os meus professores da UFPI que participaram da minha formação.

Ao CNPq pelo apoio financeiro para realização deste trabalho de pesquisa.

A todos, meu sincero muito obrigado.

"E se o mundo não corresponde em todos os aspectos a nosso desejos, é culpa da ciência ou dos que querem impor seus desejos ao mundo."

(Carl Sagan)

Resumo

O termo GPGPU (*General Purpose Graphics Processing Unit*) é utilizado para denominar o uso de GPUs (*Graphics Processing Unit*), em conjunto com CPUs (*Central Processor Unit*), para computação de propósito geral. Embora esses sistemas sejam capazes de executar aplicações de propósito geral e SIMD (*Single Instruction Multiple Data*), CPUs e GPUs são projetadas separadamente, portanto, apresentam características distintas que dificultam a maximização do potencial dessas arquiteturas quando utilizadas em conjunto.

Este trabalho apresenta METAL (*ManycorE plaTform Adapted to opencL*), uma plataforma *manycore* baseada em rede em *chip*, cuja arquitetura foi concebida com o propósito de executar, nativamente, tanto aplicações SIMD quanto de propósito geral. A programação SIMD é realizada utilizando a linguagem OpenCL, cuja execução é facilitada pelo uso de um escalonador de *work-items* e de um modelo de memória compatível. Os núcleos de processamento utilizados na plataforma são processadores de propósito geral MIPS que, com o suporte de recursos em *hardware*, permitem o uso de programação paralela utilizando a técnica de exclusão mútua.

Algumas aplicações foram desenvolvidas para validar e avaliar o desempenho, bem como a programabilidade da arquitetura. Os algoritmos Dijkstra, multiplicação de matrizes, problema das N rainhas com solução por algoritmo genético paralelo, jantar dos filósofos e detecção de borda aplicando o filtro laplaciano foram desenvolvidos e executados em um simulador implementado em SystemC. Os experimentos mostram o desempenho da plataforma ao executar os algoritmos e impacto que a comunicação utilizando a rede em *chip* exerce nas aplicações.

Palavras-chaves: Arquiteturas Heterogêneas. CPU. GPU. GPGPU. *Manycore*. OpenCL. Rede em Chip. SIMD.

Abstract

The GPGPU term (General Purpose Graphics Processing Unit) describes the use of GPUs (Graphics Processing Unit), in conjunction with CPU (Central Processor Unit) for general purpose computing. While these systems are able to run general purpose and SIMD (Single Instruction Multiple Data) applications, CPUs and GPUs are designed separately, thus have different characteristics that make it difficult to maximize the potential of these architectures when used together.

This Dissertation work presents METAL (ManycorE plaTform Adapted to opencL) a manycore platform network on chip based, whose architecture was designed in order to run natively both SIMD applications as general purpose. The SIMD programming is performed using OpenCL language, whose execution is facilitated by the use of a work-item's scheduler and a adapted memory model. The processing core used in the platform is the general purpose processors MIPS that, with the support of hardware resources, enable the use of parallel programming using mutual exclusion technique.

Some applications were developed to validate and evaluate the performance and programmability of the architecture. The algorithms: Dijkstra; matrix multiplication; the N-queens's problem solution with a parallel genetic algorithms; Dining philosophers problem and Laplacian Edge Detector have been developed and implemented in a SystemC simulator. The experiments shows the platform's performance when running the algorithms and the impact that the communication using the network chip has on applications.

Keywords: Heterogeneous Architectures. CPU. GPU. GPGPU. Many Core. OpenCL. Network on Chip. SIMD.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

ASIC	*Application Specific Integrated Circuits*
API	*Application Programming Interface*
BIC	*Bus Interface Controller*
CBEA	*Cell Broadband Engine Architecture*
CDFG	*Control Data Flow Graph*
CGRA	*Coarse Grained Reconfigurable Array*
CLEM	*OpenCL Compiler Metal*
CPU	*Central Processor Unit*
CU2CL	*CUDA to OpenCL*
CUDA	*Unified Device Architecture*
DMA	*Direct Memory Access*
DSP	*Digital Signal Processor*
EP	*Elemento de Processamento*
EIB	*Element Interconnect Bus*
FEM	*Finite Element Method*
FIFO	*First In First Out*
FLIT	*Flow Control Unit*
FPGA	*Field Programmable Gate Array*
GP	*General Purpose*
GPGPU	*General Purpose Graphics Processin Unit*
GPU	*Graphics Processing Unit*
HDL	*Hardware Description Language*
HI	*Host Interface*

ISA	*Instruction Set Architecture*
METAL	*Manycore Platform Adapted to OpenCL*
MIC	*Memory Interface Bus*
MIPS	*Microprocessor without interlocked pipeline stages*
MIMD	*Multiple Instruction Multiples Data*
MON	*Master Input NoC*
MI	*MMU Interface*
MISD	*Multiple Instruction Single Data*
MM	*Mutex Memory*
MMU	*Memory Managemente Unit*
MOH	*Master OpenCL Hardware Manager*
MON	*Master Output NoC*
MP	*Master Processor*
MPI	*Message Passing Interface*
MPMD	*Multiple Program Multiple Data*
NI	*Network Interface*
NoC	*Network on Chip*
OpenACC	*Open Accelerator*
OpenCL	*Open Computing Language*
OpenGL	*Open Graphics Library*
OpenMP	*Open Multi-Processing*
POSIX	*Portable Operating System Interface*
PPE	*Percentual Médio de Espera*
PMP	*Percentual Médio de Processamento*
PPE	*Power Processing Element*
RISC	*Reduced Instruction Set Computer*

SCC	*Single-chip Cloud Computer*
SDK	*Software development kit*
SI	*Slave Interface*
SIGGRAPH	*Special Interest Group Graphics and Interactive Techiniques*
SIMD	*Single Instruction Multiple Data*
SIN	*Slave Input Noc*
SISD	*Single Instruction Single Data*
SLS	*Stream Load Store*
SM	*Stream Multiprocessor*
SP	*Stream Processor*
SP	*Slave Processor*
SPE	*Synergistic Processing Element*
SoCIN	*SoC Interconnection Network.*
SOH	*Slave OpenCL Hardware Manager*
SON	*Slave Output Noc*
SPMD	*Single Program Multiple Data*
TE	Tempo de Espera
TLB	*Translation Lookaside Buffer*
TP	Tempo de Processamento
TU	*Translator Unit*
UC	Unidade de Computação
VHDL	*VHSIC Hardware Description Language*
VHSIC	*Very High Speed Integrated Circuits*
VLIW	*Very Large Instruction Word*

Sumário

1 Introdução

No evento Siggraph (*Special Interest Group Graphics and Interactive Techniques*) realizado em 2003, foi relatada uma expansão do uso de GPU's (*Graphic Processors Units*) para além de processamento de imagem. A partir daquele ano, as GPU's em conjunto com CPU's (*Central Processor Unit*) passaram a ser utilizadas em programação de propósito geral. Para esses tipos de sistemas, GPU's em conjunto com CPU's, foi empregado o termo GPGPU (*General Purpose Graphic Processor Unit*) (MACEDONIA, 2003).

Os sistemas GPGPU's popularizaram-se rapidamente entre os interessados em computação de alto desempenho. Atualmente, além de empregarem CPU's e GPU's, esses sistemas incorporaram outros aceleradores de aplicações, o que os caracteriza como **Sistemas Heterogêneos**.

Para facilitar a portabilidade de código entre essa variedade de arquiteturas, a indústria de semicondutores se uniu para criar o padrão de programação OpenCL (*Open Computing Language*) (Khronos Group, 2010). Além da criação desse padrão, a indústria de semicondutores deu, nos últimos anos, sinais de que os sistemas heterogêneos serão a principal abordagem referente à computação de alto desempenho. Exemplos dessa tendência são os processadores *AMD Fusion* (BRANOVER; FOLEY; STEINMAN, 2012) e o *Intel Sandy Bridge* (ROTEM et al., 2012). Essas duas arquiteturas integram CPU's (*multicores*) e GPU's em um mesmo chip. Tal abordagem é adotada com o intuito de prover melhor taxa de transferência de dados entre os elementos de processamento, além de menor consumo de energia.

Além da indústria, a comunidade acadêmica também tem mostrado bastante interesse nos sistemas heterogêneos. Kumar et al. (2003), por exemplo, é apresentada uma arquitetura *multicore* heterogênea, dotada de um mecanismo para reduzir dissipação de potência. Esse mecanismo consiste em um sistema de *software* que, durante a execução de uma aplicação, escolhe dinamicamente o elemento de processamento mais apropriado para uma dada restrição de performance e energia. Outro exemplo é mostrado por Pericas et al. (2007), que propõe um processador *multicore* heterogêneo, flexível e capaz de se reconfigurar para atender as demandas de uma aplicação sem a necessidade de intervenção do programador.

Todo esse interesse, tanto da indústria de semicondutores quanto da comunidade acadêmica, em arquiteturas heterogêneas, principalmente GPGPU's, mostra a importância da realização de pesquisas relacionadas ao desenvolvimento de arquiteturas que procurem unir o melhor desses dois mundos: a flexibilidade da programação de propósito geral com a eficiência da programação SIMD. Esta Dissertação de mestrado se insere exatamente

nesse contexto.

1.1 Motivação

Embora as GPU's atuais apresentem desempenho significativo, seus componentes (CPU's e GPU's) são projetados separadamente, mesmo quando integradas em um mesmo chip. Deste modo, apresentam características distintas que dificultam a maximização do potencial desses componentes quando utilizadas em conjunto.

As características de cada um desses componentes são otimizadas levando em consideração seus objetivos. As CPU's, por exemplo, são processadores de propósito geral, portanto são otimizadas para executar programas escritos usando linguagens de alto nível tradicionais, onde o paralelismo é extraído automaticamente do código sequencial. Já as GPU's são otimizadas para executar aplicações escritas na forma SIMD, onde o paralelismo é expresso explicitamente pelo programador. O modelo de memória também é distinto em cada uma dessas arquiteturas. As CPU's geralmente implementam níveis de memória cache a fim de explorar o princípio da localidade temporal e espacial, já as GPU's utilizam memórias internas com tamanhos reduzidos e dedicadas para tarefas específicas. Outra diferença é em relação à quantidade de elementos de processamento que cada componente dispõe, enquanto as CPU's modernas apresentam poucos e complexos núcleos de processamento, as GPU's apresentam muitos núcleos com pequena complexidade.

Portanto, este trabalho de Dissertação propõe uma plataforma *manycore* com recursos nativos para programação SIMD e propósito geral. A programação SIMD é feita utilizando OpenCL. A escolha dessa linguagem deve-se ao fato dela ser independente de plataforma, permitindo assim portabilidade de código, além de já estar bastante popular entre os programadores de arquiteturas heterogêneas.

1.2 Objetivos

Este trabalho de mestrado tem como objetivo apresentar a implementação da plataforma METAL. Esta possui recursos em *hardware* para executar aplicações de propósito geral e SIMD.

1.2.1 Objetivos Específicos

Os objetivos específicos deste trabalho de dissertação são:

- Implementar a plataforma METAL utilizando a biblioteca C++ de descrição de *hardware* SystemC;
- Avaliar o desempenho da plataforma METAL utilizando aplicações reais;

- Apresentar os pontos fortes e fracos da plataforma METAL.

1.3 Organização do Documento

O restante do documento está organizado em cinco capítulos:

1. *Fundamentação Teórica:* apresenta conceitos básicos para o entendimento do trabalho, são eles: classificação de arquiteturas de computadores, arquiteturas paralelas, *multicores*, GPU's, arquiteturas heterogêneas, linguagens de programação paralela, OpenCL e redes em chip;

2. *Estado da Arte:* apresenta alguns trabalhos relacionados ao suporte em *hardware* para SIMD e pesquisas relacionadas ao OpenCL;

3. *A Plataforma METAL*: apresenta a plataforma METAL, suas características e modo de funcionamento;

4. *Experimentos*: apresenta os resultados obtidos com a execução das aplicações testes;

5. *Conclusões e trabalhos futuros*: Fornece uma perspectiva geral do trabalho, assim como as conclusões e trabalhos futuros.

2 Fundamentação Teórica

O termo Arquitetura de Computadores muitas vezes é utilizado como sinônimo para o conjunto de instruções que um processador disponibiliza ao programador, do inglês *Instruction Set Architecture* (ISA), além de um conjunto de definições sobre seu uso. Neste trabalho, esse termo será utilizado para fazer referência não só a ISA de um processador, mas também a sua respectiva implementação, o que inclui outros dois aspectos: organização e *hardware* (HENNESSY; PATTERSON, 2011).

Organização refere-se aos aspectos de alto nível do projeto de um processador, tais como: o modelo de memória e as implementações das unidades lógicas, aritméticas, de salto e transferência de dados. Com fins ilustrativos podem ser citados os processadores AMD Opteron (KELTCHER et al., 2003) e o Intel Core i7, ambos processadores implementam o conjunto de instruções x86, mas possuem diferentes níveis de pipeline e organização de cache. O termo microarquitetura também pode ser utilizado como sinônimo para organização.

Hardware refere-se às especificidades do processador e está mais relacionado com a tecnologia de fabricação. Frequentemente, uma linha de processadores possui ISA's idênticas e organização bem parecidas, mas eles diferem em detalhes de encapsulamento. Por exemplo, o Intel Core I7 e o Intel Xeon 7560 são praticamente idênticos, com exceção da frequência de operação e do sistema de memória (HENNESSY; PATTERSON, 2011).

Por se tratar de uma revisão da fundamentação teórica, o leitor pode dispensar a leitura deste capítulo se possuir conhecimentos básicos sobre os assuntos tratados, já que nenhum deles será abordado em profundidade. O capítulo está divido em nove seções. Na primeira seção será apresentada a Taxonomia de Flynn, que representa uma dentre as diversas formas pelas quais pode-se classificar as arquiteturas de computadores. Em seguida será feita uma revisão sobre arquiteturas paralelas, seguido de processadores multicores, GPU's e Arquiteturas Heterogêneas. Posteriormente serão apresentados alguns aspectos sobre linguagens de programação paralela, seguido por uma seção sobre OpenCL. A oitava seção aborda o assunto redes em chip, com foco na rede SoCIN. Por fim, uma seção com as considerações finais.

2.1 Taxonomia de Flynn

Em 1966 Michael Flynn desenvolveu um sistema de classificação para arquiteturas de computadores que ficou conhecido como Taxonomia de Flynn (FLYNN, 1966). Esse sistema foi desenvolvido enquanto Flynn estudava computação paralela. Por esse motivo,

levou em consideração dois aspectos que podem ser gerenciados em paralelo durante a execução de um programa: instruções e dados. Combinando esses dois fatores, Flynn chegou a um total de quatro classes:

- **SISD (*Single Instruction Single Data*):** Nessa classe encontram-se os processadores monoprocessados, ou seja, processadores com um único elemento de processamento, baseado na arquiteturea original de Von Neumann (NEUMANN, 1993). Podem executar somente uma única instrução sobre um único dado por vez. Isto significa que não existe nenhum tipo de paralelismo nesta classe. Exemplos de processadores SISD são os processadores Intel desde o modelo 8086 até o modelo 80486 (MAHMMOD, 2013);

- **SIMD (*Single Instruction Multiple Data*):** As arquiteturas dessa classe possuem múltiplos elementos de processamento, mas uma única instrução é despachada ao mesmo tempo em todos eles, que por sua vez, operam sobre diferentes unidades de dados. Essas arquiteturas exploram o paralelismo em nível de dados. Os processadores vetoriais foram as primeiras implementações desse tipo de arquitetura;

- **MISD (*Multiple Instruction Single Data*):** Nessa classe encontram-se as arquiteturas que possuem múltiplos elementos de processamento e podem executar várias instruções sobre um único conjunto de dados. Essas arquiteturas não são muito comuns, geralmente só são implementadas em cenários específicos. Halaas et al. (2004), por exemplo, é proposta uma arquitetura MISD que tem a finalidade de reconhecer padrões em grandes quantidades de dados;

- **MIMD (*Multiple Instructions Multiples Data*):** Assim como nas classes SIMD e MISD, essa classe também possui múltiplos elementos de processamento. A diferença é que cada um pode executar diferentes instruções operando sobre distintos dados em paralelo. É a classificação mais flexível, podendo explorar qualquer tipo de paralelismo. O conceito MIMD pode ainda ser subdivido em duas subcategorias: SPMD (*Single Program Multiple Data*) e MPMD (*Multiple Programs Multiple Data*) (BöHME, 2014). SPMD ocorre quando um simples programa está sendo executado em diferentes processadores. A principal diferença entre SIMD e SPMD é que em SPMD cada processador pode estar em diferentes estágios de execução ou fluxos do código. MPMD ocorre quando diferentes programas são executados em múltiplos processadores. Exemplos desse tipo de arquitetura são os atuais processadores *multicore*, que possuem dois ou mais elementos de processamento independentes e podem executar tarefas totalmente diferentes. O assunto *multicore* será tratado com mais detalhes na Seção 2.3.

A taxonomia de Flynn é considerada uma classificação de granularidade grossa, ou seja, deixa de lado detalhes de implementação. Isso faz com que esse sistema de classificação não seja tão preciso. Por exemplo, arquiteturas que utilizam pipelines são difíceis de serem classificadas utilizando esse sistema. Alguns autores defendem a ideia que execução pipeline é um tipo de SISD, já outros defendem que na categoria SISD não existe nenhum tipo de paralelismo. Contudo, neste trabalho, a taxonomia de Flynn foi apresentada apenas para dar uma perspectiva das possibilidades de variações no projeto de arquitetura de computadores, podendo ser desconsiderados tais pontos de discordância. A Figura 1 resume a representação da taxonomia de Flynn.

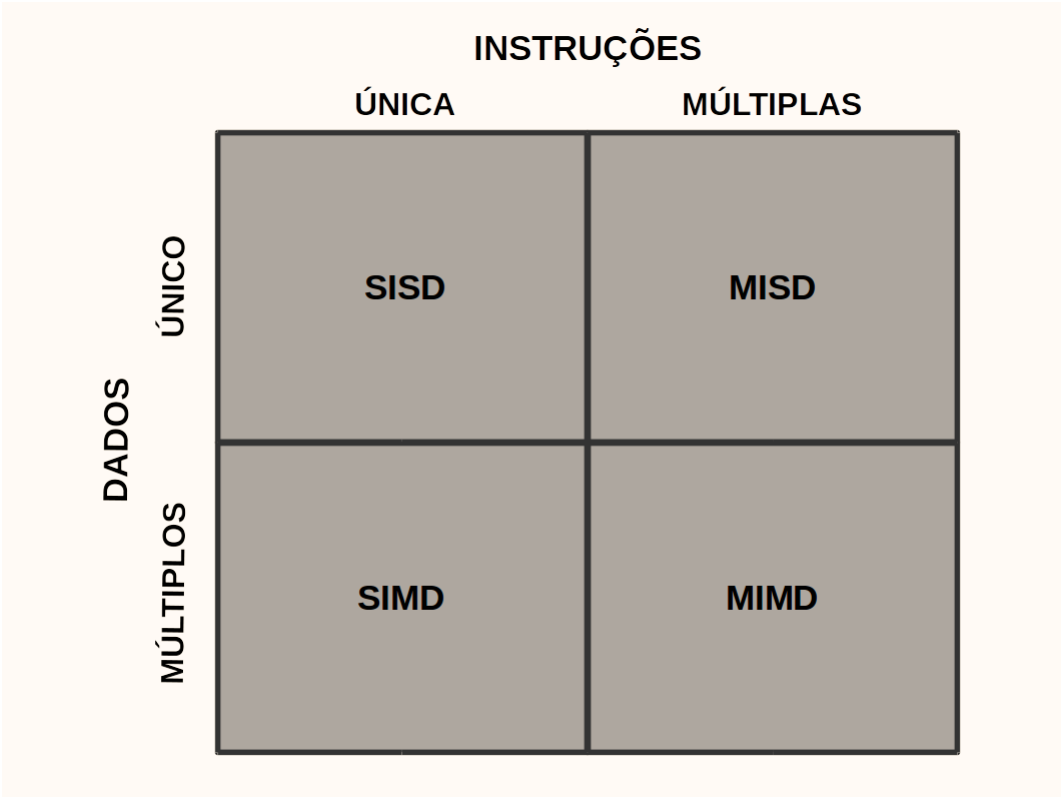

Figura 1 – Taxonomia de Flynn.

Dentre as quatro categorias apresentadas por Flynn, as duas mais populares atualmente são as categorias SIMD e MIMD. Por apresentarem algum nível de paralelismo, essas duas categorias são também conhecidas como **arquiteturas paralelas**. A próxima seção trata esse assunto em maiores detalhes.

2.2 Arquiteturas Paralelas

As arquiteturas paralelas surgiram para suprir a necessidade por computação de alto desempenho demandada, a princípio, por aplicações científicas. Elas são caracterizadas por oferecerem múltiplos elementos de processamento capazes de executar tarefas em paralelo (CULLER; GUPTA; SINGH, 1997). Atualmente, seu uso expandiu-se para

diversos nichos do mercado de computadores, dentre eles, processadores de propósito geral, sistemas embarcados, entre outros. Elas são encontradas na forma de *multicore*, *manycores*, arquiteturas heterogêneas, etc.

Dentre os diversos aspectos que devem ser considerados quanto ao projeto de arquiteturas paralelas, dois se tornam especialmente cruciais em comparação ao projeto de arquiteturas monoprocessadas: os sistemas de interconexão e memória (BLAKE; DRESLINSKI; MUDGE, 2009).

2.2.1 Sistemas de Interconexão

Ter múltiplos núcleos de processamento atuando de forma cooperativa implica a necessidade de disponibilizar um meio pelo qual esses núcleos possam se comunicar. Este meio pode ser utilizado para uma comunicação direta entre os processadores ou para acessar uma memória comum. Há sistemas de interconexão que permitem ou não comunicação paralela. Os barramentos, por exemplo, não permitem comunicação paralela e são recomendados quando o número de elementos de processamento é pequeno. As redes em chip, por outro lado, permitem comunicação em paralelo e suportam uma alta escalabilidade ao custo de um projeto mais complexo.

A Seção 2.8 entrará em maiores detalhes no assunto redes em chip, mais especificamente na rede SoCIN (ZEFERINO; SUSIN, 2003), a rede em chip que é utilizada como meio de interconexão na plataforma proposta neste trabalho.

2.2.2 Sistema de memória

Em processadores monoprocessados, o sistema de memória resume-se à implementação de uma hierarquia, utilizando níveis de cache, a fim de explorar os princípios de localidade temporal e espacial (PRZYBYLSKI, 1990). Em arquiteturas paralelas, outros três fatores precisam ser levados em consideração: a organização da memória, o modelo de consistência e o suporte à coerência de cache. Essas características têm impacto na programabilidade da arquitetura, performance e escalabilidade do número de processadores (BLAKE; DRESLINSKI; MUDGE, 2009).

2.2.2.1 Organização da Memória

A organização da memória pode ser dividida conceitualmente em quatro categorias:

- **Logicamente e fisicamente compartilhado:** compartilham uma memória global, não necessitando assim de explícita troca de mensagem para realizar comunicação;
- **Logicamente compartilhado e fisicamente distribuído:** todos os endereços do espaço de endereçamento são acessíveis de forma compartilhada por todos os

elementos de processamento. Este espaço é, entretanto, implementado em múltiplas memórias físicas distribuídas no sistema de interconexão;

- **Logicamente e fisicamente distribuído:** consistem um determinado número de elementos de processamento (chamados de nós) e uma rede de interconexão que dá suporte à transferência de dados entre estes. Um nó é uma unidade independente, e consiste em um processador e sua própria hierarquia de memória. Por esse motivo, as informações compartilhadas devem ser trocadas entre os processadores por mensagens que são explicitamente enviadas e recebidas pelas tarefas em execução. Para isto, *frameworks* e bibliotecas, tal como MPI (*Message Passing Interface*) (FORUM, 1994), estão disponíveis ao programador;

- **Logicamente distribuído e fisicamente compartilhado:** um único espaço de endereçamento é implementado em uma única memória física, entretanto, segmentos deste espaço são distribuídos entre os elementos de processamento para acesso exclusivo.

2.2.2.2 Modelo de Consistência de Memória

O modelo de consistência define como as operações de acesso à memória podem ser reordenadas em tempo de execução, e pode ser classificado como **forte** ou **fraco** (BLAKE; DRESLINSKI; MUDGE, 2009). Um modelo de consistência é dito fraco quando não existe mecanismo em *hardware* que garanta a ordem no acesso à memória. Portanto, essa abordagem permite que os processadores efetuem operações de leitura e escrita na memória sem a necessidade de comunicar a outro processador. Cabe ao programador estabelecer a correta ordem no acesso à memória para assim garantir a consistência dos dados. Para tal tarefa, as linguagens de programação geralmente disponibilizam primitivas como: *fence e barrier* (BLAKE; DRESLINSKI; MUDGE, 2009). Esses recursos são usados principalmente para controlar o acesso às variáveis de sincronização, porém, os sistemas tornam-se mais difíceis de serem programados.

Em contrapartida, encontram-se os modelos fortes, que impõem restrições na ordem pela qual as operações de escrita e leitura são efetuadas. Por exemplo, a consistência sequencial requer que todos os processadores efetuem os acessos à memória na mesma ordem definida no programa. A programação nesse modelo torna-se mais fácil, porém, a implementação desse mecanismo de consistência apresenta um impacto negativo na performance da arquitetura.

2.2.2.3 Coerência de Cache

O objetivo de uma memória cache é reduzir o tempo que um processador demora para acessar a memória principal de um sistema, que geralmente é lenta, com baixa largura

de banda e extra chip. Em processadores monoprocessados isso já era uma relevante questão. Em processadores *multicore* tornou-se ainda mais, tendo em vista que haverá disputa pelo acesso à memória. Duas principais abordagens são possíveis de serem tomadas ao se implementar uma memória cache: via *hardware* ou via *software* (BLAKE; DRESLINSKI; MUDGE, 2009).

A forma mais tradicional é a cache via *hardware*, ela isenta o programador do gerenciamento da memória, facilitando a programação. Esta é considerada uma abordagem clássica, na qual o processador divide o espaço do chip com uma região de memória que mantém cópias de parte dos dados da memória principal. O gerenciamento é todo feito em *hardware*, utilizando *tags*[1] e algoritmos de substituição para controlar o acesso e manter os dados atualizados. Geralmente são utilizados múltiplos níveis de cache seguindo a máxima de quanto mais próximo o nível for do processador, menores são o tamanho e a latência de acesso. As principais desvantagens dessa abordagem estão em apresentar tempo de acesso não determinístico, usar parte da área em chip para gerenciamento e consumir mais energia.

Na cache via *software* o gerenciamento é realizado por um programa. Nesse caso, a região de memória é chamada de memória de rascunho, e o programador é responsável por gerenciá-la. Suas principais vantagens são os fatos de apresentar um comportamento determinístico em relação ao tempo de acesso à memória, não utilizar área em chip para implementar controle de acesso, algoritmos de substituição, gerenciamento e consumir menos energia. A programação, em contrapartida, torna-se mais complexa. Por esse motivo, a implementação da cache via software é mais interessante para arquiteturas direcionadas a aplicações com restrições de tempo real. Outro aspecto importante a ser ressaltado é o tamanho da região de memória utilizada como cache. Essa decisão depende muito das aplicações que serão executadas na arquitetura em questão. Uma cache maior é apropriada para aplicações com bastante reuso de dados, esse comportamento é geralmente encontrado em aplicações de propósito geral. Já uma cache de tamanho reduzido é indicada para aplicações onde o dado é utilizado poucas vezes.

2.3 Processadores *Multicore*

Multicore são processadores de propósito geral que possuem mais de um núcleo de processamento. A evolução dos processadores monoprocessados para processadores *multicore* se deveu a limitações físicas enfrentadas na fabricação dos processadores. Durante anos, os processadores podiam melhorar de desempenho, geração após geração, apenas aumentando a frequência de operação. Essa prática era possível porque, assim como previu Moore (SCHALLER, 1997), a indústria de semicondutores era (e ainda é) capaz de reduzir

[1] Marcadores de conjuntos de dados presentes na memória principal e na cache.

o tamanho dos transistores pela metade a cada 18 meses, permitindo que esses novos transistores operassem em frequências mais altas. Porém, quanto mais alta a frequência de operação de um transistor, mais energia é consumida e mais potência é dissipada em forma de calor. Há, entretanto, um limite de calor dissipado com o qual a indústria de semicondutores é capaz de lidar (KURODA, 2001). Atingindo este limite, os fabricantes de processadores optaram por projetar arquiteturas dotadas de múltiplos núcleos menos complexos, operando a uma frequência limitada, em vez de processadores mais complexos que operassem em altas frequências (BLAKE; DRESLINSKI; MUDGE, 2009).

Atualmente, pode-se encontrar no mercado uma vasta variedade de processadores *multicore*. Eles são customizados para vários nichos, desde sistemas embarcados a sistemas de propósito geral e servidores. Um exemplo de processador *multicore* utilizado em arquitetura de propósito geral é o Intel Core i7. Esse processador implementa a ISA x86, podendo apresentar de dois a oito núcleos de processamento. Os núcleos utilizam um sistema de interconexão ponto a ponto e podem operar em frequências que variam de 2.66 GHZ a 3.33 GHZ. Cada núcleo implementa avançadas técnicas para extrair o máximo de desempenho de um código sequencial, por exemplo, execução fora de ordem e *pipeline*. O Intel Core i7 está equipado também com unidades de execução SIMD, que tiram vantagem do paralelismo em nível de dados presente em algumas aplicações. O processador é equipado ainda com 3 níveis de cache, sendo 32KB de dados, 32KB de instruções no nível 1, 256KB no nível 2 e 8MB no nível 3. Todas essas características fazem com que esse processador mostre um bom desempenho para computação de propósito geral, porém, todos esses benefícios vêm a um custo de um alto consumo de energia (TOTONI et al., 2012). A Figura 2 mostra uma representação gráfica do Intel Core i7.

Um outro exemplo de processador *multicore* é o ARM Cortex A9. Esse processador é um *softcore* utilizado em dispositivos móveis. *Softcore* é um processador especificado em uma linguagem de descrição de *hardware* (do inglês *Hardware Description Language* - HDL), que pode ser customizado para uma determinada aplicação e sintetizado em ASIC (*Application Specific Integrated Circuits*) ou FPGA (*Field Programmable Gate Array*). O ARM Cortex A9 pode ser configurado para usar de um a quatro núcleos. A memória pode ser configurada para 16, 32 ou 64 KB de instruções e dados de cache nível 1, e até 2 MB de cache nível 2. Os núcleos conectam entre si através de um barramento e apresentam um modelo de consistência fraco de memória. Essas características tornam o desempenho do ARM Cortex A9 incomparável ao Intel Core I7. No entanto, o ARM Cortex A9 consome menos energia, o que é uma virtude em processadores destinados a dispositivos móveis.

Segundo Schreiber (2007), a denominação *multicore* deve ser utilizada para processadores que possuem até 64 núcleos de processamento. A partir dessa quantidade, os processadores já passam a ser chamados de *manycores*. Um atual exemplo de processador *manycore* são as GPU's. Na próxima seção o assunto *manycore* será abordado em maiores

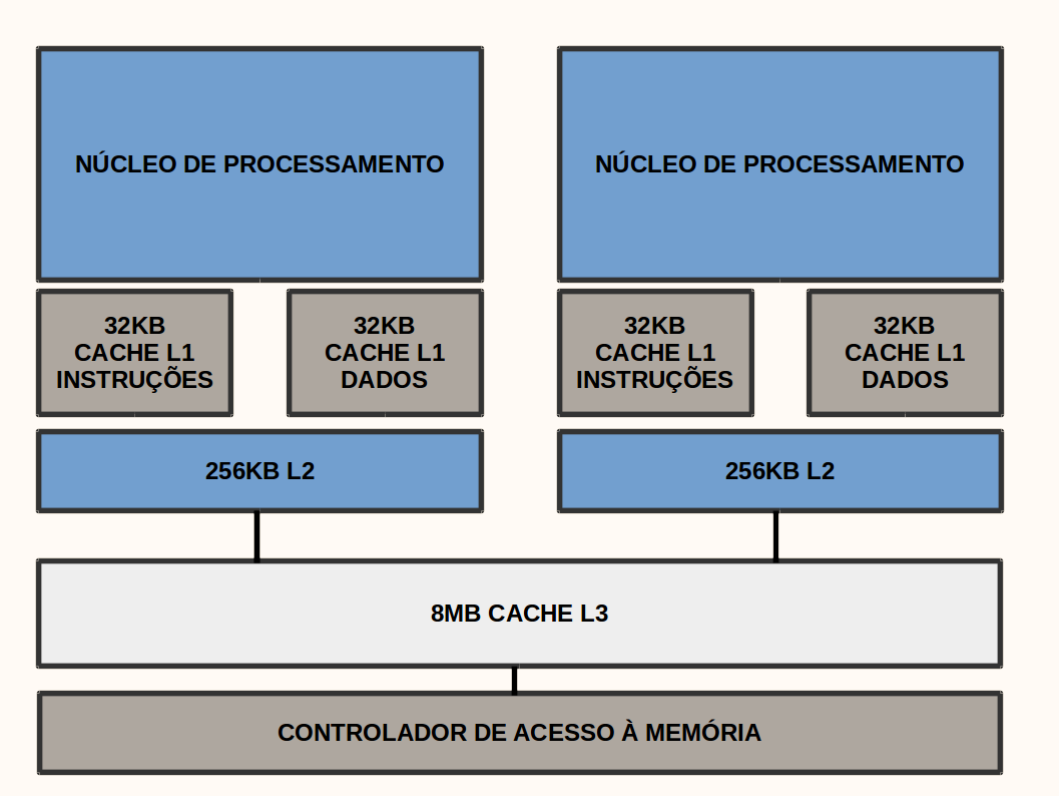

Figura 2 – Representação gráfica do Intel Core i7.

detalhes.

2.4 GPU's

GPU's representam uma evolução das antigas placas de vídeo que eram encarregadas de gerenciar a exibição de imagens e textos nos monitores dos computadores. Com o passar do tempo, essas placas foram evoluindo e passaram a ser responsáveis não só por exibir imagens no monitor, mas também a realizar alguns cálculos de processamento de imagem.

Genericamente elas são constituídas por um conjunto de *Stream Multiprocessors* (SM). Cada SM é essencialmente um processador multicore composto por um determinado número de *Stream Processors* (SP). A execução em uma GPU é altamente dirigida a *threads*. Por esse motivo, toda sua arquitetura é pensada para maximizar esse tipo de execução.

As *threads* em uma aplicação GPU podem ser agrupadas em conjuntos chamados *blocks*. A GPU executará todas as *threads* de um *block* em um único SM (embora múltiplos blocos possam executar em um mesmo SM). As *threads* de um *block* podem ainda serem divididas em *warps*, 32 *threads* por *warp* (KIRK; HWU, 2010).

Um ponto importante a ser salientado nesse modelo de execução é que todas as *threads* em um *warp* executam em modo *locstep*. Isso significa que, durante o ciclo de

busca, a mesma instrução será buscada para todas as *threads* em um *warp*. Posteriormente, no ciclo de execução, cada *thread* irá executar aquela instrução em particular ou, em caso de instruções de salto condicional, onde o salto não é tomado, permanecer em um estado de espera.

Cada SM executa suas *threads* em regime de **compartilhamento de tempo**, assim como nos sistemas operacionais tradicionais. Caso uma *thread* necessite realizar um acesso demorado à memória, o SM irá então escalonar algum outro *warp* enquanto o acesso estiver pendente.

O suporte que o *hardware* das GPU's oferece para execução de *threads* inclui também um conjunto de registradores para cada *warp*. Assim a troca de contexto torna-se menos penosa para a arquitetura, diferente do que ocorre em CPUs.

A estrutura de memória básica de uma GPU geralmente é divida em três partes:

- **Memória global**: constitui uma região de memória compartilhada por todas as *threads* da aplicação. É geralmente bem maior que a memória compartilhada e acessível pela CPU *host*. É extra-chip e lenta, consumindo aproximadamente centenas de ciclos por acesso.
- **Memória compartilhada**: representa uma região de memória compartilhada por todas as *threads* em um SM. É justamente o meio utilizado para elas se comunicarem. Pelo fato de esta memória ser intrachip, seu acesso é muito rápido. Seu tamanho é pequeno, geralmente 16K bytes para cada SM.
- **Memória privada:** é uma região de memória particular a cada *thread*, sua implementação varia entre os fabricantes de GPU's, sendo geralmente um *array* de registradores.

A Nvidia G200 (BLAKE; DRESLINSKI; MUDGE, 2009) é um exemplo de uma GPU atual. Essa arquitetura contém 240 núcleos, chamados de *Stream Processors* (SP), agrupados em 10 *clusters*. Grupos de oito SPs compartilham uma memória local de 16 KB. Os *clusters* são controlados no estilo SIMD, com a diferença de que cada SP é capaz de executar saltos condicionais. Para realizar essa tarefa, todos os SPs executam os dois caminhos possíveis e, ao final da execução, validam somente o caminho que deveria ser executado de fato. Outra diferença para uma arquitetura SIMD tradicional é o acesso à memória. Os núcleos das tradicionais arquiteturas SIMD fazem acesso a endereços sequências na memória (BLAKE; DRESLINSKI; MUDGE, 2009), enquanto que os SPs das GPU's podem acessar endereços independentes. Por exemplo, se o núcleo 1 acessar o endereço x, o núcleo 2 não obrigatoriamente acessaria o endereço x+1, mas qualquer outro endereço. Essa característica torna as GPU's mais parecidas com as arquiteturas MIMD, permitindo que possam ser usadas em aplicações de propósito geral.

A organização do sistema de memória da G200 é destinada a explorar o paralelismo em nível de dados. Dessa forma, ela não apresenta mecanismos de coerência e usa uma pequena memória de rascunho em vez de uma cache em *hardware*. A memória é pequena para que haja mais espaço em chip dedicado a recursos computacionais. Todas essas características tornam a arquitetura da G200 bem adaptada a aplicações que possuem um alto nível de paralelismo em nível de dados. Exemplo de aplicações são: processamento de imagens, análise de dados financeiros, aplicações científicas, etc. Por outro lado, esse mesmo desempenho não é encontrado em aplicações de propósito geral, pelo fato destas aplicações apresentarem elevada taxa de saltos condicionais e acesso aleatório a endereços de memória. A Figura 3 mostra a representação gráfica da arquitetura do Nvidia G200.

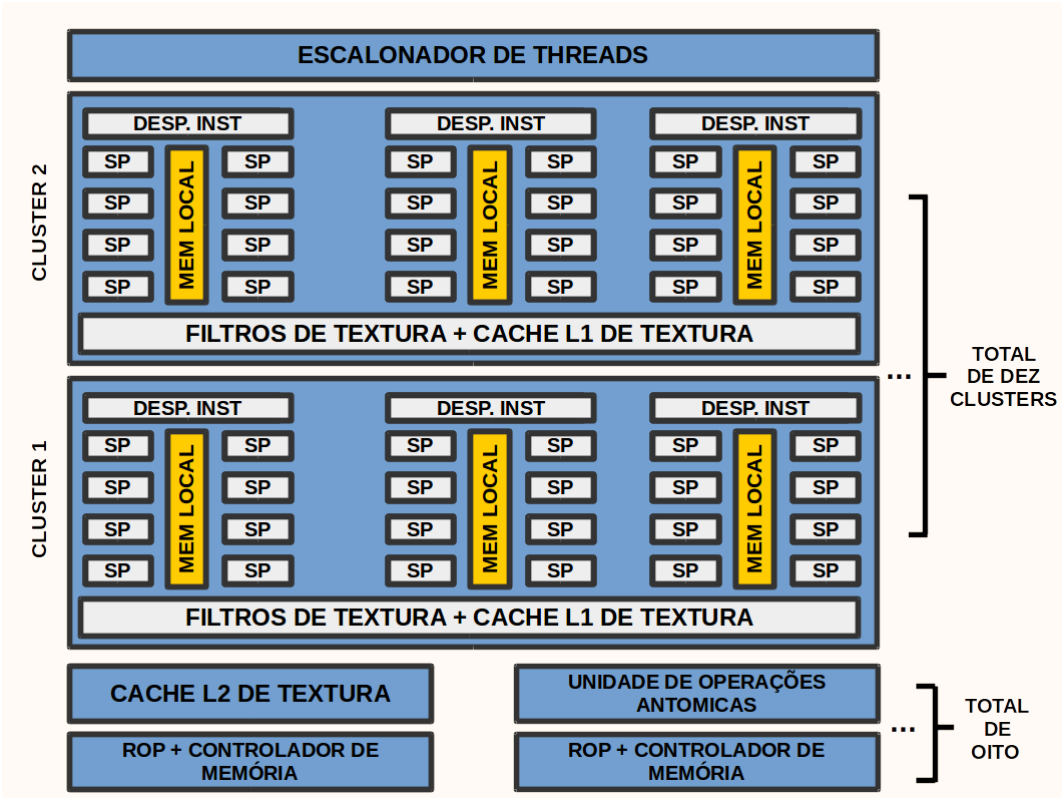

Figura 3 – Representação gráfica da GPU Nvidia G200.

Como dito anteriormente, as GPU's podem ser usadas em conjunto com CPU's a fim de criar um sistema com melhor desempenho. Esses sistemas que empregam mais de uma arquitetura são denominados **arquiteturas heterogêneas**. A seção seguinte abordará esse tema em maiores detalhes.

2.5 Arquiteturas Heterogêneas

O grande investimento feito nas GPU's, pela rica indústria de vídeogame, e sua produção em grande escala, fizeram com que estas se tornassem cada vez mais potentes e

baratas. A combinação desses dois fatores, preço e desempenho, despertou o interesse pelo uso das GPU's em aplicações não gráficas.

A princípio, essas aplicações não gráficas eram adaptadas ao contexto das GPU's, utilizando primitivas gráficas de tal forma que o problema pudesse ser resolvido utilizando alguma linguagem de processamento de imagem, por exemplo, OpenGL (WOO et al., 1999) ou DirectX (EIMANDAR, 2013). Posteriormente, surgiram linguagens como RapidMiner (HOFMANN; KLINKENBERG, 2013), Brook (BUCK et al., 2004) e CUDA (COOK, 2013) criadas especialmente para facilitar a programação das GPU's para propósito geral, assim surgia o termo GPGPU (MACEDONIA, 2003).

Com o passar do tempo, além de incorporarem GPU's e CPUs, alguns sistemas computacionais passaram a integrar outros aceleradores, por exemplo, FPGAs, DSPs, ASICs. Esses sistemas computacionais que incorporam elementos de processamento com diferentes arquiteturas denominam-se **Arquiteturas Heterogêneas**.

Um exemplo de arquitetura heterogênea é o processador CBEA (CHEN et al., 2007). Ele consiste em uma CPU tradicional e oito núcleos aceleradores SIMD. Trata-se de uma arquitetura muito flexível, onde cada núcleo SIMD pode executar programas independentes no estilo MPMD e se comunicam através de um barramento em chip. A arquitetura tem como objetivo maximar performance enquanto consome o mínimo de energia. A Figura 4 ilustra a implementação de uma variação do processador CBEA, o PowerXCell 8i. Este consiste basicamente em um núcleo de processamento com múltiplas linhas de execução, chamado de *Power Processing Element* (PPE), oito processadores RISC com arquiteturas SIMD, chamados de *Synergistic Processing Elements* (SPEs) e um barramento de comunicação, chamado de *Element Interconnect Bus* (EIB). Ele ainda conta com uma interface controladora de memória, do inglês *Memory Interface Controller* (MIC) e uma interface controladora de barramento, do inglês *Bus Interface Controller* (BIC). O MIC gerencia o acesso à memória principal e o BIC gerencia o acesso aos barramentos externos.

As arquiteturas heterogêneas oferecem recursos com características distintas que atendem a uma maior gama de aplicações. Para explorar esses recursos é necessária a utilização de linguagens de programação apropriadas, por exemplo, OpenCL. A próxima seção apresenta as principais características inerentes às linguagens de programação paralelas de um modo geral para que na Seção 2.7 a linguagem OpenCL seja explorada.

2.6 Linguagens de Programação Paralela

O modelo de memória adotado em um sistema de computação paralelo exerce um grande impacto na forma como o sistema será programado. De fato, uma forma de classificar as linguagens de programação paralela é justamente baseada na organização da

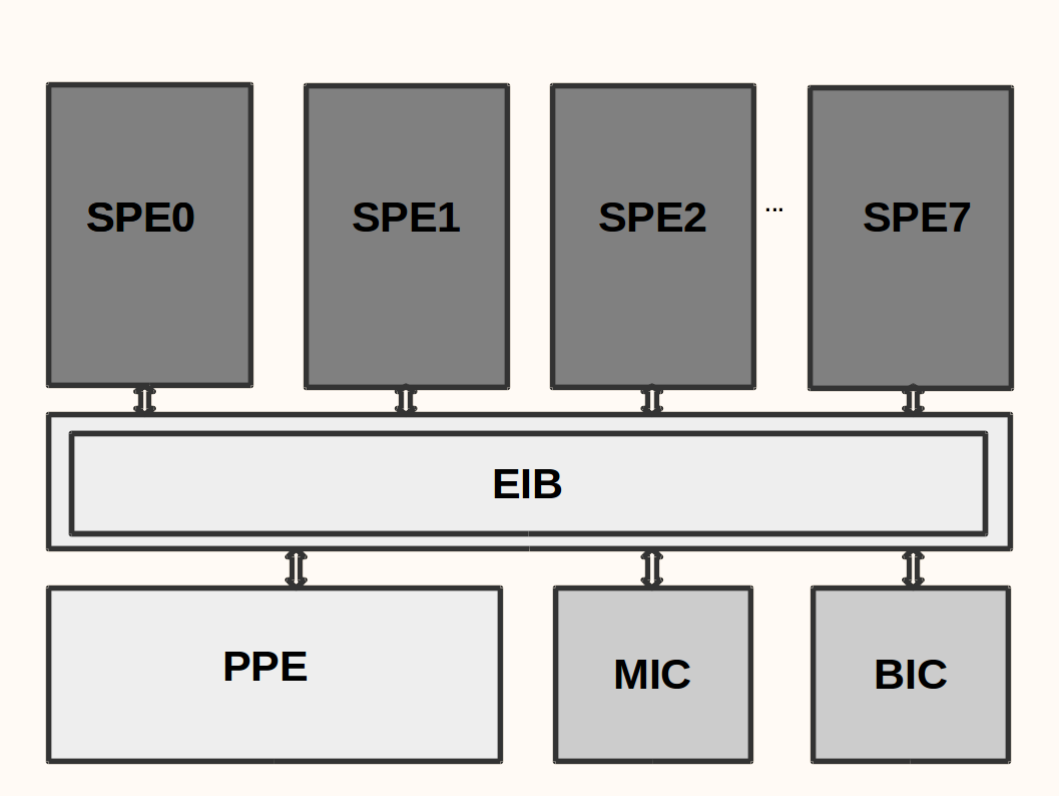

Figura 4 – PowerXCell 8i.

memória do sistema. Como já discutido, a memória pode ser compartilhada ou distribuída. Estas duas organizações de memória geram dois modelos de programação, ditos modelos paralelos puros (DIAZ; MUñOZ-CARO; NIñO, 2012), que também são base para outros modelos.

2.6.1 Programação em Memória Compartilhada

Neste modelo, tarefas paralelas compartilham um espaço de endereçamento global, onde os processos podem ler e escrever assincronamente. Essa característica torna necessário o uso de mecanismos de controle de acesso à memória compartilhada para prevenir condições de corrida e *deadlocks*, exemplos de mecanismos são *locks* e semáforos. Pode ser considerado o modelo mais simples, sua principal vantagem é a não necessidade de troca explícita de mensagem entre as tarefas (uma vez que a comunicação é feita através da memória). Desta forma, o desenvolvimento de aplicações torna-se mais simples.

Um exemplo de *framework* de programação que utiliza memória compartilhada é o OpenMP (DAGUM; MENON, 1998). A sigla deriva de *Open Multi-Processing* e consiste em uma API que dá suporte à programação de plataformas multiprocessadas com memória compartilhada. O *framework*, baseado em C, C++ e Fortran, consiste em um conjunto de diretivas de compilador, rotinas de bibliotecas e variáveis de ambiente. A parte do código que deve executar em paralelo é marcada utilizando uma diretiva de compilador que fará

com que sejam criadas *threads* para execução em paralelo. OpenMP é um modelo portátil e escalável, que fornece ao programador uma interface simples e flexível para desenvolver aplicações paralelas.

Outra biblioteca de programação que utiliza memória compartilhada é o POSIX *thread* (BUTENHOF, 1997), também conhecido como *Pthread*. Ela consiste em um padrão POSIX para criar e manipular *threads*, sua programação é feita utilizando uma API disponível em quase todos os sistemas operacionais baseados em linux, assim como no Microsoft Windows.

2.6.2 Programação em Memória Distribuída

Neste modelo, o programa é dividido em um conjunto de tarefas que utilizam sua própria memória local durante a execução. As tarefas comunicam-se através de mensagens enviadas e recebidas utilizando instruções de *sending e receive*, respectivamente. Do ponto de vista do programador, o uso dessas instruções é efetuado utilizando sub-rotinas de bibliotecas embutidas no código fonte, ou seja, o programador é responsável por determinar todo o paralelismo.

Uma biblioteca tradicional para programação em sistemas com memória distribuída é a MPI *(Message Passing Interface Standard)* (SNIR et al., 1998). MPI consiste em uma especificação de biblioteca padrão para envio de mensagem, desenvolvido e mantido por um consórcio de colaboradores acadêmicos, pesquisadores e indústria. O principal objetivo da biblioteca MPI é prover um padrão de programação que ofereça praticidade, portabilidade, eficiência e flexibilidade.

Atualmente existe uma variedade de linguagens de programação paralela para os mais diversos fins e aplicações. Dentre todas, OpenCL (Khronos Group, 2010) foi escolhida para ser a linguagem SIMD para qual a plataforma proposta neste trabalho de dissertação dá suporte. A próxima seção sumariza as principais características da linguagem, além de mostrar exemplo de como utilizá-la.

2.7 OpenCL

OpenCL é um framework de programação que tem como objetivo garantir a portabilidade de código através de diferentes plataformas heterogêneas (MUNSHI et al., 2011). Foi inicialmente proposto pela Apple e depois desenvolvido pelo grupo Khronos. Além da linguagem, que é um subconjunto da linguagem C, o programador OpenCL conta com um ambiente de execução, uma API e blibliotecas.

A fim de facilitar o entendimento do padrão de programação, as principais ideias inerentes ao OpenCL serão descritas utilizando uma hierarquia de modelos: modelo de

plataforma, modelo de execução e modelo de memória.

2.7.1 Modelo de Plataforma

A plataforma OpenCL é uma abstração que representa o sistema no qual o programa será executado, ou seja, é a forma que OpenCL enxerga o *hardware*.

A plataforma é composta por dois tipos de elementos: um processador *host*, e um ou mais dispositivos. O processador *host* é responsável por gerenciar a execução do programa. Essa gerência inclui preparar e trocar dados com os dispositivos, usando a API OpenCL, além de enviar funções para serem executadas em paralelo. Geralmente é um processador de propósito geral, otimizado para executar código sequencial. Por outro lado, os dispositivos são responsáveis por executar a parte paralela do código. Essa parte do código é descrita em funções especiais do OpenCL chamadas *kernel*. Cada dispositivo é dividido em Unidades de Computação (UC), e cada UC é subdividida em Elementos de Processamento (EP). Em sistemas reais, os dispositivos geralmente usados são GPU's, CPUs, FPGAs, etc. A Figura 5 mostra uma representação gráfica do modelo de plataforma OpenCL.

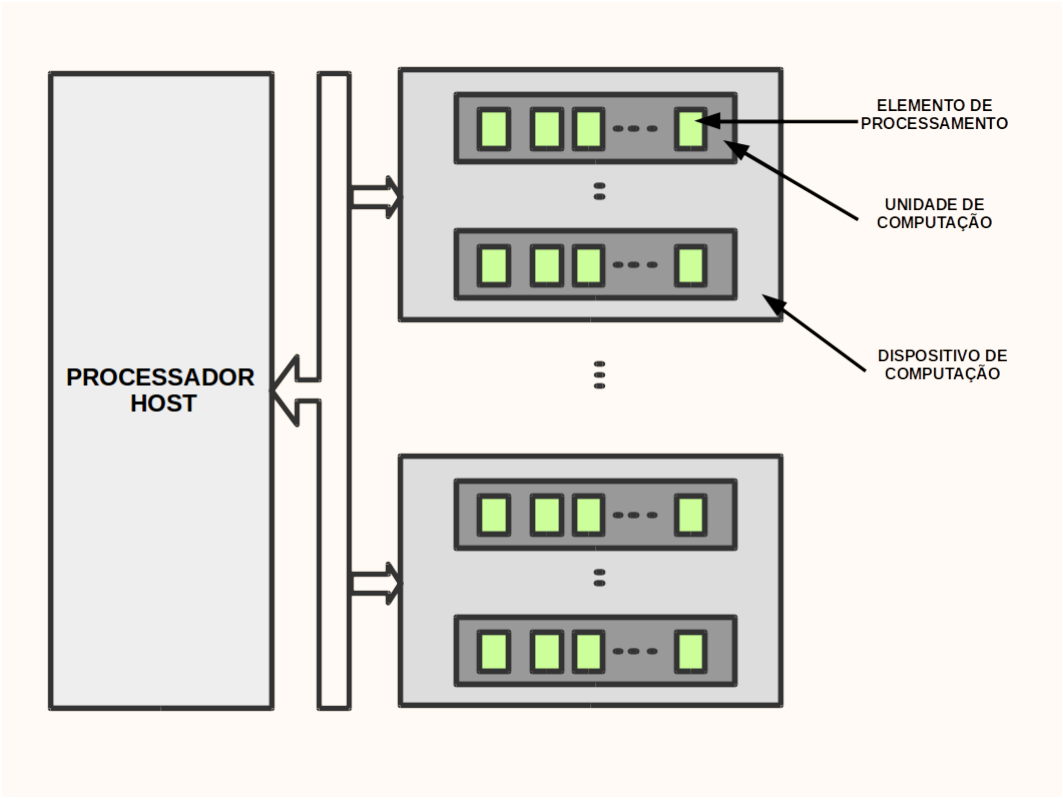

Figura 5 – Modelo de plataforma OpenCL.

2.7.2 Modelo de Execução

O modelo de execução representa a dinâmica de uma aplicação OpenCL. Ele reflete o modelo de plataforma, por esse motivo, também é dividido em duas partes: programa *host* e funções *kernel*. O programa *host* usa a API OpenCL para criar e gerenciar uma estrutura de dados chamada *context*. Essa estrutura define o ambiente no qual as funções *kernel* serão executadas. As funções *kernel* são as partes paralelizáveis da aplicação. Um *kernel* em execução é chamado de *work-item*, os *work-items* de uma aplicação podem ser agrupados em conjuntos chamados de *work-groups*.

OpenCL provê um identificador para cada *work-item* e *work-group*, sendo este identificador composto por uma tupla de N dimensões, com N variando de um a três. Essa identificação representa a localização do *work-item*, ou *work-group*, em um conceito chamado *grid* de execução. A *grid* de execução é uma representação visual da disposição dos *work-items* de uma aplicação, como mostrado na Figura 6.

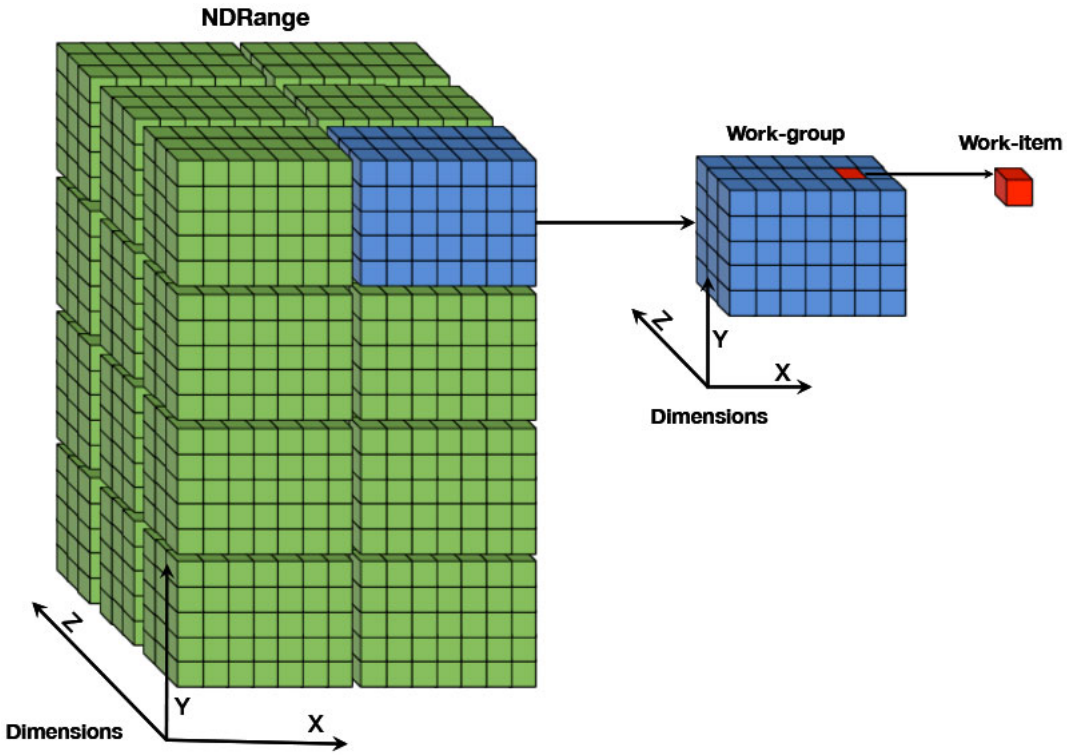

Figura 6 – NDRange de uma aplicação OpenCL.

Fonte: http://rtcmagazine.com/articles/view/103610

Depois da estrutura *context* ser criada, o elemento *host* pode interagir com um dispositivo submetendo comandos através de uma estrutura fila chamada *command-queue*. Os comandos são classificados em três categorias: comandos de *kernel*, memória e sincronização. Os comandos de *kernel* estão relacionados à configuração e envio dos *work-items* a serem executados. Os comandos de memória referem-se às trocas de dados entre o *host* e os dispositivos. Já os comandos de sincronização dizem respeito à ordem de

execução dos outros dois comandos. Figura 7 mostra uma representação gráfica do modelo de execução OpenCL.

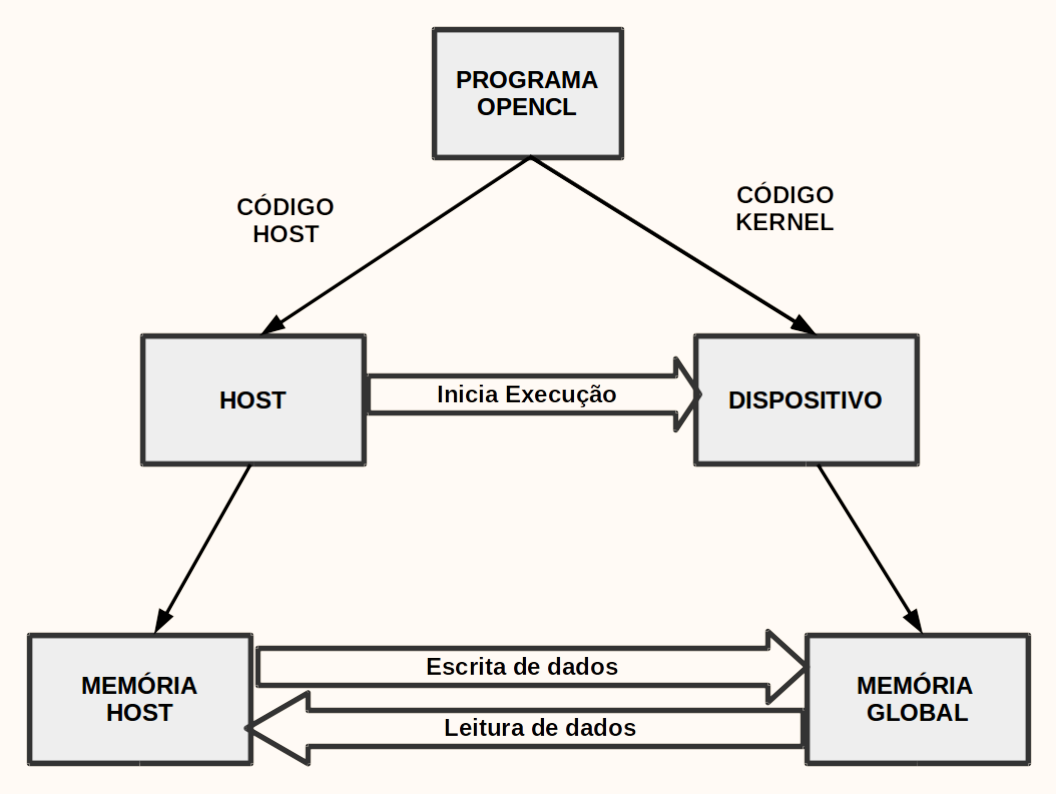

Figura 7 – Modelo de execução OpenCL.

2.7.3 Modelo de Memória

Ainda em sintonia com o modelo de plataforma, o OpenCL divide sua memória em duas partes principais, uma para cada um dos dois principais elementos da plataforma: memória do *host* e memória do dispositivo. A memória do *host* pode ser considerada a memória principal do sistema. É nela onde são armazenados os dados iniciais das aplicações. Em um sistema real, geralmente é a memória DRAM de uma CPU.

A memória do dispositivo é aquela utilizada pelos *work-items*. Ela é subdividida em quatro partes:

- **Global:** é uma região de memória acessível para leitura e escrita tanto pelo *host* quanto pelos dispositivos. Esta região da memória é utilizada pelo *host* para trocar informações com o dispositivo;
- **Constante:** é uma região constante memória global, ou seja, acessada pelos *work-items* somente para leitura;
- **Local:** região de memória compartilhada por todos os *work-items* de um determinado *work-group*;

- **Privada:** região de memória privada a cada *work-item*.

A Figura 8 mostra uma representação gráfica do modelo de memória OpenCL.

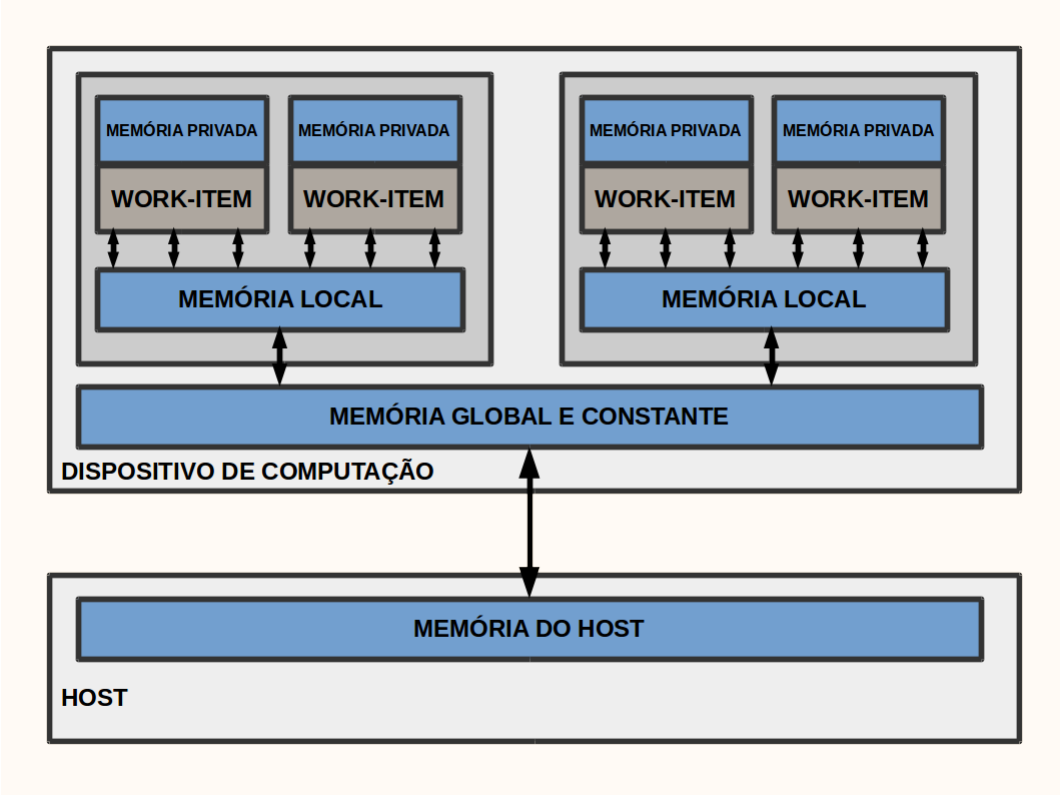

Figura 8 – Modelo de memória OpenCL.

2.7.4 Exemplo de Programa OpenCL

Nesta seção será explicado em detalhes um simples exemplo de programa OpenCL, a soma de vetores. A programação será dividida em etapas que quase todos os programas OpenCL devem seguir, como mostrado em Alvarez e Yamagiwa (2011):

1. Inicialização: a primeira etapa consiste em encontrar um dispositivo OpenCL disponível no sistema e criar as estruturas necessárias (*contexto, command-queue*). Como mostrado na Figura 9, essa operação pode ser realizada utilizando as seguintes funções:

   - *clGetPlatformIDs*: essa função retorna, por meio de seu segundo argumento, uma lista com as plataformas OpenCL disponíveis no sistema (NVIDIA, ATI, Intel, etc.).
   - *clGetDevicesIDs*: essa função retorna, através de seu quarto argumento, uma lista com os dispositivos OpenCL disponíveis para a plataforma passada como primeiro argumento.

```
// ## PASSO (1) ## //

// Obter a plataforma OpenCL
CiErr1  =  clGetPlatformIDs(1, &cpPlatform, NULL);

// Obter os dispositvos
CiErr1  =  clGetDeviceIDs(cpPlatform, CL_DEVICE_TYPE_GPU, 1, &cdDevice, NULL);

// Criar o contexto
CxGPUContext = clCreateContext(0, 1, &cdDevice, NULL, NULL, &ciErr1);

// Criar a fila de comandos
CqCommandQueue = clCreateCommandQueue(cxGPUContext, cdDevice, 0, &ciErr1);
```

Figura 9 – Primeiro passo de uma programação OpenCL.

- *clCreateContex*: essa função cria um novo contexto para o dispositivo passado como argumento. A estrutura contexto armazena todos os recursos OpenCL usados pelo dispositivo (*buffers, command-queue, kernel, program*, etc.)
- *clCreateCommandQueue*:essa função cria uma fila de comandos para um determinado contexto de um dado dispositivo. Essa estrutura é utilizada para enviar comandos (execução de *kernel*, transferência de dados, etc.) do *host* para o dispositivo.

2. Compilação do programa *kernel*: nessa etapa é criado e compilado o programa *kernel* (função executada no dispositivo por todos os *work-items*). O sistema de tempo de execução do OpenCL pode ler o código do programa *kernel* tanto de um arquivo com extensão .cl ou simplesmente de uma *string* no programa *host*. Como mostrado na Figura 10, essa operação utiliza as seguintes funções:

```
// ## PASSO (2) ## //

// Criar o programa
cpProgram  =  clCreateProgramWithSource(cxGPUContext, 1, cdDevice,
                                        (const char**) &cSourceCL,
                                        &szKernelLength, &ciErr1);

// Construir o programa
ciErr1 = clBuildProgram(cpProgram, 0, NULL, NULL, NULL, NULL);

// Criar o kernel
CkKernel = clCreateKernel(cpProgram, "VectorAdd", &ciErr1);
```

Figura 10 – Segundo passo de uma programação OpenCL.

- *clCreateProgramWithSource*: essa função cria um objeto *program* para um dado contexto, carregando o código fonte armazenado na *string cSourceCL* no programa objeto.
- *clBuildProgam*: essa função compila e liga o programa objeto criado.
- *clCreateKernel*: essa função cria o objeto *kernel* a partir do objeto *program* criado

3. Criação e inicialização dos *Buffers*: é necessário criar, inicializar e enviar dados do *host* para o dispositivo. Do lado do *host* os dados são criados pela forma tradicional (utilizando a função *malloc*, por exemplo), já nos dispositivos, como mostrado na Figura 11, são usadas as seguintes funções:

```
// ## PASSO (3) ## //
// Alocar os buffers na memória global do dispositvo
cmDevSrcA = clCreateBuffer(cxGPUContext, CL_MEM_READ_ONLY,
                           sizeof(cl_float)*szGlobalWorkSize,
                           NULL, &ciErr1);

cmDevSrcB = clCreateBuffer(cxGPUContext, CL_MEM_READ_ONLY,
                           sizeof(cl_float)*szGlobalWorkSize,
                           NULL, &ciErr2);

cmDevDst = clCreateBuffer(cxGPUContext, CL_MEM_WRITE_ONLY,
                           sizeof(cl_float)*szGlobalWorkSize,
                           NULL, &ciErr3);

// Escrita dos dados na GPU
ciErr1 = clEnqueueWriteBuffer(cqCommandQueue, cmDevSrcA,
                              CL_FALSE, 0,
                              sizeof(cl_float)*szGlobalWorkSize,
                              srcA, 0, NULL, NULL);

ciErr1 |= clEnqueueWriteBuffer(cqCommandQueue, cmDevSrcB,
                              CL_FALSE, 0,
                              sizeof(cl_float)*szGlobalWorkSize,
                              srcB, 0, NULL, NULL);
```

Figura 11 – Terceiro passo de uma programação OpenCL.

- *clCreateBuffer*: cria um objeto *buffer* na memória global do dispositivo.
- *clEnqueueWriteBuffer*: essa função enfileira uma transferência de dados do lado do *host* para um *buffer* no lado do dispositivo.

4. Execução do *kernel*: nesta etapa são configurados os argumentos da função *kernel*, que então pode ser enviado para execução. As funções usadas para essa operação, como mostrado na Figura 12, são:

```
// ## PASSO (4) ## //
// Configura os valores dos argumentos
ciErr1 = clSetKernelArg(ckKernel, 0, sizeof(cl_mem),
                        (void*)&cmDevSrcA);
ciErr1 |= clSetKernelArg(ckKernel, 1, sizeof(cl_mem),
                        (void*)&cmDevSrcB);
ciErr1 |= clSetKernelArg(ckKernel, 2, sizeof(cl_mem),
                        (void*)&cmDevDst);
ciErr1 |= clSetKernelArg(ckKernel, 3, sizeof(cl_int),
                        (void*)&iNumElements);

// Dispacha kernel
ciErr1 = clEnqueueNDRangeKernel(cqCommandQueue, ckKernel,
                                1, NULL, &szGlobalWorkSize,
                                &szLocalWorkSize, 0, NULL, NULL);
```

Figura 12 – Quarto passo de uma programação OpenCL.

- *clSetKernelArg*: essa função configura os valores dos argumentos do *kernel* para o dado apontado pelo último argumento dessa função.
- *clEnqueueNDRangeKernel*: essa função enfileira um comando para execução do *kernel* em um dado dispositivo. Nessa função ainda é especificado e dimensionado o tamanho do *NDRange*.

5. Leitura do resultado: esta é a última etapa da execução, transferir o resultado da memória do dispositivo para o a memória do *host*. Essa operação é feita, como mostrado na Figura 13, utilizando a seguinte função:

```
// ## PASSO (5) ## //
// Read of results
ciErr1 = clEnqueueReadBuffer(cqCommandQueue, cmDevDst,
                             CL_TRUE, 0,
                             sizeof(cl_float)*szGlobalWorkSize,
                             dst, 0, NULL, NULL);
```

Figura 13 – Quinto passo de uma programação OpenCL.

- *clEnqueueReadBuffer*: essa função enfileira um comando de transferência de dados do dispositivo para o *host*.

Todas as funções anteriores são usadas no programa *host* da aplicação OpenCL. A função *kernel*, ilustrada na Figura 14, deve começar com a diretiva *__kernel*. Cada argumento declarado na função é ligado a um *buffer* alocado na memória do dispositivo. Para cada um dos argumentos deve ser especificada uma região de memória onde o *buffer* será alocado. No exemplo, os *arrays* *a* e *b* são armazenados em uma região constante da memória utilizando as diretivas *__global const*. O array *c* será alocado na região global. Para uma variável escalar, como *iNumElements*, não é necessário especificar a região de memória, porque será uma variável privada para um work-item em execução.

A função *kernel* executa a soma de elementos dos *arrays* *a* e *b* e armazena em *c*. Antes de executar a soma, a função *get_global_id* é utilizada para obter o identificador do *work-item* dentro do *NDRange*. Devido o número de *work-items* poder ser maior que o tamanho do *array*, é utilizada uma estrutura de verificação para validar a posição do array. A função *barrier* é utilizada para sincronizar a execução dentro de um *work-group*. Os *work-items* só armazenarão o resultado na variável c, depois que todos os *work-items* do *work-group* alcançarem a função *barrier*.

2.8 Redes em Chip — A Rede SoCIN

Uma rede em chip, do inglês *network on chip* (NoC), é uma estrutura de interconexão de componentes (memórias, microprocessadores, etc) inspirada na rede de computado-

```
__kernel void sum(__global const int* a,
                  __global const int* b,
                  __global int*c,
                  Int iNumElements)
{

    int idi = get_global_id(0);
    int idg = get_group_id(0);
    int size = get_local_size();
    Int x;
    if (idg*size + idi >= iNumElements)
        return;

    x = a[idg*size + idi] + b[idg*size + idi];

    barrier();

    c[idg*size + idi] =   x;
};
```

Figura 14 – Kernel OpenCL.

res tradicional (JANTSCH; TENHUNEN, 2003). Ela é composta por um conjunto de roteadores e canais de conexões ponto a ponto. Cada roteador possui cinco canais de comunicação, quatro para conectar aos roteadores vizinhos (norte, sul, leste e oeste) e um para conectar o componente (local). Uma rede em chip é um meio de conexão baseado no modelo de comunicação *troca de mensagem*. Os componentes, conectados a cada roteador, comunicam-se enviando e recebendo mensagens. Cada mensagem é composta por um cabeçalho, uma carga útil e um terminador. O cabeçalho carrega informações necessárias para a mensagem trafegar pela rede, a carga útil é a informação que se deseja enviar e o terminador, como sugere o nome, informa o fim da mensagem. Para que um elemento de processamento envie uma mensagem pela rede, a mensagem precisa ser encapsulada. Esse encapsulamento é feito por um componente chamado interface de rede, do inglês *network interface* (NI).

Uma rede em chip pode ser descrita por sua topologia e pelas estratégias utilizadas para: rotear mensagens da origem ao destino; controlar o fluxo de dados entre roteadores adjacentes; encaminhar dados de uma entrada do roteador para uma de suas saídas; arbitrar o uso de um canal de saída, quando disputado por mais de uma mensagem; e armazenar dados temporariamente, quando estes não podem seguir adiante em suas rotas.

A plataforma proposta neste trabalho de dissertação utiliza uma rede em chip chamada SoCIN (ZEFERINO; SUSIN, 2003) para interconectar seus elementos de proces-

samento. Na próxima seção a rede SoCIN será apresentada em maiores detalhes.

2.8.1 SoCIN

A rede SoCIN, do ingles *System on Chip Interconnection Network*, é uma rede em chip baseada em uma arquitetura de roteador parametrizável pensada para projetos de redes de baixo custo. A seguir são descritas as seis principais características da rede SoCIN:

- Topologia: A rede SoCIN pode ser construída usando uma topologia *2-D grid* ou *torus*. A primeira opção oferece baixo custo, enquanto a segunda reduz a latência média no envio de mensagens.

- Roteamento: O esquema de roteamento utilizado é o algoritmo XY. Nessa abordagem, um pacote deve primeiro viajar na direção X, depois, quando chegar na coluna do destinatário, seguir para o alvo, agora na direção Y.

- Controle de fluxo: O controle de fluxo na rede é baseado na estratégia *handshake*, por ser simples e barata. Quando um emissor coloca um dado em um link, ele ativa um sinal de validação (*val*). Quando o receptor está pronto para consumir o dado, ele ativa um sinal de reconhecimento (*ack*).

- Encaminhamento: A abordagem utilizada para encaminhamento é conhecida como buraco de minhoca, do inglês *wormhole*. As mensagens são enviadas através de pacotes, cada pacote é composto por *flits*. Um *flit* (*flow control unit*) é a menor unidade sobre a qual o controle de fluxo age. Na rede SoCIN um flit tem mesma largura física do canal. Em um encaminhamento *wormhole*, quando um *flit* de cabeçalho de um pacote chega a um roteador, ele é encaminhado para o canal de saída e, assim que esse estiver disponível, o pacote é escalonado. Em seguida, a carga útil segue o cabeçalho pelo mesmo caminho.

- Arbitragem: O algoritmo *round-robin* é utilizado para arbitrar o acesso a um canal de saída de um roteador. Essa abordagem não é a mais barata, porém é a mais rápida, e garante justiça na disponibilidade dos canais e impossibilita a existência de *starvation* (nenhum pacote permanece indefinidamente esperando por um canal de saída).

- Armazenamento: É feito utilizando uma memória do tipo FIFO em cada canal de entrada dos roteadores. O tamanho da FIFO é ajustável de acordo com as necessidades do projeto da plataforma.

2.9 Considerações finais

Nesse capítulo de fundamentação teórica, foram apresentados os principais conceitos necessários para o entendimento do restante da dissertação. Primeiramente, foi definido arquitetura de computadores como sinônimo para ISA, organização e *hardware* de um projeto de um processador. Em seguida foi apresentada a Taxonomia de Flynn, que classifica as arquiteturas de computadores, de acordo com seu nível de paralelismo, em quatro categorias: SISD, SIMD, MISD e MIMD. Dentre essas arquiteturas, a próxima seção apresentou as *arquiteturas paralelas*, dando ênfase nos sistemas de interconexão e memória. Logo em seguida foram apresentados os conceitos e exemplos reais de *multicores, manycores e arquiteturas heterogêneas*. A seção seguinte tratou das linguagens de programação paralela, classificando-as em memória compartilhada e distribuída. Em seguida foi apresentada a linguagem OpenCL e seus três modelos hierárquicos: modelo de plataforma, execução e memória. Um exemplo de aplicação OpenCL também foi apresentado. Por fim, foi feita um breve explanação sobre redes em chip, utilizando como caso de estudo a rede SoCIN.

O próximo capítulo fará uma explanação sobre os principais trabalhos disponíveis na literatura sobre suporte em *hardware* para execução SIMD e estudos sobre OpenCL.

3 Estado da Arte

Este capítulo de estado da arte está divido em duas seções. A primeira mostra alguns trabalhos que investigam recursos para execução SIMD em processadores de propósito geral. Já na seção final são mostradas algumas publicações que estudam OpenCL de uma maneira geral.

3.1 SIMD em Propósito Geral

Um dos primeiros trabalhos que investe em recursos de *hardware* para permitir execução SIMD em processadores de propósito geral é mostrado por Gummaraju et al. (2007). Os autores argumentam que, embora processadores SIMD e de propósito geral apresentem diferenças em se tratando de programação, eles também apresentam similaridades arquiteturais que fazem com que pequenas alterações no projeto de um processador de propósito geral sejam suficientes para que eles possam executar programas SIMD eficientemente. Segundo eles, a única modificação necessária é um mecanismo para executar assincronamente operações de leitura e escrita de grande quantidades de dados entre a memória local e principal. A solução proposta no artigo consiste em uma unidade de leitura e escrita chamada SLS (do inglês, *Stream Load/ Store*). Essa unidade pode ser logicamente vista como uma extensão e generalização de um tradicional *hardware* de *prefetch*[1] e uma unidade de *write-back*[2], capaz de transferir grandes quantidades de dados, potencialmente de regiões não contíguas, entre as memórias cache e principal. Os resultados apresentados no artigo mostram que, com o aumento de somente 1% em área do chip, foi possível aumentar a eficiência no uso dos recurso da arquitetura.

A Figura 15 mostra, para um conjunto de aplicações exploradas pelos autores, as frações do tempo total de execução, durante as quais há execução do *kernel* (%kernel) e acesso à memória utilizando a unidade SLS (%SLS). Foram executados quatro algoritmos, utilizando uma variedade de configurações para cada um, são eles: FEM (BARTH, 2000), CDP (MAHESH et al., 2006), SPAS (VUDUC et al., 2002) e NEO (BAŞAR; ITSKOV, 1999). O objetivo é que o %kernel ou o %SLS estejam em 100% indicando uma completa sobreposição de execução. Quando uma aplicação é *CPU-bound*[3] (FEM e NEO), todo o tempo de execução é utilizado realizando computação dos *kernels*, e o tempo usado

[1] Mecanismo que faz busca de dados de uma memória mais longe para uma próxima, antes que o dado seja requisitado.

[2] O dado é escrito na cache toda vez que ele é modificado, mas só é escrito na memória principal em certos intervalos de tempo ou sobre certas condições.

[3] Aplicações em que o tempo total de execução depende mais do tempo de processamento do que do tempo de acesso à memória.

pela unidade SLS é completamente escondido. Similarmente, quando as aplicações são *memory-bound*[4] (CDP e SPAS), todo o tempo é utilizado pela unidade SLS, e o tempo de computação é completamente sobreposto.

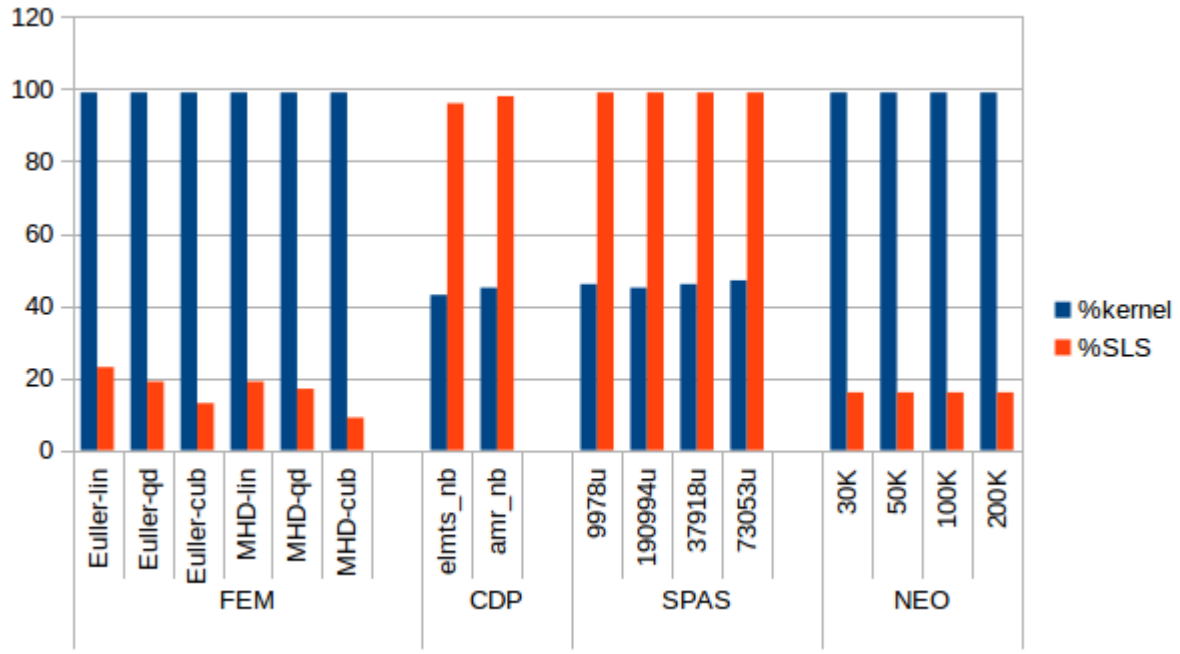

Figura 15 – Percentuais dos tempos de execução e utilização da unidade SLS.

Embora os resultados mostrados sejam satisfatórios, a abordagem apresentada restringe-se apenas à utilização da memória, negligenciando a intensa troca de contexto que existe em uma execução SIMD. Dessa forma, em uma aplicação com um número elevado de *threads*, a arquitetura seria penalizada.

Outra abordagem encontrada na literatura é o mapeamento de aplicações SIMD para CPU utilizando um sistema de compilação (GUMMARAJU; ROSENBLUM, 2005). Alguns aspectos do mapeamento proposto são diretos, por exemplo, a memória privada e a global da execução SIMD são mapeados diretamente para o conjunto de registradores e memória principal de uma CPU, respectivamente. A Figura 16 mostra os tempos de execução (em milhões de ciclos) do algoritmo FEM (BARTH, 2000) escrito na forma regular e na forma SIMD. Foram obtidas acelerações entre 1.13x a 1.26x. Uma das principais razões desse desempenho é o fato de as operações de computação e acesso à memória serem eficientemente sobrepostas.

Essa abordagem também mostrou alguns resultados satisfatórios. No entanto, a solução pode ser considerada paliativa, tendo em vista que o gerenciamento da cache como memória local é forçado, e subutiliza o espaço alocado.

[4] Aplicações em que o tempo total de execução depende mais do tempo de acesso à memória do que do tempo processando.

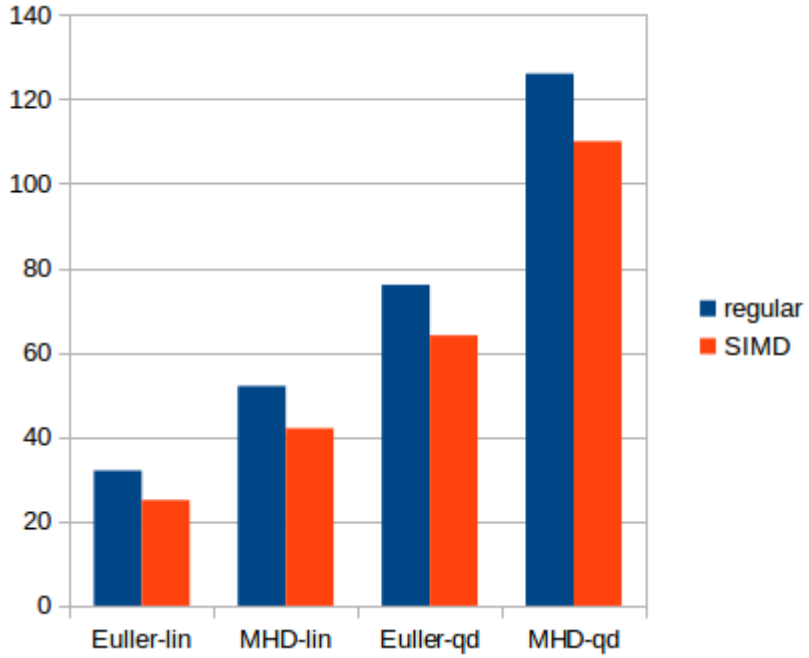

Figura 16 – Tempos de execução do algoritmo FEM.

Recentemente, em Morad, Yavits e Ginosar (2015), foi proposto um processador híbrido, propósito geral e SIMD, chamado GP-SIMD. Ele é composto por uma CPU sequencial, um array de blocos de memória compartilhada com acesso em duas dimensões, caches de instruções e dados, um coprocessador SIMD, um sequenciador SIMD, uma árvore de redução e uma NoC responsável por interligar os componentes. O coprocessador contém uma grande quantidade de unidades de processamento. O sequenciador é o meio de comunicação entre a CPU e coprocessador. A árvore de redução é um somador que possibilita uma soma paralela dos valores das unidades de processamento. Não é necessário sincronização de dados entre o segmento sequencial e o paralelo, uma vez que ambos acessam a mesma memória. A Figura 17 mostra uma representação do GP-SIMD.

Os autores listam cinco vantagens que o GP-SIMD exerce sobre as arquiteturas SIMD convencionais:

- Não existe a necessidade de transferência de dados entre a memória sequencial e a SIMD;
- Permite execução concorrente do processador sequencial e do coprocessador SIMD, permitindo que a CPU delegue tarefas ao coprocessador enquanto continua processando algumas funções sequenciais;
- A quantidade de unidades de processamento SIMD coincide com o número de linhas de memória, possibilitando máximo paralelismo;

- Permite a CPU endereçar associativamente o *array* de memória, permitindo redução de complexidade de código para alguns algoritmos sequenciais;
- A dissipação de potência é distribuída uniformemente sobre todo o *array* em vez de em um pequeno número de complexos processadores.

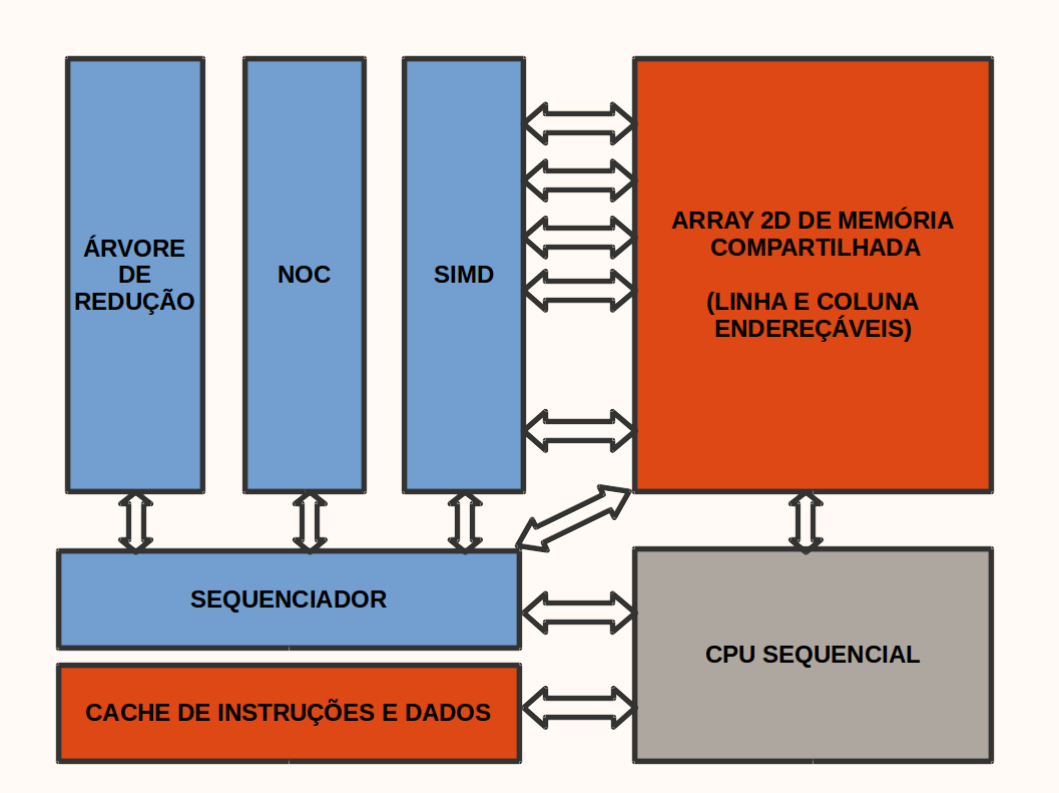

Figura 17 – Estrutura do processador GP-SIMD.

3.2 Trabalhos sobre OpenCL

As pesquisas relacionadas a OpenCL estão geralmente direcionadas a cinco tópicos: (i) revisões e avaliações da plataforma; (ii) implementação ou adaptação de algoritmos; (iii) tradução automática de outras linguagens; (iv) melhorias e implementações da linguagem para arquiteturas específicas; (v) soluções em *hardware*. As próximas seções mostra alguns trabalhos para cada item citado.

3.2.1 Revisões e avaliações

Chu e Hsiao (2010) faz uma avaliação positiva da linguagem OpenCL, com ênfase na relação custo-benefício de sua utilização em computação de alto desempenho. Os autores argumentam que o baixo custo das GPUs, associado ao uso da linguagem, tornou a computação de alto desempenho bem mais acessível. Para provar que é possível obter

alta performance a um baixo custo, foram realizados alguns experimentos utilizando plataformas que variaram de preço de USD$ 350 a USD$ 30000. As aplicações OpenCL executaram de 20 a 100 vezes mais rápido, em comparação às mesmas aplicações escritas em OpenMP, ou seja, os resultados se mostraram favoráveis à argumentação inicial dos autores.

Outros artigos também realizam comparações entre OpenCL e outras linguagens. Fang, Varbanescu e Sips (2011), por exemplo, realiza uma análise das diferenças de desempenho entre OpenCL e CUDA. A análise leva em consideração o modelo de programação, estratégias de otimização, detalhes de arquitetura e compiladores. Os resultados mostraram que, para a maioria das aplicações, CUDA apresentou uma performance 30% melhor que OpenCL. No entanto, essa diferença foi devida ao que os autores chamaram de *comparação injusta*. Eles afirmam que, sob condições justas, OpenCL pode alcançar performance similar ou superior à demonstrada por CUDA. A publicação finaliza oferecendo uma lista com oito passos para criar as condições justas que permitam uma comparação mais adequada: (i) descrição do problema; (ii) tradução em algoritmo; (iii) implementação; (iv) otimizações; (v) compilação e otimização (1º estágio); (vi) compilação e otimização (2º estágio); (vii) configuração e preparação do programa; (viii) execução.

Além dos artigos que realizam comparações entre OpenCL e outras linguagens, existem trabalhos que comparam a execução de OpenCL entre as diferentes plataformas que ele pode executar. Shen et al. (2013), por exemplo, faz uma análise detalhada dos fatores que impactam no desempenho de aplicações OpenCL em CPUs. Os autores focam nas duas principais diferenças entre as plataformas GPU e CPU: granularidade de paralelismo e modelo de memória. O artigo conclui que são oito os principais fatores que contribuem para essa diferença de desempenho: (i) transferência de dados entre o *host* e os dispositivos; (ii) padrão de acesso à memória; (iii) uso da memória local; (iv) granularidade de paralelismo; (v) *tiling*[5]; (vi) vetorização explícita; (vii) vetorização implícita; e (viii) dimensão do *work-group*.

Tendo em vista esse problema, Su et al. (2012) apresentam técnicas de programação para minimizar a diferença de desempenho entre as plataformas. O *benchmark* utilizado foi uma mini-aplicação chamada Hydro. O código foi compilado utilizando o compilador OpenACC CAPS, e foi executado nas plataformas Intel Xeon Phi, Nvidia k20C e AMD 7970 GPU. Os resultados mostraram que é possível diminuir a diferença entre as performances para menos de 12%.

3.2.2 Implementação ou adaptação de algoritmos

Alguns trabalhos também foram publicados mostrando a eficiência da linguagem OpenCL para solução de alguns problemas. Razmyslovich et al. (2010), por exemplo,

[5] Processo de subdividir uma imagem gráfica com a finalidade de facilitar o uso dos recursos do *hardware*.

apresenta uma implementação do algoritmo *Smith-Waterman*. Os autores citam 5 vantagens de sua implementação:

1. A implementação é capaz de processar, eficientemente, longas sequências (acima de 28 milhões);

2. Os caminhos de alinhamento são calculados com eficiência;

3. O desempenho computacional da implementação é similar à implementação de Farrar (2007), e três vezes mais rápida que a implementação de Liu, Schmidt e Maskell (2010);

4. Em comparação com implementação para CPU, o algoritmo apresentou uma aceleração de 9 vezes para a versão com cálculo de caminho, e 130 vezes para a versão sem cálculo de caminho;

5. Por último, os autores exploram a vantagem de um código OpenCL usufruir de portabilidade entre diferentes plataformas.

Yan, Shi e Sun (2012) apresenta o projeto e implementação de uma suíte de *micro-benchmark* OpenCL para GPUs e CPUs. O trabalho objetiva fornecer um entendimento mais profundo das características arquiteturais do modelo OpenCL, para assim, facilitar o desenvolvimento de aplicações para uma desejada plataforma. A execução do *micro-benchmark*, em uma CPU AMD e em uma GPU Nvidia, evidenciou as diferenças arquiteturais entre os dois processadores.

3.2.3 Tradução automática de linguagens

Outros segmentos de pesquisa são a tradução automática de OpenCL para linguagens de descrição de *hardware* (do inglês *Hardware Description Language — HDL*) e de outras linguagens para OpenCL.

Czajkowski et al. (2012) apresenta uma estrutura de compilação para gerar *hardware* de alto desempenho em FPGAs. O compilador proposto recebe como entrada um conjunto de *kernels* e o programa host. Os *kernels* são compilados em um circuito de *hardware*. O processo começa com um *parse* da linguagem C, que produz uma representação intermediária, na forma de instruções e dependências, para cada *kernel*. A representação é otimizada objetivando uma determinada plataforma FPGA, então, é convertida para um grafo de fluxo de controle e dados (do inglês *Control Data Flow Graph* - CDFG), o qual pode ser otimizado para diminuir o consumo de área e melhorar o desempenho do sistema. Já o programa *host* é compilado usando um compilador C/C++.

A proposta gera circuitos que operam a uma frequência que excede os 160MHz, dando suporte a sincronização utilizando barreiras, uso de memória local e operações de ponto flutuante.

Um exemplo de trabalho que traduz uma terceira linguagem para OpenCL é Martinez, Gardner e Feng (2011). Nesse artigo, os autores propõem um tradutor automático de CUDA para OpenCL, chamado CU2CL. O tradutor suporta as principais funcionalidades da API CUDA. Para demonstrar isso e a eficiência do tradutor, os autores traduziram algumas aplicações da SDK CUDA e da suíte de *benchmark* Rodinia (CHE et al., 2009). A Tabela 2 sumariza os resultados obtidos com as traduções. As aplicações variam em tamanho, entre centenas e milhares de linhas de código. No entanto, observa-se que o tempo de tradução não é estritamente dependente do tamanho. Em geral, os programas com mais funções em CUDA, ou mais complicados, tendem a demorar mais.

Tabela 1 – Resultados obtidos com CU2CL

Fonte	Aplicação	Linhas CUDA	Tempo Total de Tradução (s)	Tempo CU2CL (ms)	Linhas Modificadas Manualmente	Percentual Traduzido Automaticamente
CUDA SKD	asyncAPI	136	0.331	3.35	4	97.1
	bandwidthTest	891	0.623	7.98	9	98.9
	BlackSholes	347	0.606	5.24	4	98.9
	matrixMul	351	0.607	5.47	2	99.4
	scalarProd	171	0.327	3.75	4	97.7
	vectorAdd	147	0.287	3.11	0	100.0
Rodinia	Back Propagation	313	0.300	4.46	5	98.4
	Breadth-First Search	306	0.301	4.51	8	97.4
	Hotspot	328	0.297	4.90	7	97.9
	Needleman-Wunsch	418	0.303	5.46	0	100.0
	SRAD	541	0.303	6.56	0	100.0

3.2.4 Implementação do framework OpenCL

Na literatura são encontrados alguns trabalhos que propõem uma implementação do OpenCL, seja uma proposta de melhoria para uma implementação já existente, ou uma nova implementação para uma arquitetura específica.

Lee et al. (2011), por exemplo, apresenta um projeto e implementação de um *framework* OpenCL para processadores *manycores*, homogêneos e sem coerência de cache, assim como o Intel SCC. Os autores argumentam que o modelo de programação OpenCL pode ser uma camada de software ideal para tais arquiteturas. A implementação proposta mapeia cada núcleo do Intel SCC para uma unidade de computação no modelo de plataforma OpenCL. Para emular eficientemente os elementos de processamento em um único núcleo, o compilador proposto aplica uma técnica de coalescência[6]. Os testes mostraram que, utilizando a implementação OpenCL proposta, a arquitetura Intel SCC alcança melhor ou igual escalabilidade quando comparado à utilização dos sistemas convencionais para *manycores*.

[6] Transformar os *work-items* em um único laço de repetição.

Nah et al. (2013) apresenta técnicas de otimização para compiladores que objetivam processadores reconfiguráveis. A arquitetura alvo explorada no artigo consiste em um processador de propósito geral e um acelerador embutido, configurável e com unidades vetoriais. O acelerador é capaz de alternar sua arquitetura entre o modo VLIW (do inglês *Very Large Instruction Word*) e o modo CGRA (do inglês *Coarse Grained Reconfigurable Array*). Para explorar o poder de processamento paralelo da arquitetura, o compilador mapeia códigos paralelos para o modo CGRA, traduzindo o código para uma versão vetorizada. Os resultados dos teste mostraram que a abordagem proposta é efetiva e promissora para a arquitetura explorada.

3.2.5 Soluções em *hardware*

Publicações que propõem soluções em *hardware* para execução OpenCL são as mais escassas. Uma publicação, entre as poucas, é proposta por Chen e Chen (2014). O artigo apresenta uma plataforma de simulação *manycore* baseada em ARM, e um sistema de tempo de execução OpenCL para essa plataforma. O sistema de tempo de execução inclui um compilador *on-the-fly* e recursos para gerenciar a plataforma.

A arquitetura do *manycore* conta com um conjunto de unidades de computação (do inglês *Compute Unit* - CU), cada uma composta por um processador ARM, duas caches de nível 1 para dados e instruções, uma unidade de gerenciamento de memória (do inglês *Management Memory Unit* - MMU) e uma cache de nível 2 unificada. A comunicação entre as unidades de computação é feita utilizando um barramento, que serve também para comunicar com a CPU e acessar a memória externa.

Tendo em vista que o modelo de memória da arquitetura não coincide com o do modelo OpenCL, a MMU é responsável por gerenciar o endereçamento, convertendo endereços virtuais em endereços reais. Nessa conversão, a MMU faz um mapeamento entre as regiões de memória do OpenCL (global, constante, local e privada) e o endereço gerado. Outro artifício utilizado na proposta foi a coalescência de *work-items*, que consiste em agrupar vários work-items em um único laço de repetição. Essa medida foi necessária devido ao reduzido número de elementos de processamento disponibilizado na arquitetura, o que causaria uma intensa troca de *work-items*, resultando em queda de desempenho. Os resultados dos testes executados mostraram que a técnica de coalescência de *work-items* alcançou uma média de aceleração de 4 vezes.

Outro trabalho encontrado na literatura, foi proposto pelo próprio autor desta dissertação de mestrado (NEPOMUCENO et al., 2015). O artigo apresenta uma implementação VHDL de uma arquitetura *multicore*, de propósito geral e provida de recursos em *hardware* para facilitar a execução OpenCL. A arquitetura possui um modelo de execução chamado de mestre-escravo, sendo composta por um processador mestre, oito processadores escravos, uma *Memory Management Unit*, uma memória principal, uma memória local,

nove memórias de instruções (uma para cada processador) e um barramento para conectar os dispositivos. A Figura 18 mostra o diagrama de blocos da arquitetura proposta por Nepomuceno et al. (2015).

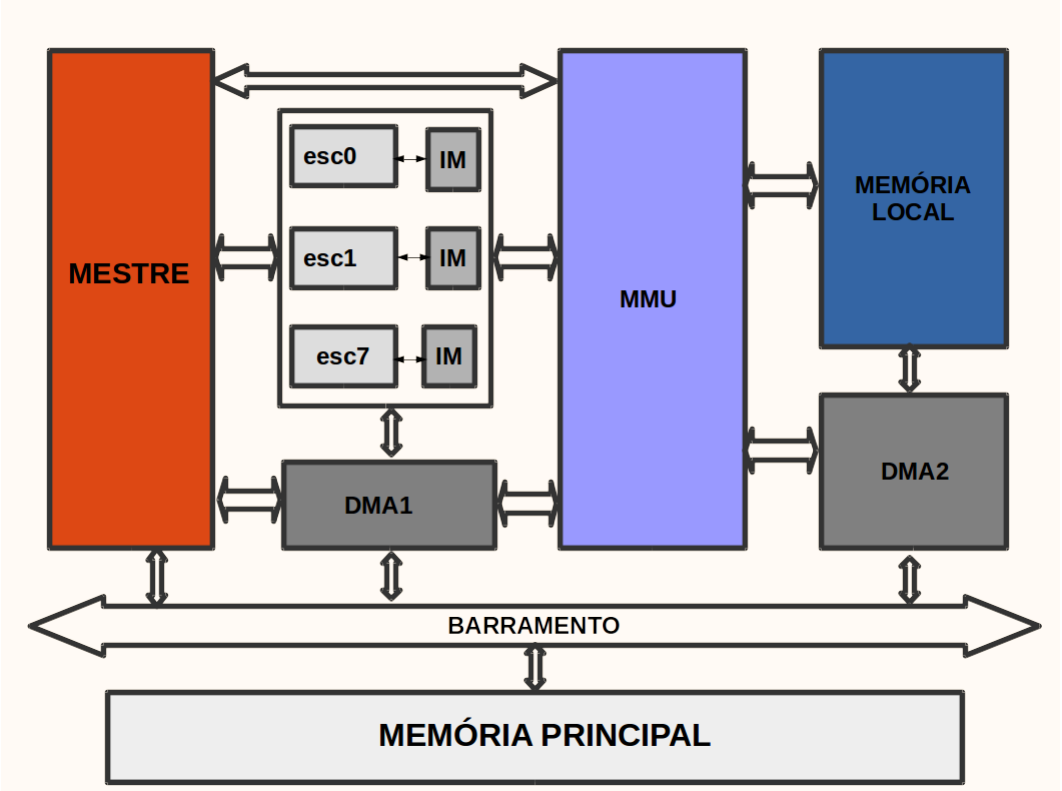

Figura 18 – Diagrama de blocos da arquitetura.

O processador mestre é responsável por executar o programa *host* de uma aplicação OpenCL, enquanto os escravos são responsáveis por executar o programa kernel. A MMU é responsável por fazer o mapeamento do modelo de memória OpenCL (global, constante, local e privada) para os endereços físicos da memória local. Já os DMAs 1 e 2, são responsáveis por fazer as transferências de dados entre a memória principal, a memória local e as memórias de instruções dos escravos.

Os testes realizados na plataforma apresentaram resultados satisfatórios quanto a execução de programas OpenCL. No entanto, a arquitetura apresenta restrições quanto ao escalonamento do número de processadores, uma vez que utiliza um barramento como sistema de interconexão. Uma solução seria substituir o barramento por uma rede em chip e implementar um escalonador em *hardware*. O escalonador solucionaria o problema do escalonamento interno a um nó, evitando que, a cada troca de *work-item*, o nó escravo acionasse o mestre, sobrecarregando, assim, o uso da rede em chip. Esse trabalho de Dissertação apresenta justamente essas duas soluções.

3.3 Considerações finais

Nesse capítulo de estado da arte, foram mostrados alguns trabalhos que, em algum nível, se relacionam como trabalho proposto nesta dissertação. O capítulo foi dividido em duas seções. A primeira mostrou algumas publicações voltadas a investigar a execução de código SIMD em arquiteturas de propósito geral, mostrando suas virtudes e defeitos. O objetivo era mostrar que, mesmo já tendo alguns trabalhos na literatura, ainda existe espaço para mais pesquisa, uma vez que nenhum trabalho propôs uma arquitetura homogênea com suporte a SIMD, como apresentado nesta dissertação.

A segunda parte tratou de mostrar as pesquisas referentes a OpenCL. O objetivo era justificar o uso do OpenCL como a linguagem SIMD para arquitetura, e mostrar as dificuldades de programar OpenCL para propósito geral.

Os próximos capítulos deste documento mostram em detalhes a arquitetura desenvolvida, os resultados obtidos com os testes e trabalhos futuros.

4 A Plataforma METAL

Como mostrado no Capítulo 3, existem alguns trabalhos na literatura que propõem meios para melhorar a execução SIMD em arquiteturas de propósito geral. No entanto, observa-se que a maioria das abordagens propõem arquiteturas heterogêneas equipadas com recursos para que os módulos SIMD trabalhem harmonicamente com os módulos CPU. Neste trabalho usa-se uma abordagem diferente. Desenvolveu-se e implementou-se a plataforma METAL, que consiste em uma arquitetura paralela de propósito geral, homogênea e equipada com recursos que facilitam a execução SIMD da linguagem OpenCL.

A generalidade da arquitetura é garantida pela utilização do processador de propósito geral, MIPS, e recursos em *hardware* que permitam a programação paralela com compartilhamento de variáveis. Os recursos que facilitam a execução SIMD consistem em uma hierarquia de memória adaptada ao modelo OpenCL e um escalonador de *work-items*. A arquitetura é interconectada utilizando a rede em chip SoCIN. Seu modo de funcionamento é baseado no estilo mestre-escravo. Um nó da rede, dito mestre, envia tarefas para serem executadas nos nós restantes, chamados de escravos. A Figura 19 mostra uma visão geral da plataforma.

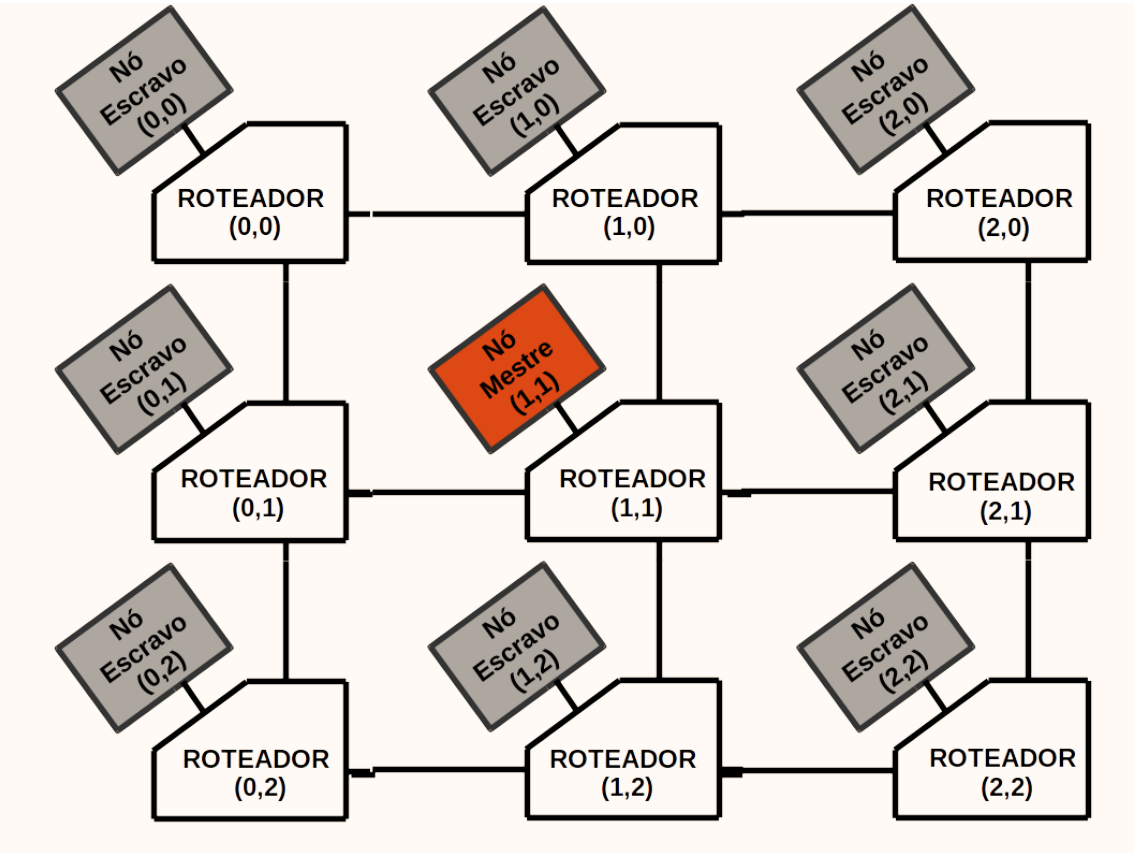

Figura 19 – Representação gráfica da plataforma METAL

4.1 Nó Mestre

O nó mestre tem a função de gerenciar a execução em toda a plataforma. Sua composição inclui três memórias (instrução, dados e da rede), um *Master Processor* (MP), um DMA, uma *Network Interface* (NI), um árbitro para controlar o acesso a NI e um módulo em *hardware* responsável por gerenciar a execução OpenCL, chamado de *Master OpenCL Hardware Manager* (MOH). A localização do nó mestre é central na rede, para facilitar a comunicação com todos os nós. A Figura 20 mostra o diagrama de blocos do nó mestre.

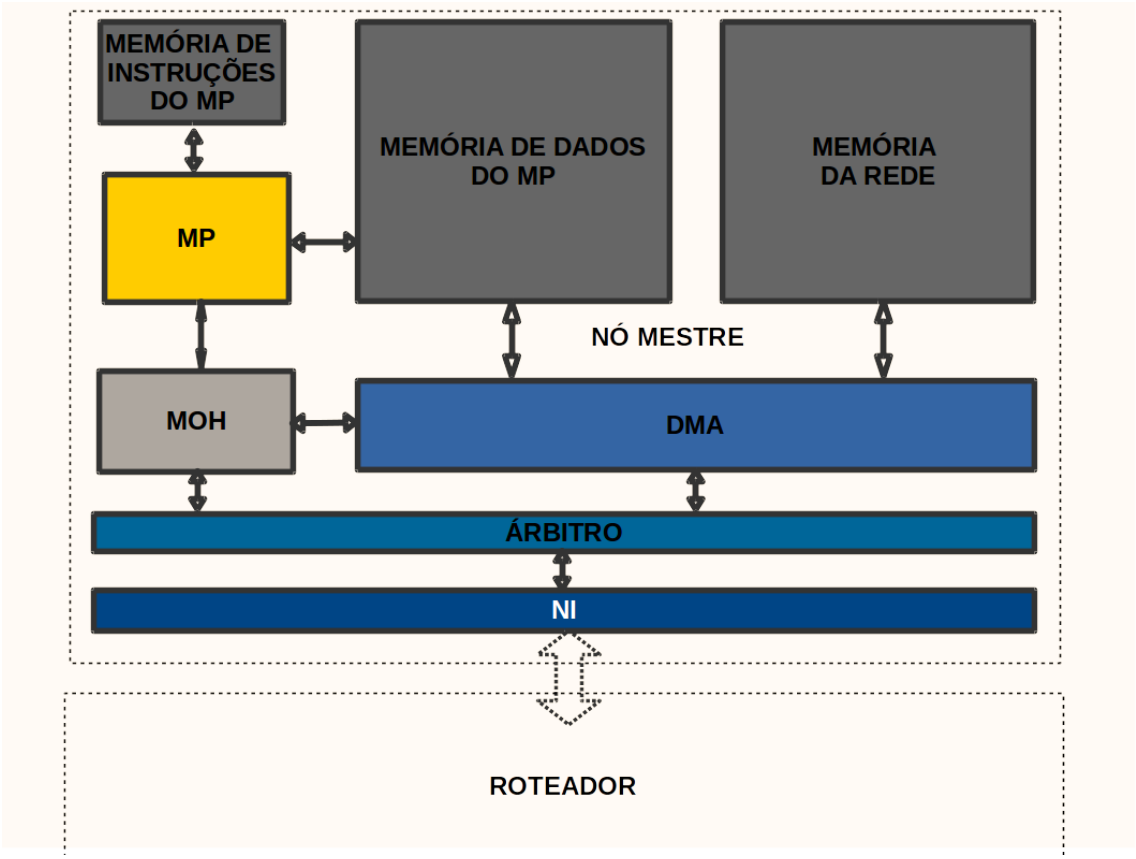

Figura 20 – Representação gráfica do nó mestre.

4.1.1 MP, Memória de Instruções e Memória do *Host*

Os componentes MP, memória de instruções e memória de dados, juntos, representam um ambiente CPU tradicional. Fazendo uma analogia ao modelo de plataforma OpenCL, tais componentes representariam o processador *host*. O MP é uma implementação do MIPS (HENNESSY et al., 1982) com memória de dados e instruções separadas. A única alteração feita foi em relação às instruções de acesso à memória, que passaram a ser bloqueantes. Essa mudança foi necessária devido à concorrência entre o MP e o DMA ao acesso à memória de dados do MP. Excetuando essa alteração, é um ambiente MIPS convencional.

4.1.2 MOH, Memória da Rede e DMA

A memória da rede é a região de memória compartilhada por todos os nós escravos da rede. Esta representa as memórias global e constante no modelo de memória OpenCL. Seu acesso ocorre somente via DMA (do inglês, *Direct Memory Access*), que transfere grandes quantidades de dados entre a memória de dados do MP, memória da rede e os nós escravos na NoC.

O MOH é o responsável por dar suporte as funções OpenCL, do programa *host*, que necessitam de interação com o *hardware* e atender às requisições enviadas pelos nós escravos. As funções que interagem com MOH são a *clEnqueueWriteBuffer*, a *clEnqueueReadBuffer* e a *clEnqueueNDRangeKernel*. Sempre que essas três funções são executadas no MP, este aciona o MOH utilizando um mecanismo de comunicação igual à comunicação com um dispositivo de entrada e saída mapeada na memória.

Para as duas primeiras funções, o MP informa ao MOH a quantidade de dados a serem transferidos e os endereços fonte e destino. Em posse dessas informações, o MOH aciona o DMA para que a transferência seja efetivada. Já na função *clEnqueueNDRangeKernel*, o MP informa ao MOH a configuração da *grid* de execução (quantidade de *work-items* e *work-groups* nos eixos x, y e z) e o ponteiro para a função *kernel*. O MOH, por sua vez, dispara a execução nos nós escravos. A função *kernel*, em execução nos escravos, aciona o MOH em casos de acesso à memória da rede, finalização de *work-group* ou requisição e liberação de *mutex*.

O MOH é composto pelos componentes *Host Interface* (HI), *Master Output NoC* (MON), *Master Input NoC* (MIN), FIFO e *Mutex Memory* (MM). A Figura 21 representa o diagrama de blocos do componente MOH.

Os componentes FIFO e MM são blocos de memória. A FIFO consiste em um bloco de memória do tipo fila que serve como um meio de comunicação entre os componentes MON e MIN. O MM, por sua vez, armazena informações referentes ao gerenciamento e uso de *mutex* para programação paralela no modelo de memória compartilhada. O gerenciamento é realizado por intermédio de uma estrutura de dados que contém um identificador, o status do *mutex* (livre ou em uso) e uma fila de requisições para cada *mutex*.

O HI é responsável por atender as requisições vindas do MP, que podem ser de transferência de dados ou início de execução. A requisição de transferência de dados, como já mencionado, é enviada pelas funções *clEnqueueWriteBuffer* e *clEnqueueReadBuffer*. Nesse caso, o HI aciona o DMA para realizar a transferência. Já a requisição de início de execução, enviada pela função *clEnqueueNDRangeKernel*, informa ao HI a configuração da *grid* de execução e um ponteiro para a função kernel. Essas informações são enviadas ao componente MON que, por sua vez, dispara a execução da função kernel nos escravos.

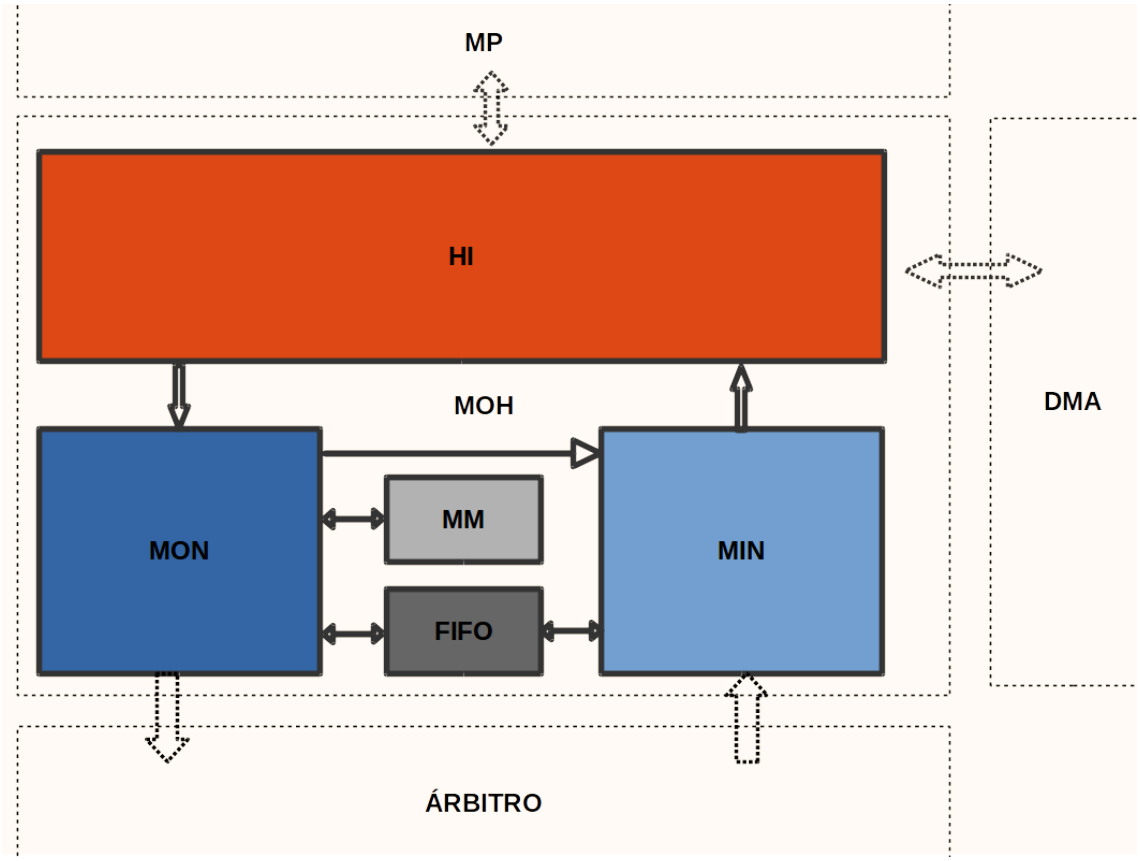

Figura 21 – Representação gráfica do componente MOH.

O componente MON envia para cada nó da rede o código do programa kernel a ser executado, a configuração da *grid* de execução e um identificador de *work-group*. Cada nó da rede será responsável por realizar a execução de todos os work-items pertencentes ao *work-group* recebido.

Além de disparar a execução nos nós escravos, o componente MON tem como atribuição realizar toda operação que injeta mensagem na rede. Essas operações são requisitadas pelo componente MIN, por intermédio do componente FIFO. O componente MIN, por sua vez, é responsável por receber todas as requisições que chegam da rede e enfileirá-las para que o MON as execute. A Tabela 2 mostra os tipos de requisições que podem chegar pela rede, enviadas pelos nós escravos, assim como ações tomadas pelos componentes MIN e MON.

4.1.3 Árbitro e NI

O árbitro serve para controlar os acessos à NI, que podem ser solicitados tanto pelo MOH quanto pelo DMA. A NI é responsável por encapsular as mensagens que serão enviadas pela rede. Cada mensagem é constituída por três partes: (i) uma palavra contendo as coordenadas x e y do remente e do destinatário (quatro bits para cada coordenada); (ii) uma palavra de 32 bits informando a quantidade de palavras da carga útil; (iii) um

Tabela 2 – Execução dos componentes MIN e MON.

REQUISIÇÃO	AÇÃO DO MIN	AÇÃO DO MON
LEITURA	Insere na FIFO uma requisição de leitura contendo a página requisitada e o nó requerente.	Envia ao nó requerente a página solicitada.
ESCRITA	Insere na FIFO uma requisição de escrita contendo o endereço e dado a ser escrito.	Escreve o dado informado na memória da rede.
OBTENÇÃO DE MUTEX	Insere na FIFO uma requisição de mutex e nó requerente	Verifica se o mutex solicitado está livre. Se estiver, altera o status do mutex para em uso e o retorna ao nó requerente. Caso não esteja, insere na fila do mutex uma requisição informando o nó requerente.
LIBERAÇÃO DE MUTEX	Insere na FIFO uma requisição de liberação de mutex contendo o código do mutex.	Verifica se existe alguma solicitação na fila do mutex informado. Se existir, retorna o mutex ao primeiro da fila. Caso não exista, altera o status do mutex para livre.
FINALIZAÇÃO	Verifica se ainda existe algum work-group para ser executado. Se existir, insere na FIFO uma requisição de finalização de work-group informando o nó requerente e o código do novo work-group a ser executado. Caso não exista, aguarda até que todos os work-groups finalizem para informar do fim da execução.	Envia o novo identificador de work-group ao nó requerente.

determinado número de palavras contendo a carga útil. A comunicação com a NI é realizada utilizando um protocolo *handshake*[1].

4.2 Nó Escravo

Os nós escravos são responsáveis por realizar a computação massiva das aplicações. No modelo de plataforma OpenCL, cada nó escravo representa uma unidade de computação. Eles são compostos por um conjunto de processadores chamados de *Slaves Processors* (SPs), cada um com sua memória de instruções, um escalonador de *work-items*, uma *Memory Management Unit* (MMU), uma memória interna, uma *Network Interface* (NI) e

[1] É o processo pelo qual duas máquinas afirmam uma a outra que a reconheceu e está pronta para iniciar a comunicação.

um módulo em *hardware* responsável por gerenciar a execução OpenCL, chamado de *Slave OpenCL Hardware Manager* (SOH). A Figura 22 ilustra uma representação em blocos de um nó escravo.

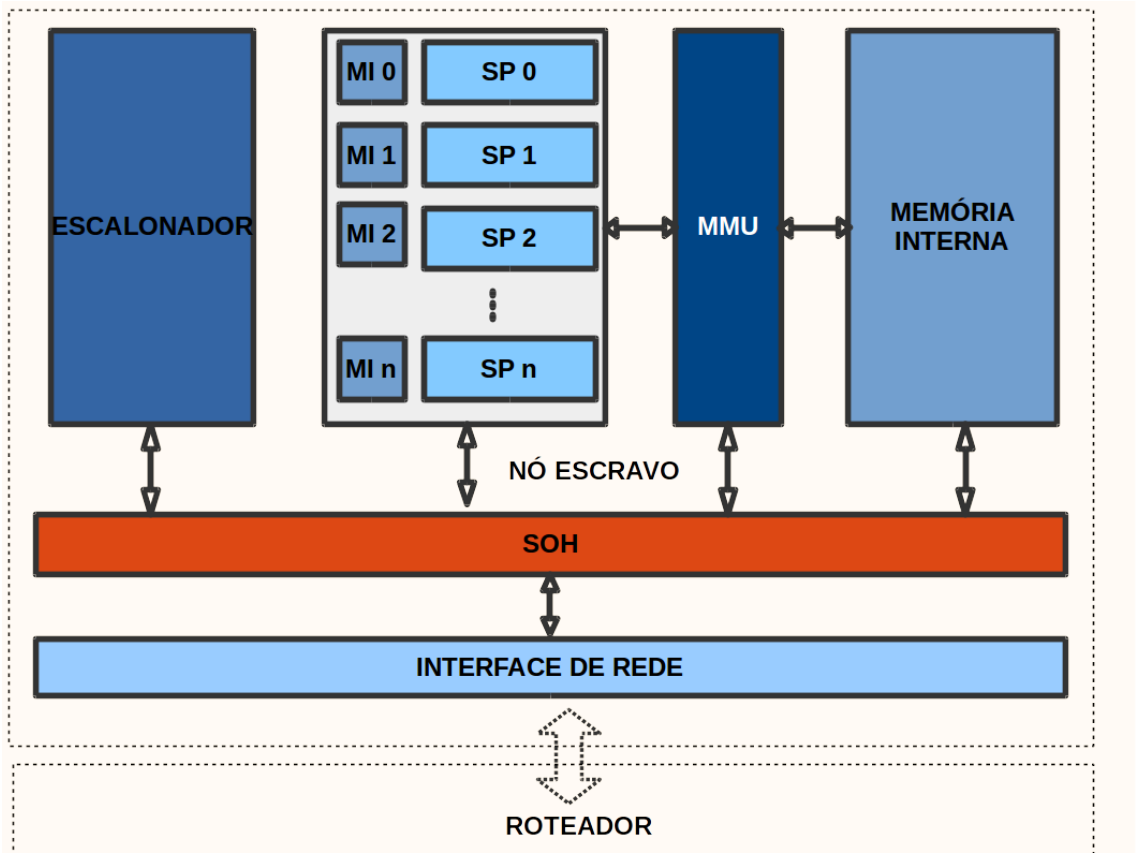

Figura 22 – Representação gráfica do nó escravo.

4.2.1 SOH

O SOH é o componente responsável por dar suporte às funções OpenCL que necessitam de interação com o *hardware*, dar suporte às funções de programação por variável compartilhada e realizar a comunicação com o nó mestre. Ele é composto por um array de *Slave Interface* (SIs), um *Slave Input NoC* (SIN), *Slave Output NoC* (SON) e uma rede de interconexão ponto a ponto que interliga todos esses componentes. A Figura 23 mostra um diagrama em blocos do componente SOH.

Os SIs são responsáveis por atender as requisições enviadas, via mapeamento de memória, pelos SPs. Estes, por sua vez, podem requisitar informações sobre configuração da *grid* de execução, escalonamento, acesso ou liberação de *mutex*. A configuração da *grid* é requisitada pelas funções opencl *get_global_id*, *get_local_id*, *get_local_size* e *get_global_size*. Essas funções solicitam, respectivamente, os identificadores globais *xyz* do *work-item* em execução, os identificadores locais (dentro do *work-group*) *xyz* do *work-item* em execução, a quantidade global de work-items nos eixos *xyz* e a quantidade local (dentro

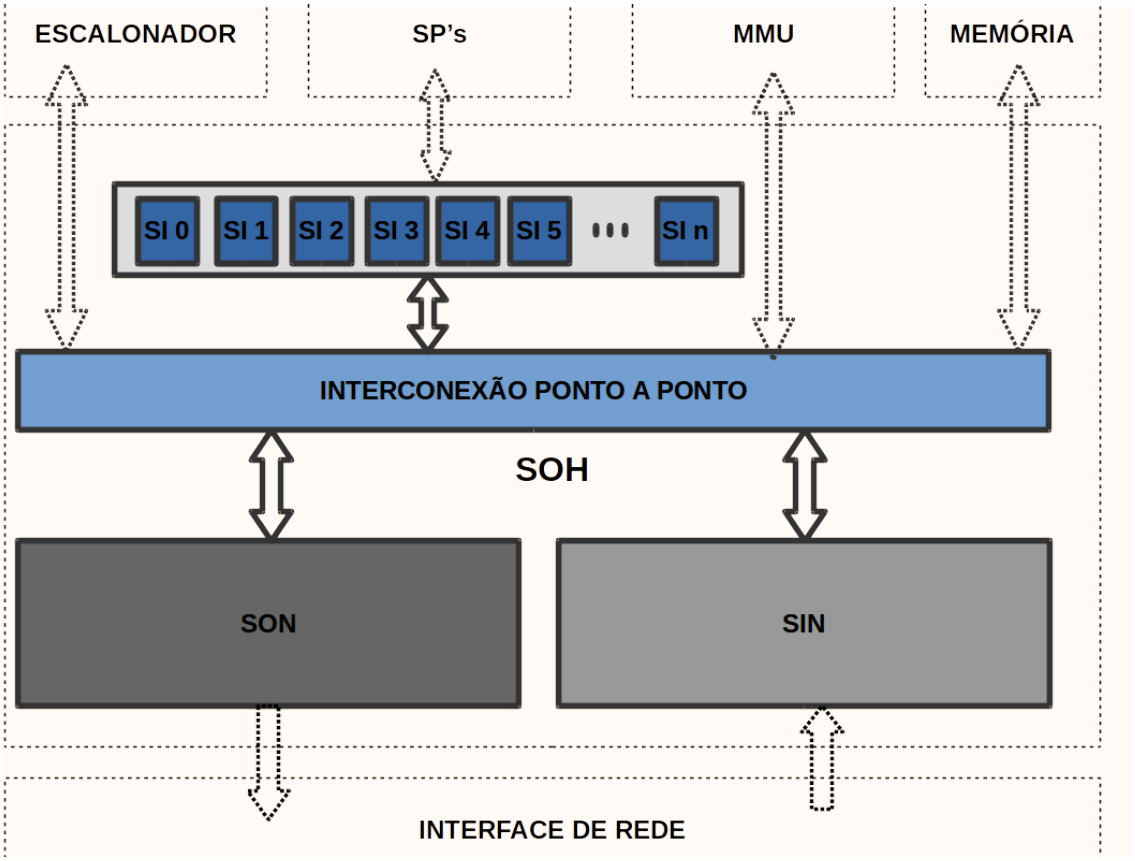

Figura 23 – Diagrama de blocos do componente SOH.

do *work-group*) de work-items nos eixos *xyz*. Todas essas informações são recebidas na inicialização do nó escravo e são retornadas ao SP. Uma requisição de escalonamento é recebida quando um SP alcança uma função *barrier* ou finaliza a execução de um *work-item*. Neste caso, o SI aciona o escalonador requisitando um novo work-item para o SP executar, se não existir nenhum *work-item* disponível, o SP referido permanece em estado de espera. A requisição de *mutex* ou liberação de *mutex* são geradas, respectivamente, pelas funções *mutex_lock* e *mutex_unlock*. Ao receber essas requisições, o SI aciona o componente SON que envia essas requisições ao nó mestre.

O componente SON é responsável por toda operação que injeta mensagens na rede. Além de ser acionado pelos SIs para enviar mensagens de requisição e liberação das variáveis *mutex*, ele pode ser acionado pela MMU e pelo escalonador. A MMU o aciona quando ocorre *miss* ou escrita na memória global. Nesse caso, o SON envia uma requisição ao nó mestre solicitando a página da memória da rede ou enviando um dado a ser escrito na mesma. O escalonador aciona o SON quando não existe mais *work-item* para ser executado no atual *work-group*, ou seja, a execução do *work-group* finalizou. O SON, por sua vez, envia uma mensagem ao nó mestre informando do término da execução do *work-group* e pedindo um novo *work-group* para executar.

O componente SIN é responsável por receber e tratar as mensagens que chegam

pela rede. Essas mensagens podem ser de quatro tipos:

- Inicialização: nessa mensagem a componente SIN recebe o código do programa kernel, a configuração da *grid* de execução e o identificador do *work-group* a ser executado. A SIN, por sua vez, aciona o escalonador para que ele inicie a execução nos SPs;
- Página de memória: a SIN recebe uma página da memória da rede previamente solicitada e a armazena na memória local;
- Mutex: A SIN recebe a confirmação de acesso a uma variável *mutex* previamente solicitada e informa ao SP requerente do recebimento;
- Novo *work-group*: a componente SIN recebe um novo identificador de *work-group* para executar e informa ao escalonador para reiniciar a execução no nó.

4.2.2 Escalonador

O escalonador é responsável por realizar a troca de *work-items* em execução nos SPs. O escalonamento é realizado mediante requisições enviadas pelos SIs. O componente escalonador possui três componentes: interface, gerenciador e memória; A Figura 24 mostra o diagrama de blocos do componente escalonador.

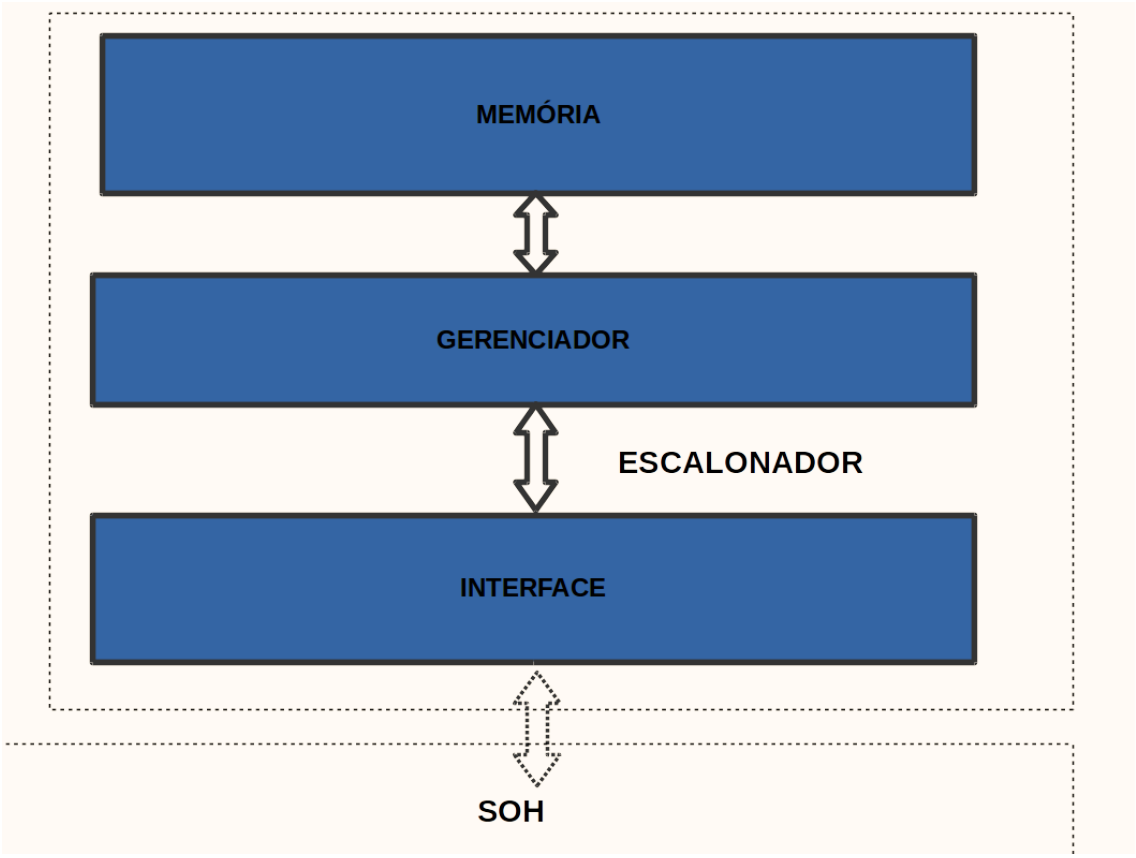

Figura 24 – Diagrama de blocos do componente escalonador

A interface é responsável por serializar o acesso ao escalonador. Ela recebe requisição dos SIs, arbitra utilizando a política *round-robin* e aciona o gerenciador. Essa operação é feita de acordo com a máquina de estados simplificada da Figura 25, representada em seis estados:

- INT_WAIT1 - é o estado inicial no qual permanece até que um *work-group* chegue para ser executado no nó;
- INT_START - aciona o escalonador requisitando os primeiros work-items. A quantidade de *work-items* requisitados é no máximo a quantidade de processadores disponíveis;
- INT_WAIT2 - aguarda até que chegue uma requisição de troca de contexto, ocasionada por uma barreira ou finalização de *work-item*;
- INT_REQUEST - requisita ao gerenciador um novo *work-item* disponível. Se houver, vai para INT_DISPATCH. Caso não haja, vai para INT_END.
- INT_DISPATCH - responde ao SI com um novo identificador de work-item e novo PC para execução;
- INT_END - informa ao SOH que a execução dos *work-items* finalizou.

O gerenciador implementa o algoritmo de escalonamento de work-items. Para tal tarefa, utiliza uma memória para armazenar o status (pronto, executando, barreira ou finalizado) e o PC de cada *work-item* do *work-group* em execução. Sua máquina de estados simplificada é mostrada na Figura 26 e possui oito estados:

- MANAGER_START - é o estado inicial. Permanece nesse estado até que o SOH o configure informando a disposição do *NDRange*;
- MANAGER_WAIT - aguarda até que um *work-item* finalize ou entre em barreira para ir para os estados MANAGER_FINISH ou MANAGER_BARRIER, respectivamente. O estado MANAGER_NEXT é chamado na inicialização dos SPs;
- MANAGER_NEXT - procura um *work-item* com o status pronto para executar. Se existir, vai para o MANAGER_DISPATCH. Caso contrário, vai para MANAGER_DONE;
- MANAGER_FINISH - atualiza o status do *work-item* que foi concluído para *finalizado*. Caso todos *work-itens* estejam finalizados, o próximo estado será *MANAGER_RESTART*. Do contrário, procura por outro *work-item* no estado pronto. Se existir, vai para o MANAGER_DISPATCH. Senão, vai para MANAGER_DONE;

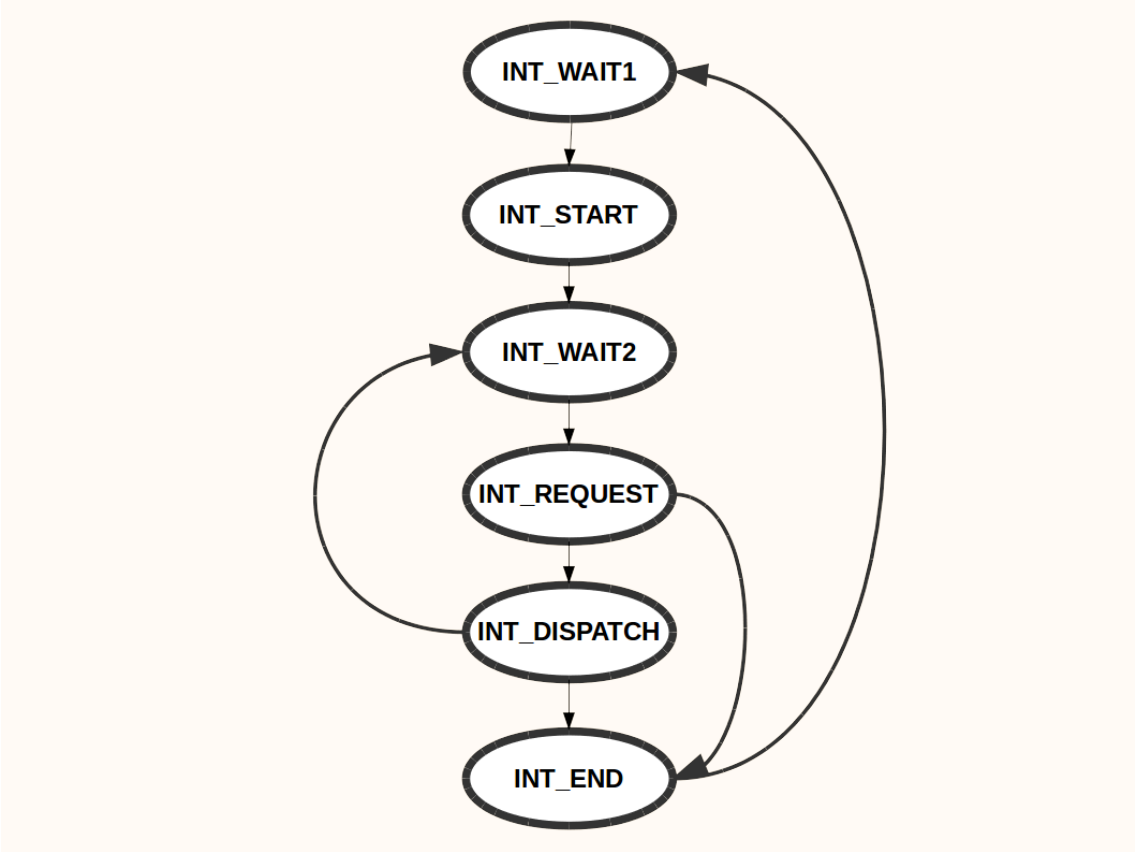

Figura 25 – Máquina de estados implementada pelo componente interface.

- MANAGER_BARRIER - atualiza o status do *work-item* que alcançou a função *barrier* e de verificar se todos os *work-items* estão na mesma função. Caso estejam, atualiza todos para *pronto* e segue para o estado *MANAGER_DISPATCH*. Se o status barreira não estiver determinado à todos os *work-items*, verifica se existe algum *pronto* e segue para MANAGER_DISPATCH. Caso não haja *work-items* disponíveis, vai para MANAGER_DONE;

- MANAGER_RESTART - informa ao SOH que a execução finalizou e volta para GERENCIADOR_START;

- GERENCIADOR_DISPATCH - informa um novo identificador do *work-item* e PC para o SOH, atualiza o status desse work-item para *executando*. Em seguida, vai para o estado GERENCIADOR_DONE;

- GERENCIADOR_DONE - volta para o estado GERENCIADOR_WAIT.

4.2.3 MMU

A MMU (do inglês *Memory Management Unit*) é responsável por gerenciar a memória interna do nó. De acordo com o modelo de memória OpenCL, um *work-item*

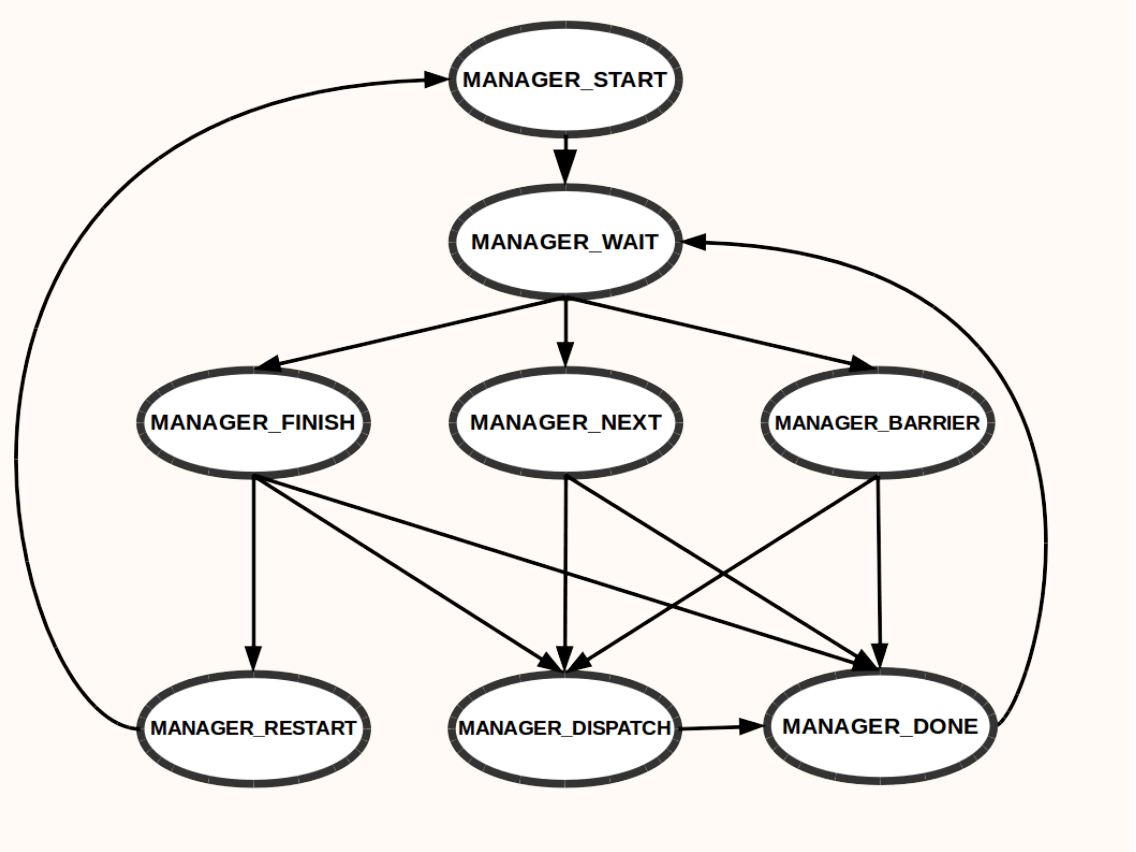

Figura 26 – Máquina de estados implementada pelo componente *manager*

em execução pode endereçar três tipos de memória: global/constante, local e privada. Por esse motivo, a memória interna também é dividida nessas três partes. A memória global/constante funciona como uma cache da memória da rede existente no nó mestre. A memória local é compartilhada por todos os *work-items* dentro do *work-group*. Já a memória privada é dividida, mas não compartilhada, entre todos os *work-items* existentes no *work-group*. A Figura 27 exemplifica a divisão da memória interna de um nó.

A MMU é composta por um conjunto de componentes responsáveis por fazer a tradução do endereço virtual para o endereço físico, chamados de *Translators Units* (TU), uma *Translation Lookaside Buffer* (TLB) e um componente responsável por fazer a comunicação com o SOH, chamado de *MMU Interface* (MI). A Figura 28 ilustra o diagrama de blocos da MMU.

Os componentes TUs recebem endereços virtuais, gerados pelos SPs, e os traduzem em endereços físicos da memória. Em caso de acesso à memória privada, a tradução é feita utilizando o identificador do *work-item* que está realizando o acesso. Já em caso de acesso à memória global, a tradução é feita consultando a TLB que, por sua vez, consiste em uma tabela que guarda as informações sobre o estado atual da memória. Os valores armazenados em cada linha da TLB são listados na Tabela 3.

Em caso de escrita ou *miss* na memória global, os componentes TUs acionam o com-

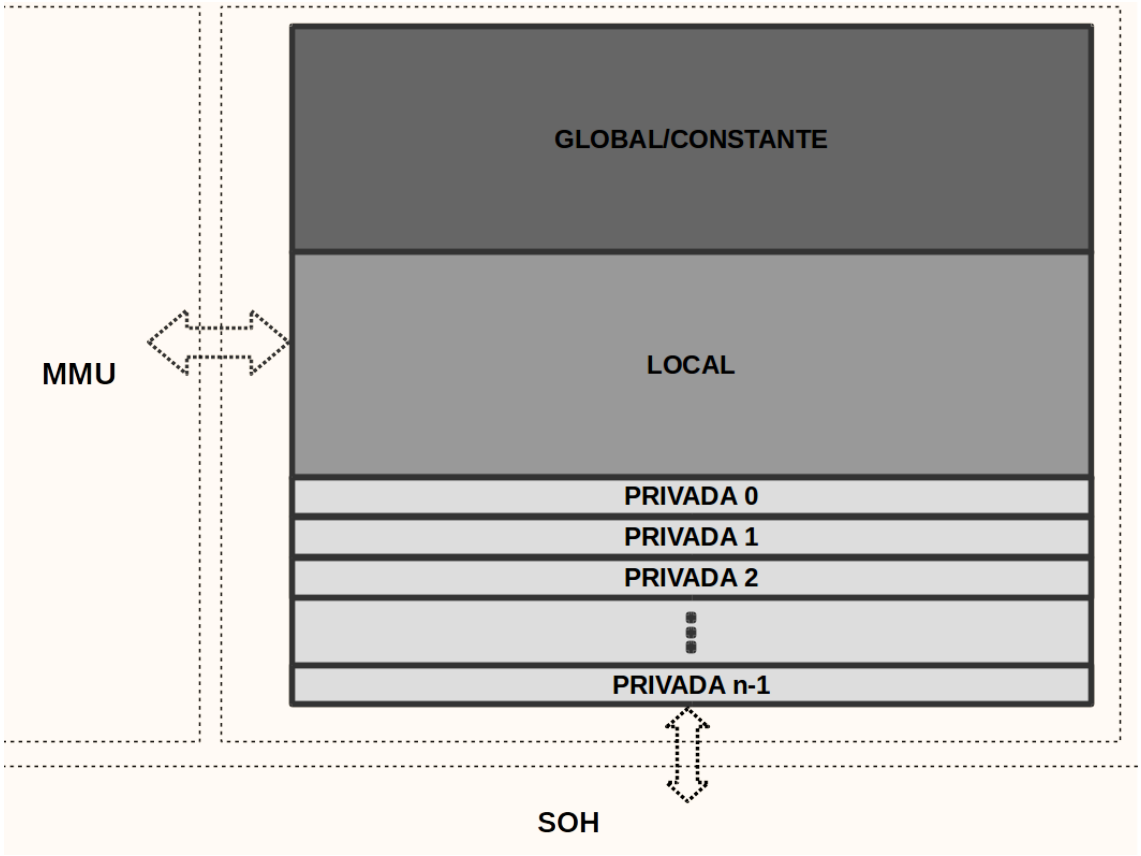

Figura 27 – Divisão da memória interna de um nó escravo; n representa a quantidade de *work-items* no *work-group*.

Tabela 3 – Detalhamento da tabela TLB.

tag	**Detalhes**
VPN	do inglês *Virtal Page Number*, armazena o número da página virtual de entrada
PFN	do inglês *Physical Frame Number*, armazena o número do frame da memória global
VALID	Bit de validação, informa se a entrada é válida
PROT	Bit de proteção, determina a permissão para escrita

ponente MI. A Figura 29 mostra uma representação da máquina de estados implementada pelo componente TU e possui três estados:

- TU_WAIT - aguarda por um acesso à memória. Se acesso for uma escrita na memória global ou leitura de uma página que não esteja em cache, vai para TU_REQUEST. Em qualquer outro acesso (memória global em cache, endereço à memória privada ou local), vai para o estado TU_ACCESS.

- TU_REQUEST - Manda uma requisição de leitura ou escrita para o MI e aguarda até a permissão de acesso para seguir para TU_ACCESS.

- TU_ACCESS - Efetiva o acesso e volta ao estado *TU_WAIT*.

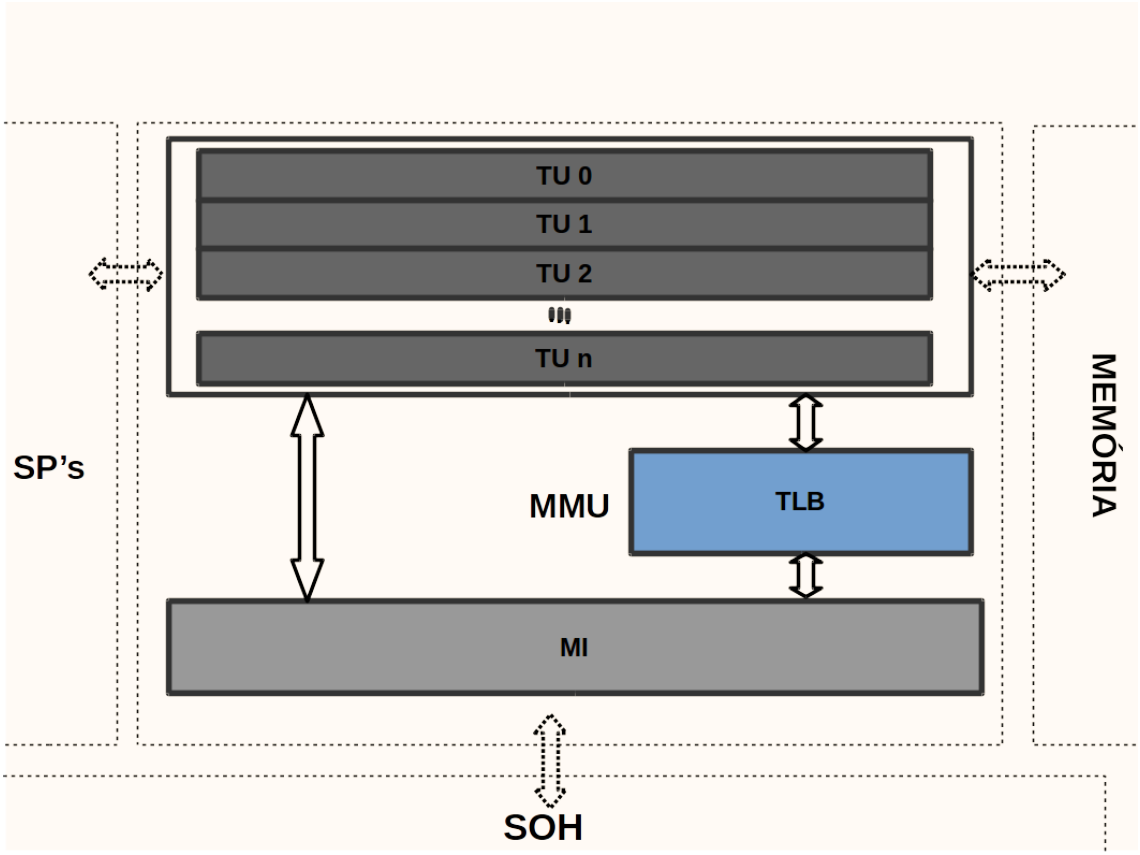

Figura 28 – Diagrama de blocos do componente MMU.

O MI é responsável por requisitar ao SOH uma leitura de página ou escrita na memória da rede. A Figura 30 mostra a máquina de estados implementada pelo componente, com quatro estados:

- MI_WAIT - aguarda por uma requisição de leitura ou escrita de algum TU para seguir para os estados MI_READ ou MI_WRITE, respectivamente;
- MI_READ - aciona o SOH requisitando uma leitura de página, informando os números das páginas das memórias da rede e interna. Aguarda até que a leitura seja efetivada para ir para o estado *MI_ACCESS*.
- MI_WRITE - aciona o SOH requisitando uma escrita informando o dado a ser escrito e o endereço da memória da rede. Vai de imediato ao estado *MI_ACCESS*.
- MI_ACCESS - libera o TU requerente para realizar o acesso à memória.

4.3 Modo de Funcionamento

A fim de concluir o entendimento da arquitetura apresentada, esta seção mostra um pequeno exemplo de funcionamento de todos os componentes operacionais apresentados nas

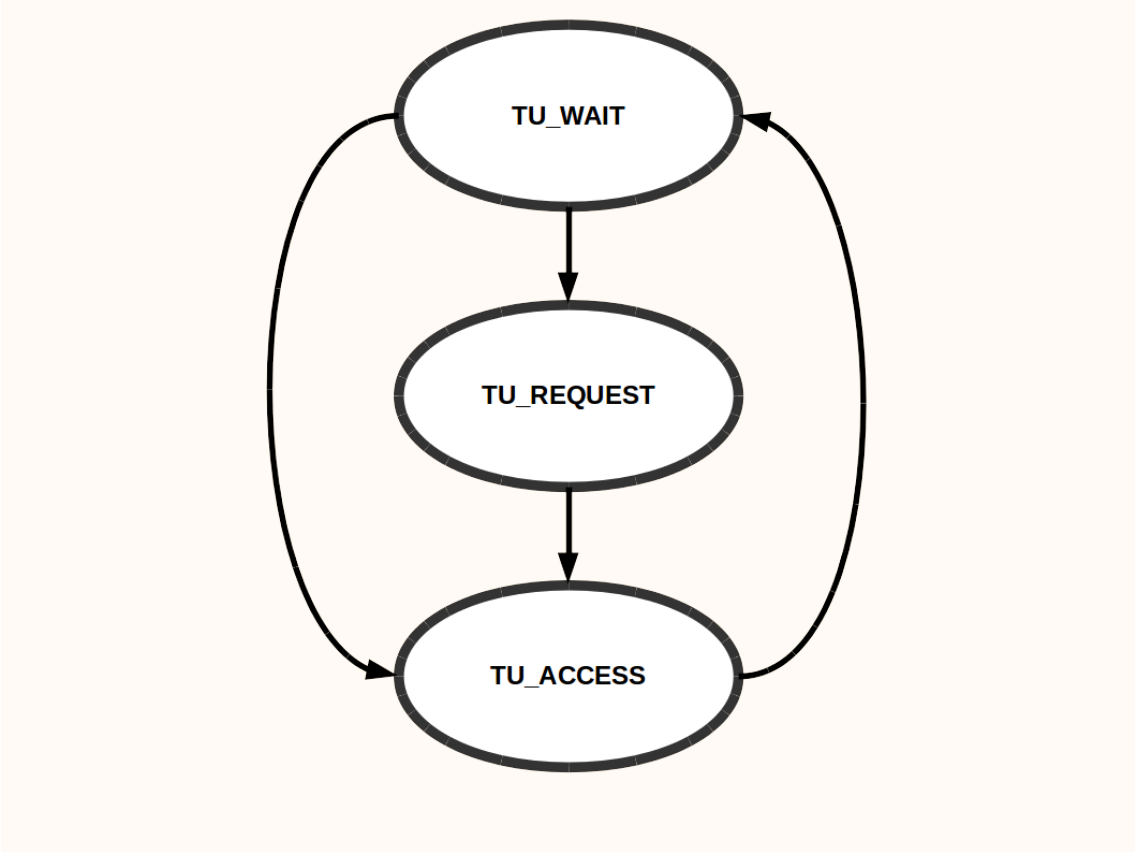

Figura 29 – Máquina de estados implementada pelo componente TU.

Seções 4.1 e 4.2. A aplicação exemplo foi configurada contendo seis *work-items* agrupados em um único *work-group*, todos no eixo x. O código *kernel*, mostrado na Figura 31, calcula o quadrado dos elementos do vetor a e armazena no vetor b. Os *work-items* sincronizam a finalização utilizando uma barreira. A Figura 32 mostra os tempos de execução dos componentes HI (*Host Interface*), MIN (*Master Input NoC*) e MON (*Master Output NoC*) do nó mestre; os componentes Interface, gerenciador e MI (*MMU Interface*) do nó escravo de coordenadas $x = 0$ *e* $y = 0$, assim como os componentes SI (*Slave Interface*), SON (*Slave Output NoC*), SIN (*Slave Input NoC*) e TU (*Translator Unit*) do SP0 (*Slave Processor 0*) do mesmo nó.

A Figura 32 mostra resultado da simulação. A execução inicia no nó mestre, com o *Master Processor* (MP) executando o código da aplicação *host*. Durante toda a configuração inicial executada pelo MP, todos os componentes permanecem em estado de espera indicado pela caixa ESPERA. A primeira interação ocorre na chamada da função *clenqueuewritebuffer*, onde o MP requisita ao componente HI que realize uma transferência de dados da memória de dados do MP para a memória da rede. Este ponto é mostrado pela primeira caixa TRANSFERENCIA na Figura 32.

Em seguida, o MP executa a função *clEnqueueNDRangeKernel* informando ao HI o ponteiro para início do kernel (KERNEL), configuração da *grid* de execução (NDRANGE)

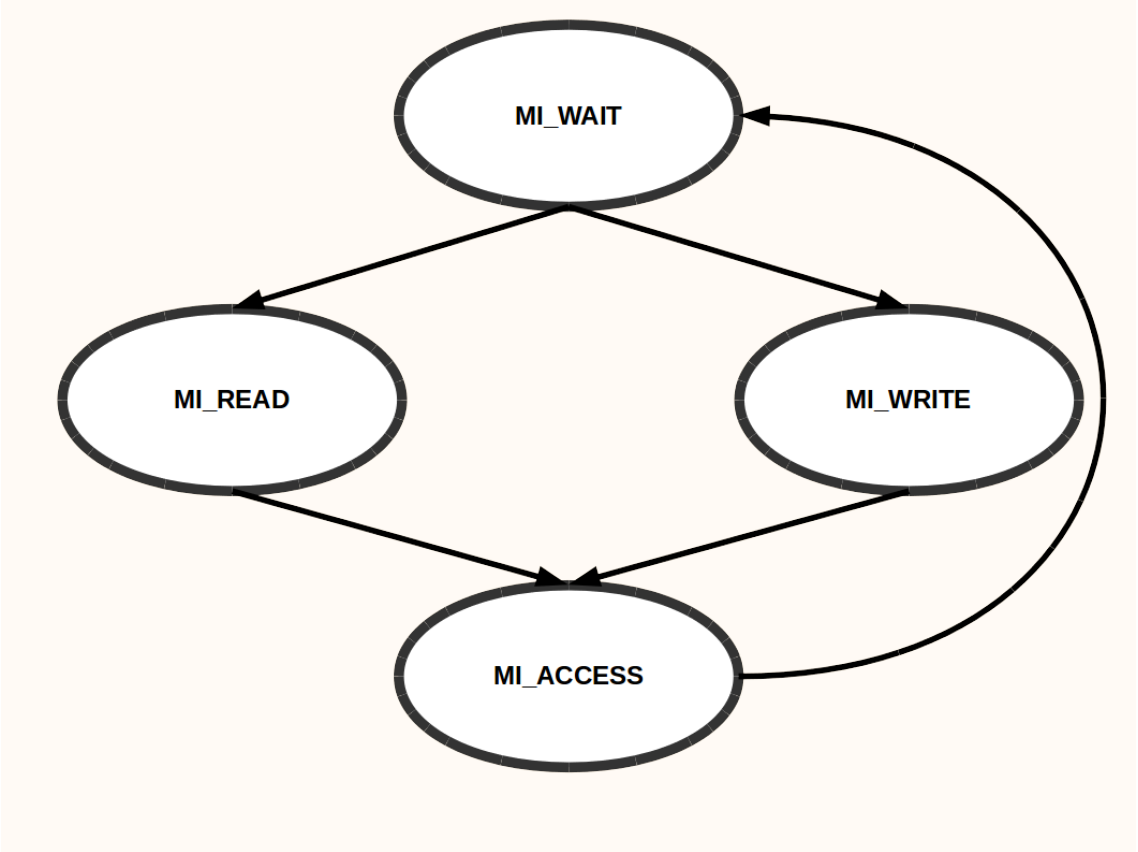

Figura 30 – Máquina de estados implementada pelo componente MI.

e solicitando o início da execução do programa *kernel* (INICIO). Neste ponto, a componente HI aciona o componente MON para que este inicie a transferência do código *kernel* e da configuração da *grid* para o nó escravo (INICIALIZAÇÃO WORK-GROUP). A componente SIN, por sua vez, começa a receber essas informações e, ao término do recebimento, aciona o componente interface para que este inicie a execução dos *work-items* nos SPs (INICIA WORK-ITEMS).

O componente interface requisita ao componente gerenciador os identificadores de *work-items* para serem executados. O gerenciador, por sua vez, dispara a execução dos seis *work-items* da aplicação. Logo no início da execução no SP0, o componente TU acusa um miss na memória global, gerando uma requisição ao componente MI (LEITURA MEMÓRIA GLOBAL). Este aciona o SON para que uma requisição seja enviada ao nó mestre.

O componente MIN do nó mestre recebe a requisição e a encaminha ao componente MON que, por sua vez, devolve a página solicitada. O componente SIN recebe a página e a armazena na memória interna liberando o componente TU para realizar o acesso.

No decorrer do restante da execução, ocorrem as chamadas das funções *get_global_id*, *get_group_id*, *get_local_size* e da função *barrier* que são atendidas pela componente SI(IDS/TAMANHO)(BARREIRAS). Durante a função *barrier* o componente SI aciona o

componente interface informando da execução da função. Este, aguarda até que todos os *work-items* alcancem a barreira, para liberar novamente a execução.

No final da execução dos *work-items* ocorrem as escritas na memória global (ESCRITAS). Ao final da execução, o MP requisita a transferência do vetor resultante da memória da rede para a memória do host, como mostrado na segunda caixa TRANSFERENCIA da Figura 32.

```
__kernel void square(__global const int* a,
                     __global const int* b)
{

    int idi = get_global_id(0);
    int idg = get_group_id(0);
    int size = get_local_size();
    int x = a[idg*size+idi] * a[idg*size+idi];

    barrier();

    b[idg*size + idi] =   x;
};
```

Figura 31 – Programa kernel executado na plataforma METAL.

4.4 Considerações Finais

Neste capítulo foram apresentadas as principais características da plataforma METAL. Foi mostrada a divisão da plataforma em nó mestre e nós escravos, e como o nó mestre coordena a execução nos nós escravos. Mostrou-se também a dinâmica de funcionamento dentro de cada nó. O objetivo deste capítulo é fornecer um entendimento básico da plataforma para que o leitor compreenda os resultados obtidos e mostrados no próximo capítulo.

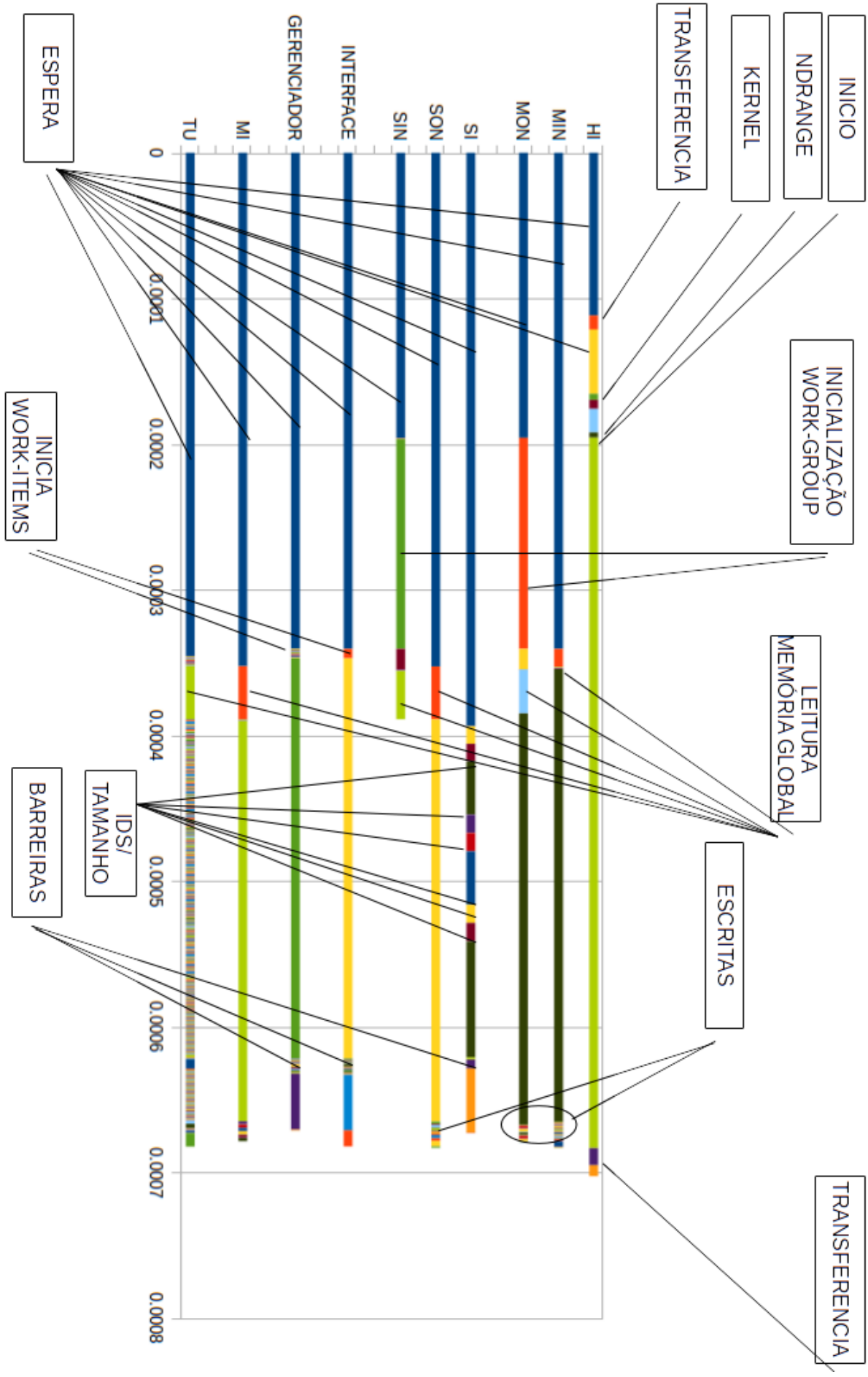

Figura 32 – Exemplo de funcionamento de um programa na plataforma METAL.

5 Resultados e Discussão

A plataforma METAL foi implementada utilizando uma biblioteca para criação de simuladores de hardware chamada SystemC (GROTKER, 2002). Esta biblioteca, implementada em C++ (STROUSTRUP, 2000), fornece um conjunto de macros, classes e uma interface de simulação, dirigida a eventos, que permitem ao projetista simular processos concorrentes usando a sintaxe C++ convencional. O simulador da plataforma METAL pode ser configurado quanto a quantidade de nós existentes na rede, a quantidade de *Slave Processors* (SPs) nos nós escravos, o tamanho das memórias e o tamanho de uma página da memória.

A fim de validar e avaliar o desempenho da plataforma METAL, foram implementados os algoritmos Dijkstra, multiplicação de matrizes, genético simples para a resolução do problema das n-rainhas, jantar dos filósofos e detecção de borda aplicando o filtro Laplaciano. Os algoritmos foram escritos na linguagem C, utilizando o *framework* OpenCL, e compilados utilizando um compilador, chamado CLEM (*open**CL** compil**E**r **M**etal*), que foi desenvolvido pelo mesmo grupo de pesquisa que o autor desta dissertação participa. A Tabela 4 mostra a configuração que foi utilizada para a plataforma durante a execução dos algoritmos listados.

A avaliação dos algoritmos foi realizada utilizando duas métricas: percentual médio de processamento e percentual médio de espera. O percentual médio de processamento é a média do percentual de processamento de todos os *Slaves Processors* que foram utilizados na execução do algoritmo. O cálculo é feito com base no tempo total de execução e é expressado pela seguinte fórmula:

$$PMP = (\sum TP_i/qnt) * 100/Ttotal$$

- **PMP**: Percentual médio de processamento;
- **TP$_i$**: Tempo de processamento no SP i, com i variando de 0 a 63;
- **qnt**: Quantidade de processadores utilizados na aplicação;
- **Ttotal**: Tempo total de execução da aplicação.

O percentual médio de espera é a média do percentual de tempo que todos os SPs passam esperando. A espera ocorre quando o SP acessa a rede em chip, em operações de leitura, escrita na memória da rede, operações com mutex, ou o tempo inicial até que a tarefa chegue ao nó. O cálculo do percentual médio de espera também é calculado com base no tempo total de execução e é expressado pela seguinte fórmula:

$$PME = (\sum TE_i/qnt) * 100/Ttotal$$

- **PME**: Percentual médio de espera;
- **TE_i**: Tempo de espera no SP i, com i variando de 0 a 63;
- **qnt**: Quantidade de processadores utilizados na aplicação;
- **Ttotal**: Tempo total de execução da aplicação.

As próximas seções mostram os resultados obtidos com a execução de cada algoritmo.

Tabela 4 – Configuração da Plataforma METAL

Quantidade de *Slaves Processors* por nó escravo	8
Quantidade de nós na rede	3X3
Tamanho da memória de instruções do mestre	512KB
Tamanho memória de dados do mestre	512KB
Tamanho da memória da rede	512KB
Tamanho da memória de instruções dos escravos	512KB
Tamanho da memória interna	2MB
Tamanho da memória do escalonador	384KB
Tamanho da *mutex memory*	100KB
Tamanho da FIFO	100KB
Tamanho da página de memória	256B

5.1 Dijkstra

O algoritmo Dijkstra é utilizado para encontrar o menor caminho entre os nós de um grafo. Este é um dos algoritmos propostos no *benchmark* ParMiBench (IQBAL; LIANG; GRAHN, 2010). A estratégia de paralelização do código OpenCL utilizado para implementar este algoritmo, atribui a cada *work-item* a responsabilidade de calcular o menor caminho de um dos nós do grafo até todos os outros. A quantidade total de *work-items* na aplicação, que depende do tamanho do grafo, foi dividida em 8 *work-groups* de modo a utilizar os oito nós escravos da rede.

A Figura 33 mostra o percentual dos tempos médios de processamento e espera, dos 64 SPs, em relação ao tempo total de execução do algoritmo para grafos completos contendo 32, 64 e 128 nós. Observa-se que, para as três entradas, o tempo médio que um SP gasta realizando computação útil corresponde a aproximadamente 86% do tempo total de execução do algoritmo, enquanto que o tempo médio que um SP gasta acessando a rede, para realizar operações de leitura e escrita na memória da rede, corresponde a aproximadamente 10% do tempo total de execução. Os 4% restantes correspondem ao

intervalo de tempo em que o primeiro SP termina e o momento em que todos os resultados são armazenados na memória da rede, em que um SP não está esperando nem computando.

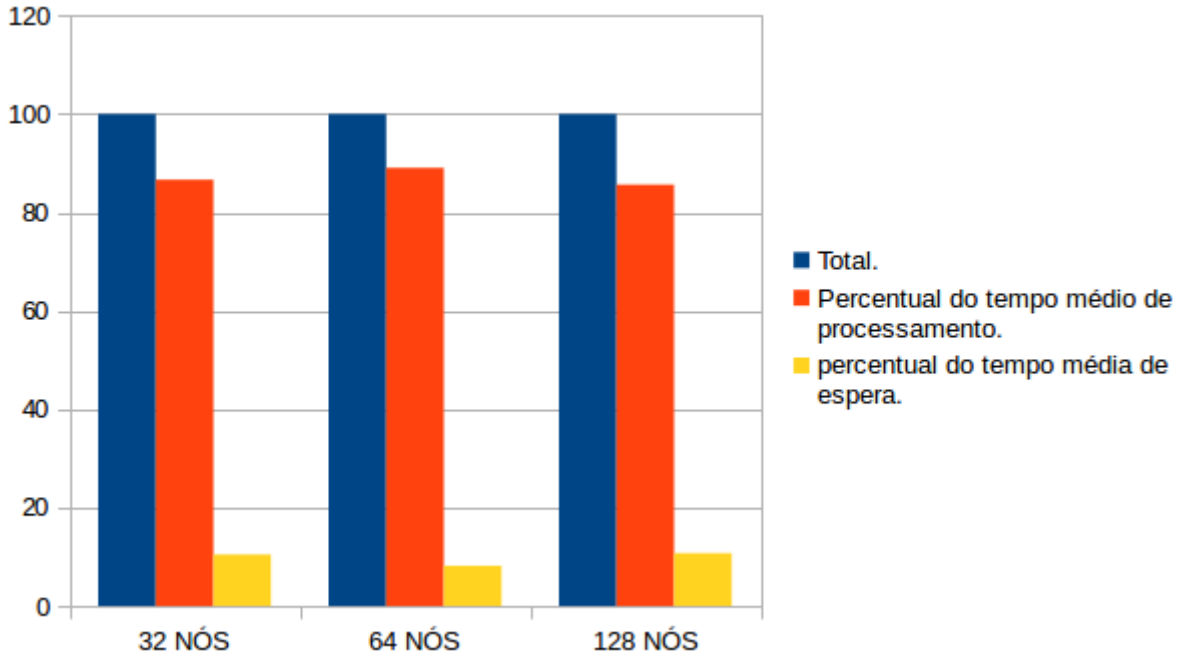

Figura 33 – Percentual dos tempos médios de processamento e espera do algoritmo Disjkstra.

O fato de os SPs passarem em média 86% do tempo total da aplicação realizando processamento, significa que aplicação é considerada *CPU-bound*, ou seja, o tempo total de execução é mais influenciado pelo tempo de processamento do que pelo tempo gasto realizando outras atividades, acessando a memória por exemplo.

A Figura 34 mostra os tempos reais (em ciclos de clock) de processamento e espera para as execuções mostradas na Figura 33. Observa-se que, mesmo com o aumento do tamanho do grafo de entrada, as proporções dos tempos médios de espera e processamento de um SP são mantidas. Isto significa que a taxa de uso da rede em chip para acessar a memória da rede também permanecerá constante mesmo com o crescimento do grafo de entrada.

5.2 Multiplicação de matrizes

Multiplicação de matrizes é uma operação básica utilizada por vários algorítimos nas mais diversas áreas, por exemplo, processamento de imagens, resolução de sistemas lineares, criptografia, etc. Por ser um algoritmo que possui paralelismo naturalmente implícito, é também amplamente utilizado na avaliação de desempenho de arquiteturas paralelas. Para a realização dos testes na plataforma METAL, o código OpenCL da multiplicação de matrizes foi implementado de forma que cada *work-item* ficasse responsável por calcular

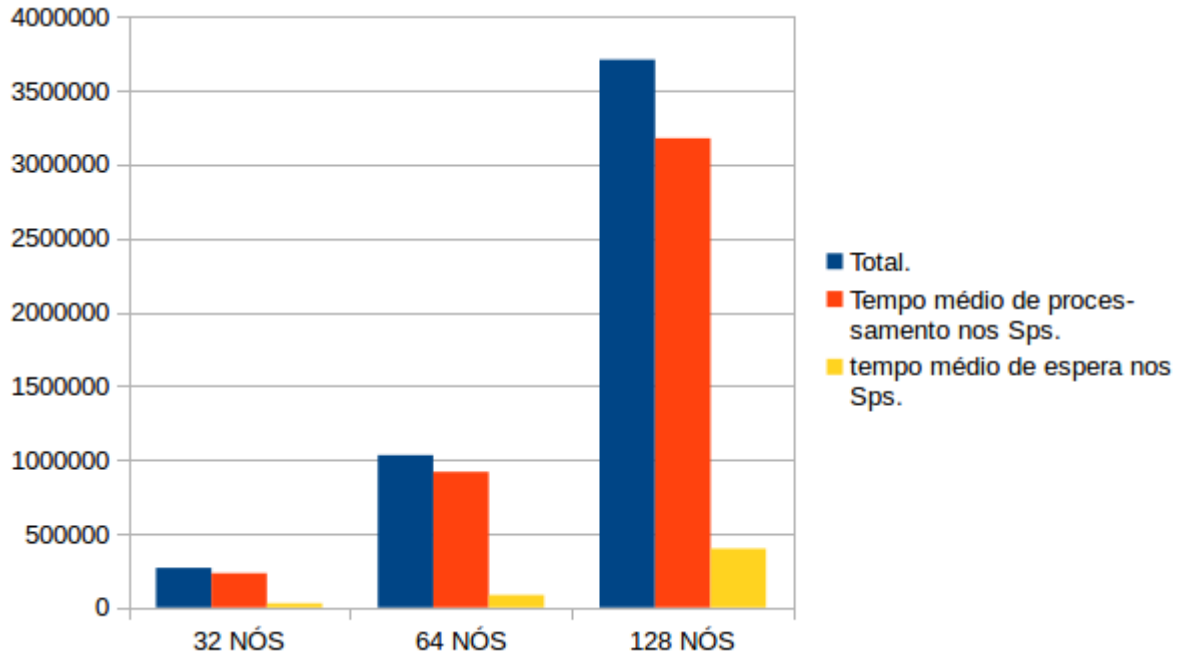

Figura 34 – Tempos de execução do algoritmo Dijkstra.

um elemento da matriz resultante. A quantidade total de *work-items*, que depende das dimensões das matrizes inicias, foi dividida por 64 *work-groups* de modo a maximizar o uso dos nós escravos da rede.

A Figura 35 mostra os percentuais dos tempos médios de processamento e espera para execução do algorítimo com matrizes de entradas com dimensões 32x32, 64x64 e 96x96. Observa-se que, com o aumento dos tamanhos das matrizes de entrada, o percentual do tempo médio de processamento aumenta, enquanto que o percentual de tempo médio de espera diminui. Isso se deve ao fato de que a quantidade de processamento realizado por cada SP aumenta a uma taxa maior do que o aumento do tempo médio de espera. Isto significa que, quanto maior as matrizes de entrada maior será a razão *tempo em processamento/tempo em espera*. Na Figura 36, que mostra os tempos reais (em ciclos) de processamento e espera da execução do algoritmo, observa-se que os tempos de espera aumentam junto com o tamanho das matrizes de entrada, porém a uma taxa menor que o aumento do tempo médio de processamento.

5.3 Detecção de bordas aplicando filtro Laplaciano

Algoritmos de detecção de bordas são técnicas utilizadas em processamento de imagens e visão computacional, para identificação de contornos em imagens (CANNY, 1986). O filtro Laplaciano é um filtro espacial que pode ser usado para detecção de bordas (PARIS; HASINOFF; KAUTZ, 2011). O código OpenCL para detecção de bordas utilizando o filtro Laplaciano foi implementado de forma que cada *work-item* da aplicação

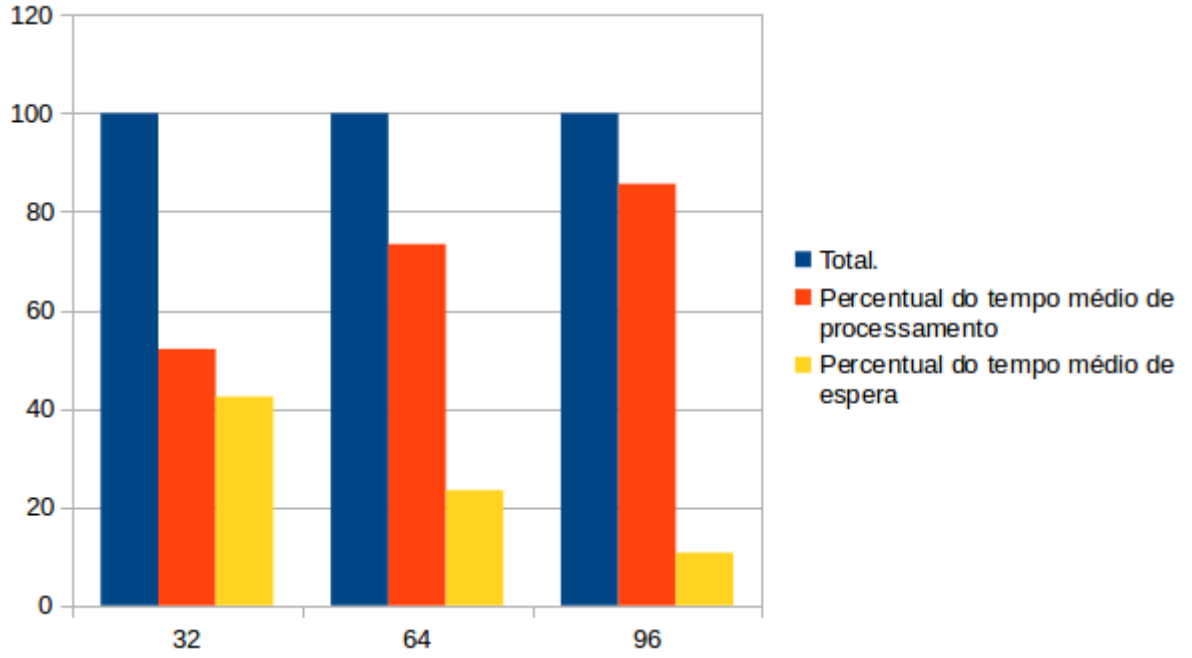

Figura 35 – Percentual dos tempos médios de processamento e espera do algoritmo multiplicação de matrizes.

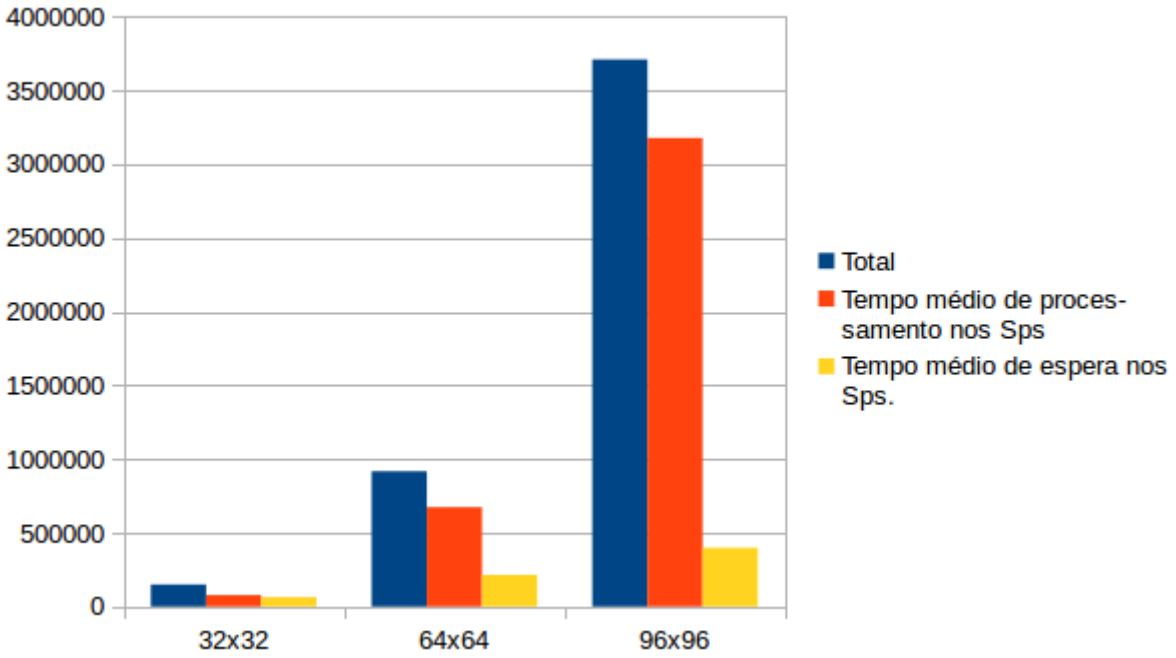

Figura 36 – Tempos de execução da Multiplicação de Matrizes.

calculasse um pixel da imagem resultante. A quantidade total de *work-item*, que depende do tamanho da imagem original, foi dividida entre 64 *work-groups*, sendo 8 no eixo x e 8 no eixo y.

A Figura 37 mostra os percentuais dos tempos médios de processamento e espera dos SPs em relação ao tempo total de execução do algoritmo tendo como entradas imagens

com resoluções 32x32, 64x64 e 128x128. Os resultados mostram que, com imagens de resolução até 64x64, o tempo que um SP passa esperando é maior que o tempo que ele passa processando. Isto significa que, com imagens de entrada com resolução até 64x64, a aplicação é considerada *memory-bound*, ou seja, o tempo gasto realizando operações de acesso à memória é maior que o tempo gasto realizando computação útil. Esse comportamento é explicado devido a forma como os *work-items* acessam a memória. Para calcular o pixel resultante, cada work-item só precisa acessar os pixels vizinhos ao qual ele é responsável, dessa forma, a quantidade de busca de dados na memória da rede será reduzida, devido ao princípio da localidade dos dados (DENNING, 2005). Na Figura 38, que mostra os tempos reais (em ciclos) de processamento e espera, é possível observar também um grande aumento no tempo médio de processamento, isto devido a maior quantidade de pixels a serem calculados.

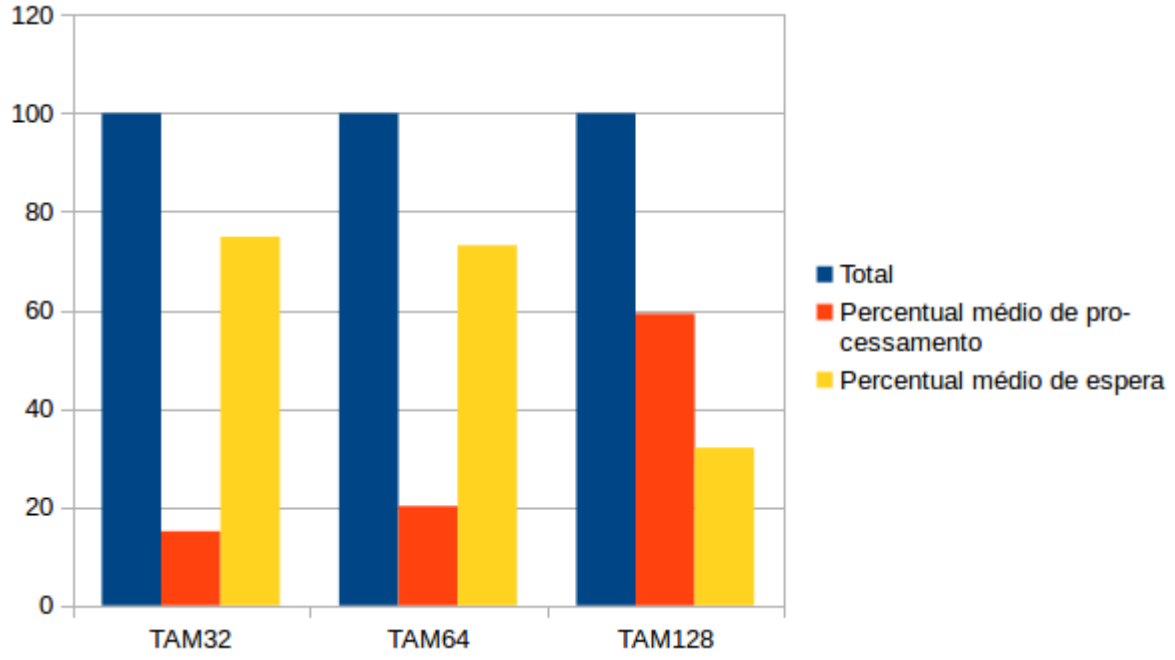

Figura 37 – Percentual dos tempos médios de processamento e espera do algoritmo detecção de bordas laplaciano.

5.4 Algoritmo Genético Simples para resolução problema das N-Rainhas

Algoritmos genéticos são heurísticas de busca inspirados na teoria da evolução dos seres vivos (MITCHELL, 1998). Sua aplicação para resolução do problema das N-Rainhas é uma prática clássica no ensino de algoritmos genéticos. A Figura 39 mostra os percentuais dos tempos de processamento e espera para que o algoritmo calcule uma geração com os tabuleiros de tamanhos 4x4, 8x8 e 16x16. Nesta figura é possível observar que, com

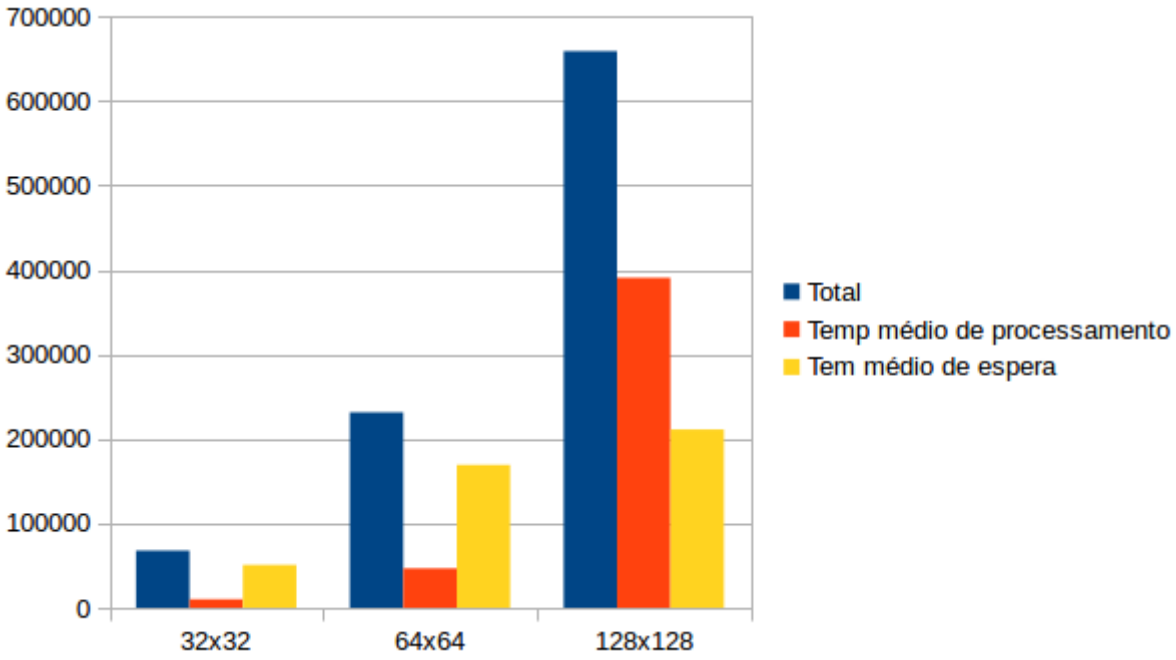

Figura 38 – Tempos de execução do algoritmo detecção de bordas laplaciano.

o aumento do tamanho do tabuleiro, o percentual do tempo de processamento aumenta, enquanto que, o percentual do tempo de espera diminui. Isto é explicado porque o algoritmo não gera requisições às páginas da memória da rede e tamanho do kernel é constante em relação ao tabuleiro inicial. Dessa forma, com o aumento do tamanho do tabuleiro inicial, aumenta a quantidade de processamento, mas não aumenta o tempo de espera.

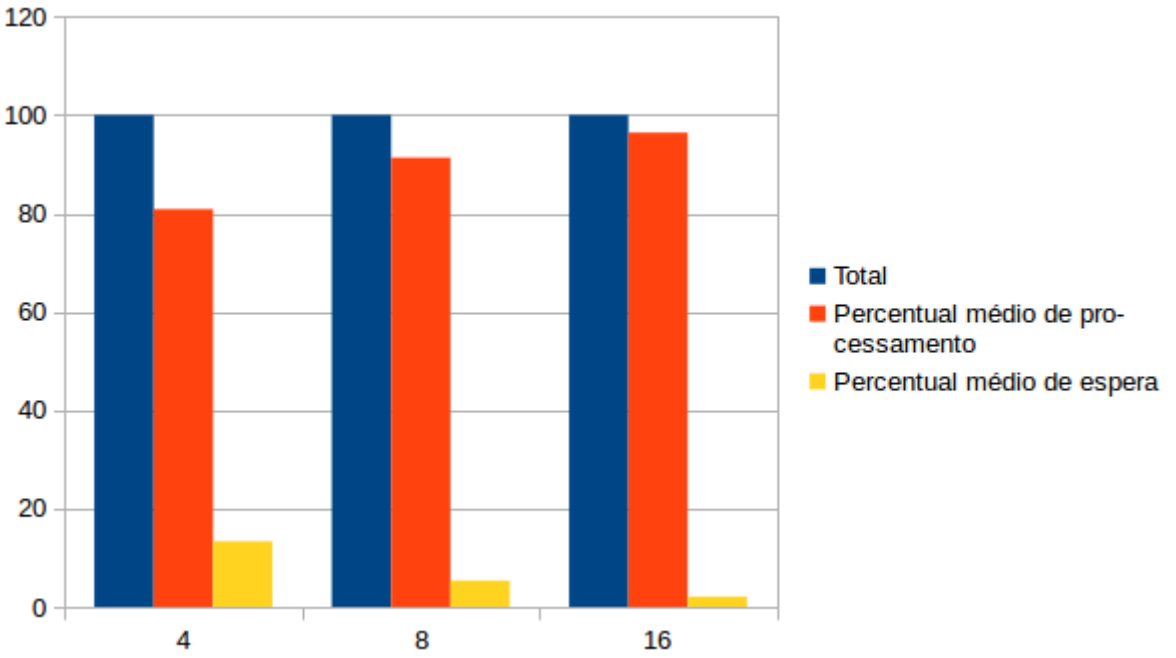

Figura 39 – Percentual dos tempos médios de processamento e espera do algoritmo genético.

Na Figura 40, que mostra os tempos reais (em ciclos) de processamento e espera para a mesma execução, observa-se que o tempo médio de processamento aumenta, enquanto o tempo de espera mantêm-se constante, fazendo com que o percentual do tempo de processamento cresça e o de espera diminua. Este tipo de comportamento mostrado pelo algoritmo genético, muito processamento e pouco acesso à memória da rede, é o mais adequado para a utilização da plataforma METAL, uma vez que não sobrecarrega o uso da rede.

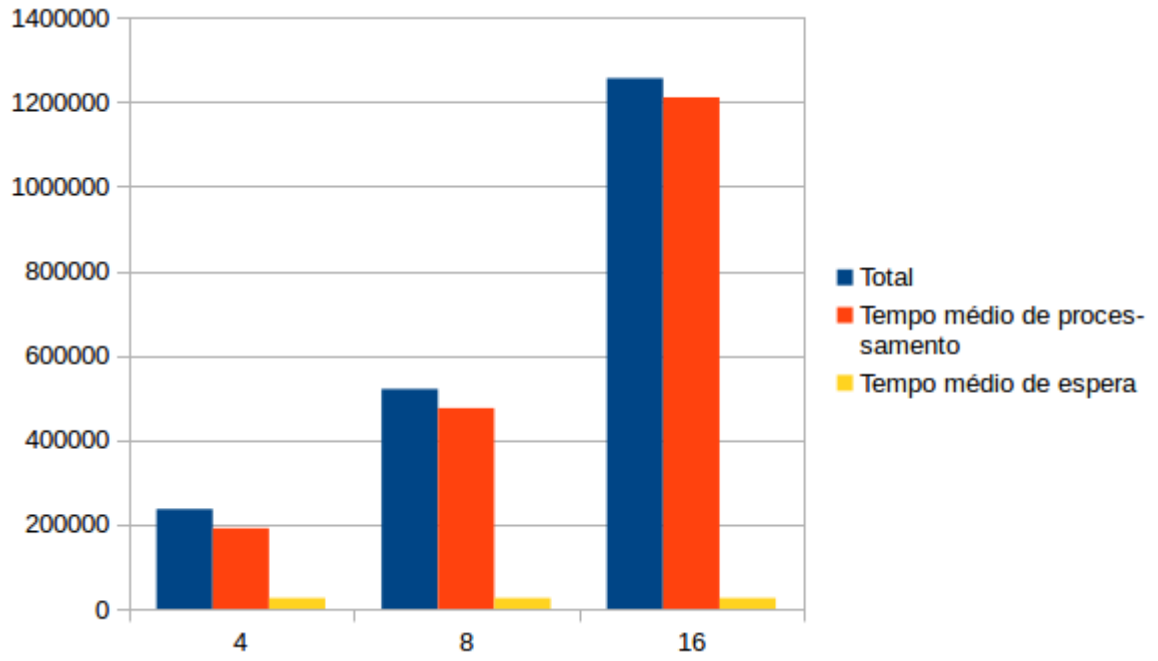

Figura 40 – Tempos de execução do algoritmo genético.

5.5 O problema do Jantar dos Filósofos

Em Ciência da Computação, o problema do Jantar dos filósofos é um exemplo clássico de sincronização de processos em programação concorrente. O problema se resume a cinco filósofos sentados ao redor de uma mesa redonda, cada qual fazendo exclusivamente duas atividades: comendo ou pensando. Enquanto está comendo, um filósofo não pode pensar, e vice-versa. Para realizar a atividade comer, um filósofo precisa de dois garfos, no entanto, só existem cinco garfos disponíveis na mesa, o que significa que os filósofos deverão compartilhar os garfos. Na implementação do problema do Jantar dos Filósofos utilizada para execução na plataforma METAL, o acesso aos garfos foi resolvido utilizando a técnica de exclusão mútua (mutex) para programação concorrente. Essa técnica evita que dois processos ou threads tenham acesso simultâneos a um recurso compartilhado, acesso esse denominado de seção crítica. A Figura 41 mostra os percentuais dos tempos médios de processamento e espera para execução do problema utilizando 8 mesas, cada mesa composta por 8, 16 e 32 filósofos disputando por 8, 16 e 32 garfos respectivamente.

Os resultados mostram um comportamento similar à execução do algoritmo genético para o problema das n-rainhas, também pelos mesmos motivos, o pouco acesso à memória da rede.

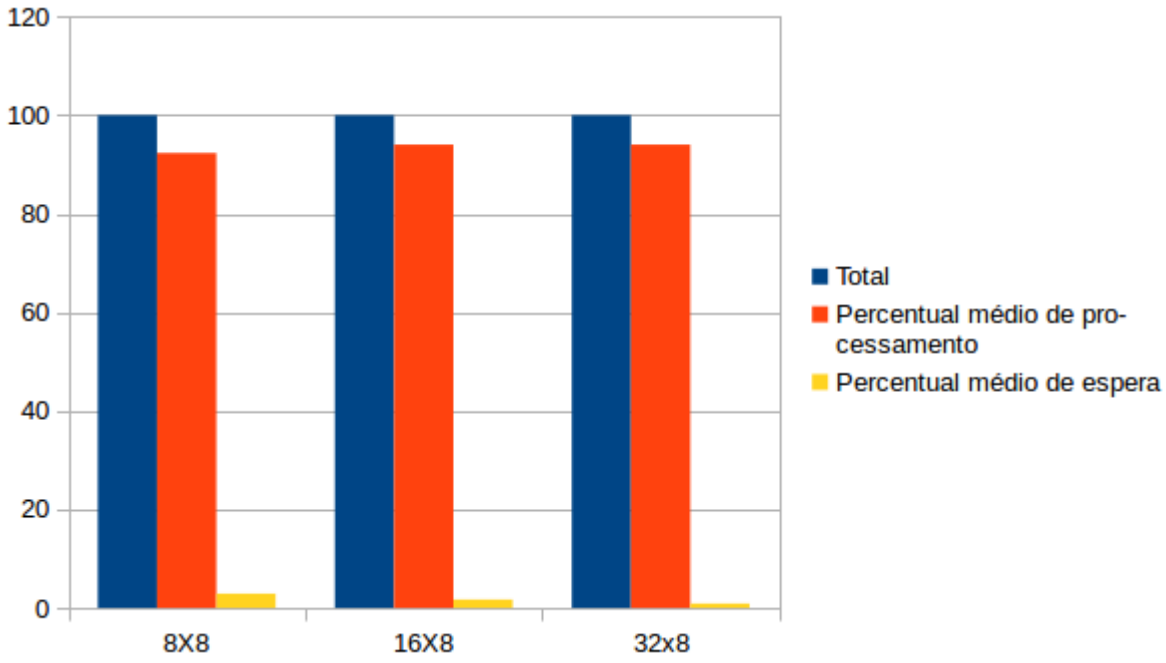

Figura 41 – Percentual dos tempos médios de processamento e espera do jantar dos filósofos.

A Figura 42 mostra os tempos de execução reais (em ciclos) para a execução do algoritmo jantar dos filósofos. Da mesma forma que o algoritmo genético, o tempo médio de processamento cresce, enquanto que o tempo de espera se mantém constante.

5.6 Uso do Escalonador em *Hardware*

Devido a escassez de arquiteturas com o propósito semelhante ao da plataforma METAL, realizar uma comparação justa com outras arquiteturas disponíveis na literatura mostrou-se ser uma tarefa inviável. A solução encontrada para pôr em perspectiva a utilização do escalonador em *hardware* foi realizar a comparação com a versão proposta por Nepomuceno et al. (2015).

A Figura 43 mostra a execução do algoritmo multiplicação de matrizes utilizando duas matrizes 16x16. A aplicação foi organizada com 16 *work-items* e 1 *work-group*, onde cada *work-item* é responsável por computar uma linha da matriz resultante. A figura mostra os tempos de *idle*, processamento, troca e processamento novamente dos oito processadores utilizados. Na versão sem escalonador é possível observar o tempo de troca de contexto entre os dois tempos de processamento. Já na versão com escalonador esse tempo é imperceptível. Nessa aplicação, o tempo de troca de contexto não tem muita

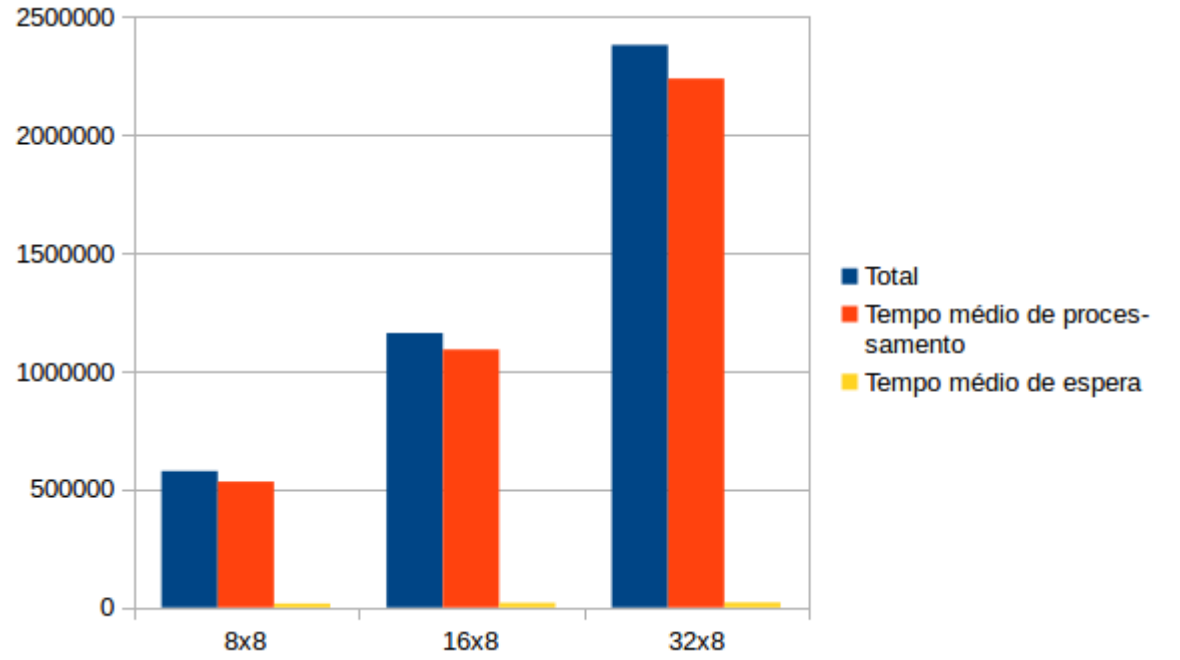

Figura 42 – Tempos de execução do jantar dos filósofos.

interferência no tempo total devido à simplicidade da mesma. No entanto, em aplicações com intensa troca de contexto esse tempo teria impacto maior no tempo total da aplicação.

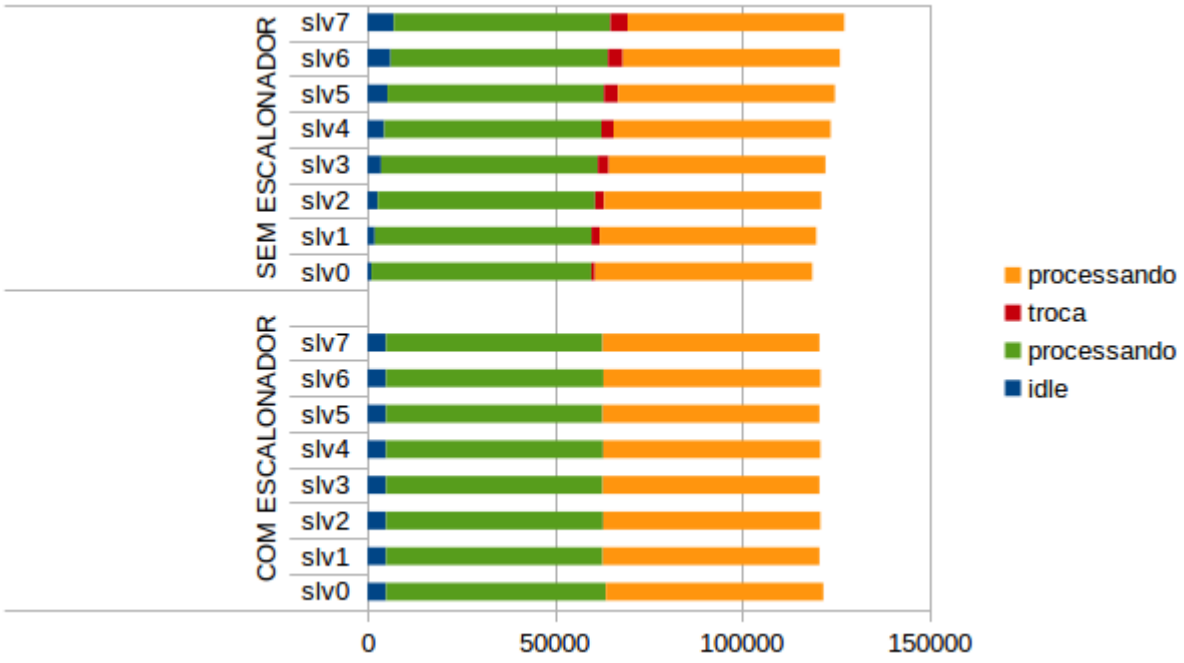

Figura 43 – Tempo de escalonamento.

5.7 Considerações Finais

Neste capítulo de resultados e discussão foi mostrado o desempenho da arquitetura ao executar cinco algoritmos: Dijkstra, multiplicação de matrizes, genético simples para a resolução do problema nas n-rainhas, jantar dos filósofos e detecção de borda aplicando o filtro Laplaciano.

O desempenho foi avaliado de acordo com duas métricas: percentual médio de processamento e percentual médio de espera. Resumidamente, os resultados mostraram que o desempenho da arquitetura aumenta a medida que a quantidade de dados a serem processados também aumenta. Característica típica de arquiteturas que possuem muitos núcleos de processamento.

Na próxima seção, serão apresentadas as conclusões finais e os trabalhos futuros.

Conclusões e Trabalhos Futuros

Conclusões

As arquiteturas *manycore*, por possuírem dezenas de núcleos de processamento, levam ao extremo a ideia de computação paralela. No entanto, o uso eficiente de todo esse potencial ainda é um privilégio de poucas aplicações. Em geral, aplicações que apresentam, de forma nativa, massivo paralelismo de dados são mais adaptadas para execução em arquiteturas *manycore*. Já aplicações que são naturalmente sequenciais ainda carecem que sejam disponibilizados recursos, tanto em nível de *software* quanto em nível de *hardware*, para que estas possam utilizar com eficiência as funcionalidades oferecidas pelas arquiteturas *manycore*.

Em nível de *software*, o suporte oferecido ao desenvolvimento de aplicações geralmente é dado em forma de *frameworks*, bibliotecas e compiladores que ajudam a extrair paralelismo de aplicações sequenciais. Em nível de hardware, o suporte é oferecido através da implementação de componentes que auxiliam esses frameworks, bibliotecas e compiladores a executarem de maneira mais eficiente.

A plataforma METAL, em conjunto com o compilador CLEM (*open**CL** compil**E**r **M**etal*), tem como intuito justamente viabilizar a coerência desde o desenvolvimento do software, até sua execução na arquitetura. Mais especificamente, a plataforma METAL disponibiliza um escalonador em hardware, uma unidade de gerenciamento de acesso memória compartilhada (mutex) e um modelo de memória adaptado a execução SIMD, como uma forma de dar suporte às aplicações geradas pelo compilador CLEM. O compilador, por sua vez, implementa técnicas de otimização, e funções específicas da arquitetura, a fim de gerar um código objeto que seja capaz de utilizar todos os recursos oferecidos pela arquitetura.

Os resultados apresentados no Capítulo 5 mostram o desempenho da arquitetura ao executar algumas aplicações tradicionais no que diz respeito a avaliar arquiteturas paralelas. A principal característica observada com os resultados, foi o fato de o desempenho da arquitetura aumentar a medida que a quantidade de dados a serem processados também aumenta. Esse comportamento é típico das arquiteturas *manycore*, uma vez que elas oferecem centenas de núcleos de processamento.

Portanto, tendo em vista os resultados obtidos na execução de aplicações SIMD utilizando OpenCL, e na realização de programação paralela por meio de variável compartilhada utilizando a técnica de exclusão mútua, conclui-se que o objetivo inicial, que era desenvolver uma plataforma *manycore* com recursos em *hardware* para executar aplicações

de propósito geral e SIMD, foi alcançado.

Trabalhos Futuros

Como trabalhos futuros, sugere-se a ampliação das funcionalidades do nó mestre, para que este seja capaz de tomar a decisão, baseado em alguns dados fornecidos pela própria arquitetura, de como distribuir aplicações na plataforma.

Outra alteração seria tornar o escalonador mais flexível, para ele pudesse escalonar não só *work-items* de uma aplicação OpenCL e sim um conceito mais abstrato de hierarquia de processos e *threads* configurado pelo nó mestre. Estas duas alterações abririam margem a permitir que a plataforma fosse programada utilizando os mais diversos modelos de programação. Seria possível, por exemplo, o nó mestre mandar um *work-group* de uma aplicação OpenCL para ser executada em um determinado nó, e um *block* de *threads* CUDA em outro.

Outra sugestão seria equipar os nós escravos com recursos para que eles pudessem operar de modos distintos, habilitando e desabilitando funcionalidades dependendo da aplicação que ele fosse executar. Por exemplo, se a um determinado nó fosse atribuída uma tarefa sequencial, este poderia desabilitar os SPs que não seriam utilizados, para que dessa forma, fosse possível reduzir o consumo de energia.

Em termos de programabilidade, sugere-se a implementação dos recursos necessários para dar suporte a um sistema operacional. Esta ampliação seria importante porque aumentariam as possibilidades de desenvolvimento de aplicações para a plataforma, aumentando assim, sua visibilidade.

Referências

ALVAREZ, P. L.; YAMAGIWA, S. Invitation to opencl. In: *2011 Second International Conference on Networking and Computing (ICNC)*. [S.l.: s.n.], 2011. p. 8–16. Citado na página 23.

BARTH, T. J. Simplified discontinuous galerkin methods for systems of conservation laws with convex extension. In: _____. *Discontinuous Galerkin Methods: Theory, Computation and Applications*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000. p. 63–75. ISBN 978-3-642-59721-3. Disponível em: <http://dx.doi.org/10.1007/978-3-642-59721-3_3>. Citado 2 vezes nas páginas 31 e 32.

BAŞAR, Y.; ITSKOV, M. Constitutive model and finite element formulation for large strain elasto-plastic analysis of shells. *Computational Mechanics*, v. 23, n. 5, p. 466–481, 1999. ISSN 1432-0924. Disponível em: <http://dx.doi.org/10.1007/s004660050426>. Citado na página 31.

BLAKE, G.; DRESLINSKI, R. G.; MUDGE, T. A survey of multicore processors. *IEEE Signal Processing Magazine*, v. 26, n. 6, p. 26–37, November 2009. ISSN 1053-5888. Citado 5 vezes nas páginas 10, 11, 12, 13 e 15.

BRANOVER, A.; FOLEY, D.; STEINMAN, M. Amd fusion apu: Llano. *IEEE Micro*, v. 32, n. 2, p. 28–37, March 2012. ISSN 0272-1732. Citado na página 3.

BUCK, I.; FOLEY, T.; HORN, D.; SUGERMAN, J.; FATAHALIAN, K.; HOUSTON, M.; HANRAHAN, P. Brook for gpus: Stream computing on graphics hardware. In: *ACM SIGGRAPH 2004 Papers*. New York, NY, USA: ACM, 2004. (SIGGRAPH '04), p. 777–786. Disponível em: <http://doi.acm.org/10.1145/1186562.1015800>. Citado na página 17.

BUTENHOF, D. R. *Programming with POSIX Threads*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997. ISBN 0-201-63392-2. Citado na página 19.

BöHME, D. *Characterizing Load and Communication Imbalance in Parallel Applications*. 1. ed. [S.l.]: Forshungszentrum Jülich GmbH, 2014. v. 23. ISBN 9783893369409. Citado na página 8.

CANNY, J. A computational approach to edge detection. *IEEE Trans. Pattern Anal. Mach. Intell.*, IEEE Computer Society, Washington, DC, USA, v. 8, n. 6, p. 679–698, jun. 1986. ISSN 0162-8828. Disponível em: <http://dx.doi.org/10.1109/TPAMI.1986.4767851>. Citado na página 62.

CHE, S.; BOYER, M.; MENG, J.; TARJAN, D.; SHEAFFER, J. W.; LEE, S. H.; SKADRON, K. Rodinia: A benchmark suite for heterogeneous computing. In: *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. [S.l.: s.n.], 2009. p. 44–54. Citado na página 37.

CHEN, K. C.; CHEN, C. H. An opencl runtime system for a heterogeneous many-core virtual platform. In: *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*. [S.l.: s.n.], 2014. p. 2197–2200. ISSN 0271-4302. Citado na página 38.

CHEN, T.; RAGHAVAN, R.; DALE, J. N.; IWATA, E. Cell broadband engine architecture and its first implementation: A performance view. *IBM J. Res. Dev.*, IBM Corp., Riverton, NJ, USA, v. 51, n. 5, p. 559–572, set. 2007. ISSN 0018-8646. Disponível em: <http://dx.doi.org/10.1147/rd.515.0559>. Citado na página 17.

CHU, S. L.; HSIAO, C. C. Opencl: Make ubiquitous supercomputing possible. In: *High Performance Computing and Communications (HPCC), 2010 12th IEEE International Conference on*. [S.l.: s.n.], 2010. p. 556–561. Citado na página 34.

COOK, S. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. 1st. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013. ISBN 9780124159334, 9780124159884. Citado na página 17.

CULLER, D. E.; GUPTA, A.; SINGH, J. P. *Parallel Computer Architecture: A Hardware/Software Approach*. 1st. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997. ISBN 1558603433. Citado na página 9.

CZAJKOWSKI, T. S.; AYDONAT, U.; DENISENKO, D.; FREEMAN, J.; KINSNER, M.; NETO, D.; WONG, J.; YIANNACOURAS, P.; SINGH, D. P. From opencl to high-performance hardware on fpgas. In: *22nd International Conference on Field Programmable Logic and Applications (FPL)*. [S.l.: s.n.], 2012. p. 531–534. ISSN 1946-147X. Citado na página 36.

DAGUM, L.; MENON, R. Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 5, n. 1, p. 46–55, jan. 1998. ISSN 1070-9924. Disponível em: <http://dx.doi.org/10.1109/99.660313>. Citado na página 18.

DENNING, P. J. The locality principle. *Commun. ACM*, ACM, New York, NY, USA, v. 48, n. 7, p. 19–24, jul. 2005. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/1070838.1070856>. Citado na página 64.

DIAZ, J.; MUñOZ-CARO, C.; NIñO, A. A survey of parallel programming models and tools in the multi and many-core era. *IEEE Transactions on Parallel and Distributed Systems*, v. 23, n. 8, p. 1369–1386, Aug 2012. ISSN 1045-9219. Citado na página 18.

EIMANDAR, P. *DirectX 11.1 Game Programming*. [S.l.]: Packt Publishing, 2013. ISBN 184969480X, 9781849694803. Citado na página 17.

FANG, J.; VARBANESCU, A. L.; SIPS, H. A comprehensive performance comparison of cuda and opencl. In: *2011 International Conference on Parallel Processing*. [S.l.: s.n.], 2011. p. 216–225. ISSN 0190-3918. Citado na página 35.

FARRAR, M. Striped smith–waterman speeds database searches six times over other simd implementations. *Bioinformatics*, Oxford University Press, Oxford, UK, v. 23, n. 2, p. 156–161, jan. 2007. ISSN 1367-4803. Disponível em: <http://dx.doi.org/10.1093/bioinformatics/btl582>. Citado na página 36.

FLYNN, M. J. Very high-speed computing systems. *Proceedings of the IEEE*, v. 54, n. 12, p. 1901–1909, Dec 1966. ISSN 0018-9219. Citado na página 7.

FORUM, T. M. *MPI: A Message-Passing Interface Standard*. Knoxville, TN, USA, 1994. Citado na página 11.

GROTKER, T. *System Design with SystemC*. Norwell, MA, USA: Kluwer Academic Publishers, 2002. ISBN 1402070721. Citado na página 59.

GUMMARAJU, J.; EREZ, M.; COBURN, J.; ROSENBLUM, M.; DALLY, W. J. Architectural support for the stream execution model on general-purpose processors. In: *16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*. [S.l.: s.n.], 2007. p. 3–12. ISSN 1089-795X. Citado na página 31.

GUMMARAJU, J.; ROSENBLUM, M. Stream programming on general-purpose processors. In: *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*. [S.l.: s.n.], 2005. p. 12 pp.–. ISSN 1072-4451. Citado na página 32.

HALAAS, A.; SVINGEN, B.; NEDLAND, M.; SAETROM, P.; SNOVE, O.; BIRKELAND, O. R. A recursive misd architecture for pattern matching. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, v. 12, n. 7, p. 727–734, July 2004. ISSN 1063-8210. Citado na página 8.

HENNESSY, J.; JOUPPI, N.; PRZYBYLSKI, S.; ROWEN, C.; GROSS, T.; BASKETT, F.; GILL, J. Mips: A microprocessor architecture. In: *Proceedings of the 15th Annual Workshop on Microprogramming*. Piscataway, NJ, USA: IEEE Press, 1982. (MICRO 15), p. 17–22. Disponível em: <http://dl.acm.org/citation.cfm?id=800036.800930>. Citado na página 42.

HENNESSY, J. L.; PATTERSON, D. A. *Computer Architecture, Fifth Edition: A Quantitative Approach*. 5th. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN 012383872X, 9780123838728. Citado na página 7.

HOFMANN, M.; KLINKENBERG, R. *RapidMiner: Data Mining Use Cases and Business Analytics Applications*. [S.l.]: Chapman & Hall/CRC, 2013. ISBN 1482205491, 9781482205497. Citado na página 17.

IQBAL, S. M. Z.; LIANG, Y.; GRAHN, H. Parmibench - an open-source benchmark for embedded multiprocessor systems. *IEEE Computer Architecture Letters*, v. 9, n. 2, p. 45–48, Feb 2010. ISSN 1556-6056. Citado na página 60.

JANTSCH, A.; TENHUNEN, H. (Ed.). *Networks on Chip*. Hingham, MA, USA: Kluwer Academic Publishers, 2003. ISBN 1-4020-7392-5. Citado na página 27.

KELTCHER, C. N.; MCGRATH, K. J.; AHMED, A.; CONWAY, P. The amd opteron processor for multiprocessor servers. *IEEE Micro*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 23, n. 2, p. 66–76, mar. 2003. ISSN 0272-1732. Disponível em: <http://dx.doi.org/10.1109/MM.2003.1196116>. Citado na página 7.

Khronos Group. *The OpenCL Specification*. [S.l.], 2010. Citado 2 vezes nas páginas 3 e 19.

KIRK, D. B.; HWU, W.-m. W. *Programming Massively Parallel Processors: A Hands-on Approach*. 1st. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010. ISBN 0123814723, 9780123814722. Citado na página 14.

KUMAR, R.; FARKAS, K. I.; JOUPPI, N. P.; RANGANATHAN, P.; TULLSEN, D. M. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In: *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2003. (MICRO 36), p. 81–. ISBN 0-7695-2043-X. Disponível em: <http://dl.acm.org/citation.cfm?id=956417.956569>. Citado na página 3.

KURODA, T. Cmos design challenges to power wall. In: *2001 International Microprocesses and Nanotechnology Conference*. [S.l.: s.n.], 2001. p. 6–7. Citado na página 13.

LEE, J.; KIM, J.; KIM, J.; SEO, S.; LEE, J. An opencl framework for homogeneous manycores with no hardware cache coherence. In: *2011 International Conference on Parallel Architectures and Compilation Techniques (PACT)*. [S.l.: s.n.], 2011. p. 56–67. ISSN 1089-795X. Citado na página 37.

LIU, Y.; SCHMIDT, B.; MASKELL, D. L. Cudasw++2.0: enhanced smith-waterman protein database search on cuda-enabled gpus based on simt and virtualized simd abstractions. *BMC Research Notes*, v. 3, n. 1, p. 1–12, 2010. ISSN 1756-0500. Disponível em: <http://dx.doi.org/10.1186/1756-0500-3-93>. Citado na página 36.

MACEDONIA, M. The gpu enters computing's mainstream. *Computer*, v. 36, n. 10, p. 106–108, Oct 2003. ISSN 0018-9162. Citado 2 vezes nas páginas 3 e 17.

MAHESH, K.; CONSTANTINESCU, G.; APTE, S.; IACCARINO, G.; HAM, F.; MOIN, P. Large-eddy simulation of reacting turbulent flows in complex geometries. *Journal of Applied Mechanics, Transactions ASME*, American Society of Mechanical Engineers(ASME), v. 73, n. 3, p. 374–381, 5 2006. ISSN 0021-8936. Citado na página 31.

MAHMMOD, B. S. Key note lecture - microprocessors evolution. In: *2013 International Conference on Electrical, Communication, Computer, Power, and Control Engineering (ICECCPCE)*. [S.l.: s.n.], 2013. p. 1–1. Citado na página 8.

MARTINEZ, G.; GARDNER, M.; FENG, W. c. Cu2cl: A cuda-to-opencl translator for multi- and many-core architectures. In: *2011 IEEE 17th International Conference on Parallel and Distributed Systems (ICPADS)*. [S.l.: s.n.], 2011. p. 300–307. ISSN 1521-9097. Citado na página 37.

MITCHELL, M. *An Introduction to Genetic Algorithms*. Cambridge, MA, USA: MIT Press, 1998. ISBN 0262631857. Citado na página 64.

MORAD, A.; YAVITS, L.; GINOSAR, R. Gp-simd processing-in-memory. *ACM Trans. Archit. Code Optim.*, ACM, New York, NY, USA, v. 11, n. 4, p. 53:1–53:26, jan. 2015. ISSN 1544-3566. Disponível em: <http://doi.acm.org/10.1145/2686875>. Citado na página 33.

MUNSHI, A.; GASTER, B.; MATTSON, T. G.; FUNG, J.; GINSBURG, D. *OpenCL Programming Guide*. 1st. ed. [S.l.]: Addison-Wesley Professional, 2011. ISBN 0321749642, 9780321749642. Citado na página 19.

NAH, J.; LEE, J.; KIM, H.; LEE, J.; HWANG, S. J.; YOO, D.; LEE, J. An opencl optimizing compiler for reconfigurable processors. In: *2013 International Conference on Field-Programmable Technology (FPT)*. [S.l.: s.n.], 2013. p. 184–191. Citado na página 38.

NEPOMUCENO, R. S.; SANTOS, J. C.; LUZ, L. O.; SILVA, I. S. An opencl-compliant multi-core platform and its companion compiler. In: *2015 Brazilian Symposium on Computing Systems Engineering (SBESC)*. [S.l.: s.n.], 2015. p. 116–121. Citado 3 vezes nas páginas 38, 39 e 67.

NEUMANN, J. von. First draft of a report on the edvac. *IEEE Annals of the History of Computing*, v. 15, n. 4, p. 27–75, 1993. ISSN 1058-6180. Citado na página 8.

PARIS, S.; HASINOFF, S. W.; KAUTZ, J. Local laplacian filters: Edge-aware image processing with a laplacian pyramid. In: *ACM SIGGRAPH 2011 Papers*. New York, NY, USA: ACM, 2011. (SIGGRAPH '11), p. 68:1–68:12. ISBN 978-1-4503-0943-1. Disponível em: <http://doi.acm.org/10.1145/1964921.1964963>. Citado na página 62.

PERICAS, M.; CRISTAL, A.; CAZORLA, F. J.; GONZALEZ, R.; JIMENEZ, D. A.; VALERO, M. A flexible heterogeneous multi-core architecture. In: *16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*. [S.l.: s.n.], 2007. p. 13–24. ISSN 1089-795X. Citado na página 3.

PRZYBYLSKI, S. A. *Cache and memory hierarchy design : a performance-directed approach*. San Mateo, CA: Morgan Kaufmann, 1990. ISBN 1-55860-136-8. Disponível em: <http://opac.inria.fr/record=b1104120>. Citado na página 10.

RAZMYSLOVICH, D.; MARCUS, G.; GIPP, M.; ZAPATKA, M.; SZILLUS, A. Implementation of smith-waterman algorithm in opencl for gpus. In: *2010 Ninth International Workshop on Parallel and Distributed Methods in Verification, and Second International Workshop on High Performance Computational Systems Biology*. [S.l.: s.n.], 2010. p. 48–56. Citado na página 35.

ROTEM, E.; NAVEH, A.; ANANTHAKRISHNAN, A.; WEISSMANN, E.; RAJWAN, D. Power-management architecture of the intel microarchitecture code-named sandy bridge. *IEEE Micro*, v. 32, n. 2, p. 20–27, March 2012. ISSN 0272-1732. Citado na página 3.

SCHALLER, R. R. Moore's law: Past, present, and future. *IEEE Spectr.*, IEEE Press, Piscataway, NJ, USA, v. 34, n. 6, p. 52–59, jun. 1997. ISSN 0018-9235. Disponível em: <http://dx.doi.org/10.1109/6.591665>. Citado na página 12.

SCHREIBER, R. Manycores in the future. In: _____. *High Performance Computing and Communications: Third International Conference, HPCC 2007, Houston, USA, September 26-28, 2007. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007. p. 5–5. ISBN 978-3-540-75444-2. Disponível em: <http://dx.doi.org/10.1007/978-3-540-75444-2_5>. Citado na página 13.

SHEN, J.; FANG, J.; SIPS, H.; VARBANESCU, A. L. Performance traps in opencl for cpus. In: *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. [S.l.: s.n.], 2013. p. 38–45. ISSN 1066-6192. Citado na página 35.

SNIR, M.; OTTO, S.; HUSS-LEDERMAN, S.; WALKER, D.; DONGARRA, J. *MPI-The Complete Reference, Volume 1: The MPI Core*. 2nd. (revised). ed. Cambridge, MA, USA: MIT Press, 1998. ISBN 0262692155. Citado na página 19.

STROUSTRUP, B. *The C++ Programming Language*. 3rd. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000. ISBN 0201700735. Citado na página 59.

SU, C. L.; CHEN, P. Y.; LAN, C. C.; HUANG, L. S.; WU, K. H. Overview and comparison of opencl and cuda technology for gpgpu. In: *2012 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*. [S.l.: s.n.], 2012. p. 448–451. Citado na página 35.

TOTONI, E.; BEHZAD, B.; GHIKE, S.; TORRELLAS, J. Comparing the power and performance of intel's scc to state-of-the-art cpus and gpus. In: *Proceedings of the 2012 IEEE International Symposium on Performance Analysis of Systems & Software*. Washington, DC, USA: IEEE Computer Society, 2012. (ISPASS '12), p. 78–87. ISBN 978-1-4673-1143-4. Disponível em: <http://dx.doi.org/10.1109/ISPASS.2012.6189208>. Citado na página 13.

VUDUC, R.; DEMMEL, J. W.; YELICK, K. A.; KAMIL, S.; NISHTALA, R.; LEE, B. Performance optimizations and bounds for sparse matrix-vector multiply. In: *ACM/IEEE 2002 Conference Supercomputing*. [S.l.: s.n.], 2002. p. 26–26. ISSN 1063-9535. Citado na página 31.

WOO, M.; NEIDER, J.; DAVIS, T.; SHREINER, D. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*. 3rd. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN 0201604582. Citado na página 17.

YAN, X.; SHI, X.; SUN, Q. An opencl micro-benchmark suite for gpus and cpus. In: *2012 13th International Conference on Parallel and Distributed Computing, Applications and Technologies*. [S.l.: s.n.], 2012. p. 53–58. ISSN 2379-5352. Citado na página 36.

ZEFERINO, C. A.; SUSIN, A. A. Socin: a parametric and scalable network-on-chip. In: *16th Symposium on Integrated Circuits and Systems Design, 2003. SBCCI 2003. Proceedings.* [S.l.: s.n.], 2003. p. 169–174. Citado 2 vezes nas páginas 10 e 27.