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

CLEM & OCEAN: Dois Compiladores OpenCL para as Arquiteturas *Manycore* METAL e ArachNoC

Jônatas Carneiro dos Santos Ferreira

Teresina-PI, Setembro de 2016

Jônatas Carneiro dos Santos Ferreira

CLEM & OCEAN: Dois Compiladores OpenCL para as Arquiteturas *Manycore* METAL e ArachNoC

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. Raimundo Santos Moura

Teresina-PI

Setembro de 2016

A minha esposa F. Maria, minha mãe Marinalva e minha irmã Márcia por sempre estarem comigo em todos os momentos.

Agradecimentos

Agradeço a Deus por minha vida, família e amigos.

A minha esposa, Francisca Maria, pelo apoio incondicional que por diversas vezes abdicou dos passeios nos finais de semana para apoiar-me nessa jornada.

A minha mãe e irmã, Marinalva e Márcia, pelo amor e apoio incondicional.

Aos meus orientadores, prof. Dr. Ivan Saraiva Silva e prof. Dr. Raimundo Santo Moura, pelo suporte, correções e incentivos.

A esta instituição, seu corpo docente, direção e administração.

À FAPEPI pelo apoio financeiro para realização desta pesquisa.

E a todos que, direta ou indiretamente, fizeram parte na construção deste trabalho.

Muito obrigado!

"Tenha metas. Uma vida sem objetivos é uma existência triste, pois o homem é um ser, historicamente, movido a desafios."
(Renato Collyer)

Resumo

Nos últimos anos, é notória a evolução das arquiteturas *manycores*. Essas arquiteturas se caracterizam por possuírem dezenas de núcleos de processamento integrados em um único circuito que executam tarefas de maneira concorrente.

As arquiteturas serão subutilizadas caso não sejam adotadas ferramentas que ofereçam recursos que simplifiquem o desenvolvimento de aplicações. Assim sendo, este trabalho descreve o desenvolvimento de um conjunto de ferramentas de compilação para as arquiteturas *manycore* METAL (***M****anycor****E*** *Pla****T****form* ***A****dapted to OpenC****L***) e ArachNoC (*Arachnid NoC*). As duas plataformas são *manycore* com múltiplos nós de processamento conectados por uma rede em chip (*NoC - Network on Chip*) com topologia em malha 2D. Possuem oito nós de processamento *multicores* mais um nó mestre dotado de um único núcleo de processamento. Enquanto METAL possui uma arquitetura adaptada ao modelo de programação OpenCL (*Open Computing Language*), ArachNoC confere suporte à programação paralela por meio da inserção das instruções de sincronização.

Para o desenvolvimento do conjunto de ferramentas propostas, o *framework* OpenCL é adotado como modelo de programação. A adoção de OpenCL se justifica pelo fato de tal *framework* ter se tornado, nos dias atuais, a principal alternativa para programação de sistemas dotados de múltiplos núcleos de processamento. O *framework* LLVM (*Low Level Virtual Machine*) foi utilizado para criação do conjunto de ferramentas.

O objetivo é disponibilizar um conjunto de ferramentas de compilação para as plataformas citadas, bem como, contribuir no desenvolvimento de técnicas de geração de códigos para arquiteturas *manycores*, e oferecer recursos que facilitam o desenvolvimento de tais arquiteturas.

Para validação e avaliação das ferramentas propostas, foram realizadas comparações com o compilador GCC (*GNU Compiler Collection*). Os resultados mostram que as ferramentas criadas possuem tempo de compilação e execução superiores ao GCC.

Palavras-chaves: compilador. OpenCL. LLVM. programação paralela. multiprocessamento. *manycore*.

Abstract

In recent years, the evolution of manycores architectures is well known. These architectures are characterized by having dozens of processing cores integrated in a single circuit that perform tasks concurrently.

The architectures will be underutilized if tools are not used that offer features that simplify the development of applications. Therefore, this work describes the development of a compilation toolkit for the manycore METAL (**M**anycor**E** Pla**T**form **A**dapted to OpenC**L**) and Arachnid NoC architectures. The two platforms are multicore with multiple processing nodes connected by a 2D mesh topology (NoC - Network on Chip). They have eight multicore processing nodes plus a master node with a single processing core. While METAL has an architecture adapted to the OpenCL programming model (Open Computing Language), ArachNoC configures parallel programming support by inserting the synchronization instructions.

For the development of the proposed toolkit, the OpenCL framework is adopted as a programming model. The adoption of OpenCL is justified by the fact that this framework has become, today, the main alternative for programming systems with multiple processing cores. The LLVM (Low Level Virtual Machine) framework was used to create the toolkit.

The goal is to provide a set of compilation tools for the mentioned platforms, as well as to contribute to the development of coding techniques for manycores architectures, and to provide features that facilitate the development of such architectures.

For the validation and evaluation of the proposed tools, comparisons were made with the GCC compiler (GNU Compiler Collection). The results show that the tools created have more compile time and execution than GCC.

Keywords: compiler. OpenCL. LLVM. parallel programming. multi-processing. manycore.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

API	*Application Programming Interface*
ARM	*Acorn RISC Machine*
API	*Application Programming Interface*
ArachNoC	*Arachnid NoC*
CLEM	*OpenCL CompilEr Metal*
CPU	*Central Processing Unit*
CUDA	*Compute Unified Device Architecture*
GCC	*GNU Compiler Collection*
GPU	*Graphics Processing Unit*
ISA	*Instruction Set Architecture*
LLVM	*Low Level Virtual Machine*
METAL	*Manycore Platform Adapted to OpenCL*
MIPS	*Microprocessor without Interlocked Pipeline Stages*
MPI	*Message Passing Interface*
NoC	*Network on Chip*
OCEAN	*OpenCL Compiler ArachNoC*
OpenCL	*Open Computing Language*
OpenMP	*Open Multi-Processing*
PIPS	*Programming Integrated Parallel System*
SIMD	*Single Instruction Multiple Data*
SO	*Sistema Operacional*
SSE	*Streaming SIMD Extensions*
UML	*Unified Modeling Language*

Sumário

Introdução

Exitem duas maneiras de obter um programa composto de tarefas para serem executadas em paralelo. Uma é o paralelismo implícito, obtido com o uso de compiladores paralelizadores (AHO; SETHI; LAM, 2008), que detectam o paralelismo existente em um código sequencial, gerando código paralelo automaticamente. Esse método aumenta consideravelmente a complexidade da elaboração do compilador. Outra forma, chamada de paralelismo explícito, o programador fica encarregado do desenvolvimento das tarefas que vão executar em paralelo. Nesse caso, o programador faz uso de linguagens e ferramentas de programação que lhe oferecem suporte à programação paralela. Essa abordagem exige que, além de dominar o algoritmo, o programador domine certas particularidades da arquitetura alvo (MAJETI; SARKAR, 2015). Por outro lado, o compilador tem sua complexidade reduzida com relação ao primeiro caso.

Existem compiladores de paralelização automática como o PGI Unified Binary^{TM} (The Potland Group, 2016) que fornece um conjunto integrado de compiladores e ferramentas para o desenvolvimento de aplicações de alto desempenho. PGI possui paralelização automática de *loops* em códigos fontes, onde vários processadores ou núcleos são utilizados para executar esses *loops* paralelizáveis. Uma outra ferramenta, o PIPS (*Programming Integrated Parallel System*) (PIPS project, 2016), é um *source-to-source framework* para análise e transformações de código em programas escritos em C e Fortran. Um dos projetos realizados com o PIPS é o SPEAR-DE interface (2016) que realiza a transformação de um código escrito em C para OpenCL.

Construir um compilador paralelizador eficiente para uma grande variedade de casos e aplicações é uma tarefa provadamente árdua (YANG et al., 2011) e, ainda assim, o desenvolvedor deve ter cuidado para não construir códigos que dificultam a análise e a geração de códigos paralelos.

Vários *frameworks* foram desenvolvidos para dar suporte à programação paralela, o que facilita a evolução dos sistemas computacionais multiprocessados, por exemplo, CUDA (KIRK; HWU, 2010) (*Compute Unified Device Architecture*) e OpenCL (*Open Computing Language*) (TSUCHIYAMA et al., 2010), ambos são extensões para a linguagem de programação C. OpenCL e CUDA executam um trecho de código, chamado *kernel*, em paralelo mas diferem nas conversões para mapear o *kernel* em elementos de processamento. O *framework* CUDA é privado, suas extensões funcionam apenas em GPUs (*Graphics Processing Unit*) da NVidia (KARIMI; DICKSON; HAMZE, 2010). OpenCL é um *framework open source* para o desenvolvimento de programas destinados a plataformas heterogêneas. Criado por um consórcio de empresas líderes de mercado em computação paralela, o projeto

está em constante crescimento e inclui as principais arquiteturas de processamento como: CPUs (*Central Processing Unit*), GPUs, entre outros aceleradores (STONE; GOHARA; SHI, 2010; FANG; VARBANESCU; SIPS, 2011). Outro *framework*, o OpenMP (OPENMP, 2016) é uma especificação para um conjunto de diretivas de compilador, bibliotecas e variáveis de ambiente que podem ser usadas para especificar paralelismo em programas escritos em Fortran e C/C++. OpenMP é usado no desenvolvimento de programas paralelos para ambientes de memória compartilhada e fornece um modelo de execução conhecido como *fork-and-join*, onde um programa inicia executando um simples processo ou *thread* até encontrar uma diretiva de paralelização e, em seguida, a *thread* em execução cria várias *threads* filhas para serem executadas em paralelo. Após todas as filhas finalizarem, a *thread* inicial coleta possíveis dados gerados pelas filhas e finaliza o programa. MPI (*Message Passing Interface*) (SNIR et al., 1998) é uma biblioteca que baseia-se em troca de mensagens e é voltada para arquiteturas com memória distribuída. MPI é a primeira biblioteca de passagem de mensagem. As vantagens no desenvolvimento de programas utilizando MPI está na portabilidade, eficiência e flexibilidade.

Motivação

A evolução dos sistemas multiprocessados tornou possível a superação da limitação de frequência máxima em processadores *single-core* (PAONE, 2014). Porém, com o aumento do número de núcleos de processamento, surgiu a necessidade de métodos e ferramentas que forneçam suporte ao desenvolvedor para o máximo aproveitamento desses ambientes de execução. A motivação deste trabalho é o fato das plataformas não possuírem um ambiente de compilação para linguagem paralela de alto nível.

Objetivos

As plataformas não possuem um ambiente de compilação. Assim, o objetivo deste trabalho é construir duas ferramentas para suporte ao desenvolvimento de *software* em alto nível para as arquiteturas METAL e ArachNoC. Essas ferramentas são compostas de dois compiladores chamados de CLEM (*Open**CL** Compil**E**r **M**etal*) e OCEAN (***O**penCL **C**ompil**E**r **A**rach**N**oC*), cada um acompanhado de uma implementação da especificação da *API OpenCL*, desenvolvidas para o suporte à programação paralela. CLEM e OCEAN são os compiladores OpenCL das plataformas METAL e ArachNoC, respectivamente, ambos consistem em um sistema multiprocessado integrado com uma rede em chip (*NoC - Network on Chip*).

Organização do Documento

O restante do documento está organizado em cinco capítulos. o Capítulo 1 apresenta uma revisão sobre o *framework* OpenCL, seus modelos de plataforma, execução e memória. Os processos de compilação *online* e *offline* do OpenCL também são explanados. Realiza-se uma descrição das arquiteturas METAL e ArachNoC, discutindo a particularidade de cada plataforma. Apresentamos a infra-estrutura do compilador LLVM, seu *frontend* Clang, sua representação intermediária e o *Backend* MIPS.

O Capítulo 2 mostra o que existe de recente no desenvolvimento de compiladores com a ferramenta LLVM, da mesma foma, apresenta-se o estado da arte no desenvolvimento de API's OpenCL.

Já o Capítulo 3 descreve a construção dos compiladores CLEM e OCEAN e suas APIs OpenCL. São explanadas, as funções de *hardware* desenvolvidas para cada arquitetura

O Capítulo 4 mostra as validações e resultados. Realiza-se as comparações com GCC: tempo de compilação, densidade de código gerado e tempo de execução.

O Capítulo 5 mostra as conclusões finais e os trabalhos futuros.

1 Fundamentação Teórica

1.1 OpenCL

OpenCL é a primeira plataforma *open-source* para computação em sistemas heterogêneos compostos por CPUs, GPUs, e outros processadores (MUNSHI et al., 2011). Criada pela Apple e padronizada pelo Khronos Group (KHRONOS, 2015a), OpenCL oferece aos desenvolvedores um ambiente uniforme de programação paralela para escrever códigos portáveis para aqueles sistemas heterogêneos. O objetivo do padrão OpenCL é unificar em um único paradigma e conjunto de ferramentas, o desenvolvimento de soluções de computação paralela para dispositivos de naturezas distintas.

OpenCL é um *framework* que oferece um padrão de código e métodos independente do tipo de processador ou fornecedor de *hardware*. Esse padrão é definido nas duas seguintes especificações:

- ***Especificação da Linguagem OpenCL***: Define o padrão da linguagem suportada que é baseada no padrão C (C99) com algumas extensões e restrições. Exceto essas extensão e restrições, a linguagem pode ser utilizada da mesma maneira que o padrão C;

- ***Especificação da API OpenCL***: Contém as especificações para implementação de uma API. Assim, padroniza as funções usadas por um processo/*thread* que envia tarefas para computarem concorrentemente em outros núcleos de execução. Por exemplo, considerando-se uma implementação daquela API que ofereça suporte tanto para CPU como para GPU, uma CPU pode utilizar outras CPU's que incluam unidades SSE (*Streaming SIMD Extensions*), bem como todas as GPU's para realizar computação em paralelo. Além disso, tudo pode ser escrito usando OpenCL (TSUCHIYAMA et al., 2010).

O padrão OpenCL providencia uma abstração do *hardware* que define hierarquias de modelos de plataforma, memória e execução. Esses modelos são como o OpenCL vê o *hardware* e, desse modo, disponibiliza para o desenvolvedor essa visão. São apresentados os modelos OpenCL.

1.1.1 Modelo de plataforma

O modelo de plataforma é uma abstração de como o OpenCL vê um *hardware* (KHRONOS, 2015b). Ele consiste de um *host* ligado a um ou mais dispositivos. Os dispositivos

OpenCL são divididos em uma ou mais unidades de computação, que por sua vez, são divididas em um ou mais elementos de processamento. Um exemplo dessa abstração em um ambiente real seria: o *host* abstraindo uma CPU Intel core i3, por exemplo. Os dispositivos poderiam ser, por exemplo, uma GPU NVidia e outra GPU ATI. As unidades de computação são os grupos de núcleos dentro das GPUs. Já o elemento de processamento é uma abstração de um único núcleo de processamento dentro das GPUs.

O *host* coordena a execução do programa, preparando e trocando dados com os dispositivos. Também é responsável pelo envio de códigos para execução nos dispositivos. Geralmente é um processador de propósito geral. Já os dispositivos são responsáveis pelas execuções das tarefas em paralelo. Essas tarefas são implementadas em funções OpenCL chamadas de *kernels*. A Figura 1 descreve esse modelo de plataforma.

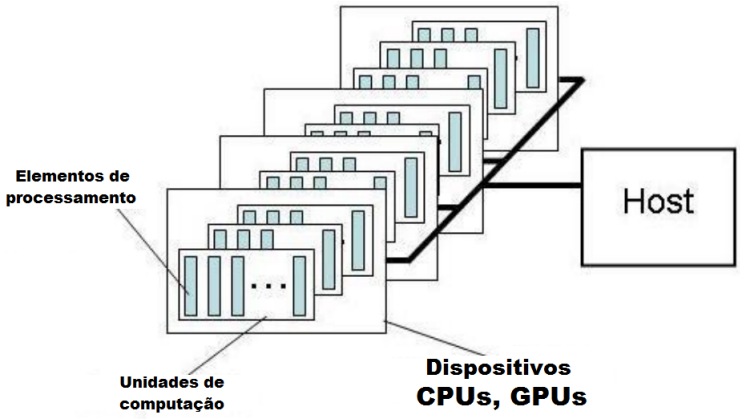


Figura 1 – Modelo de plataforma OpenCL.

Fonte: Khronos (2015b)

Existe um relacionamento entre os elementos do modelo de plataforma e o *hardware* em um sistema. Dependendo de como o compilador otimiza o código, esse relacionamento pode ser uma propriedade fixa de um dispositivo ou dinamicamente configurado para um programa, isso para uma melhor utilização do *hardware*. Esse relacionamento é transparente para o desenvolvedor OpenCL. O modelo de plataforma OpenCL é a visão do hardware disponibilizada para o desenvolvedor.

1.1.2 Modelo de execução

O modelo de execução do OpenCL define duas distintas unidades de execução, isto é, dois trechos distintos de códigos: ***kernels*** que executam sobre um ou mais dispositivos e um **programa *host*** que executa sobre o *host*. Os *kernels* são responsáveis por realizar a computação paralela, já o programa *host* fica a cargo das configurações e envio dos *kernels* para serem executados. Um *kernel* em execução é chamado de *work-item* e um *work-group* é a reunião de um ou mais *work-items*. A Figura 2 exibe as duas unidades de execução (*kernels* e programa *host*). Também exibe um dispositivo recebendo o código *kernel* para

execução. O programa *host* executa sobre o *host*, que é responsável pela configuração e envio do código *kernel*.

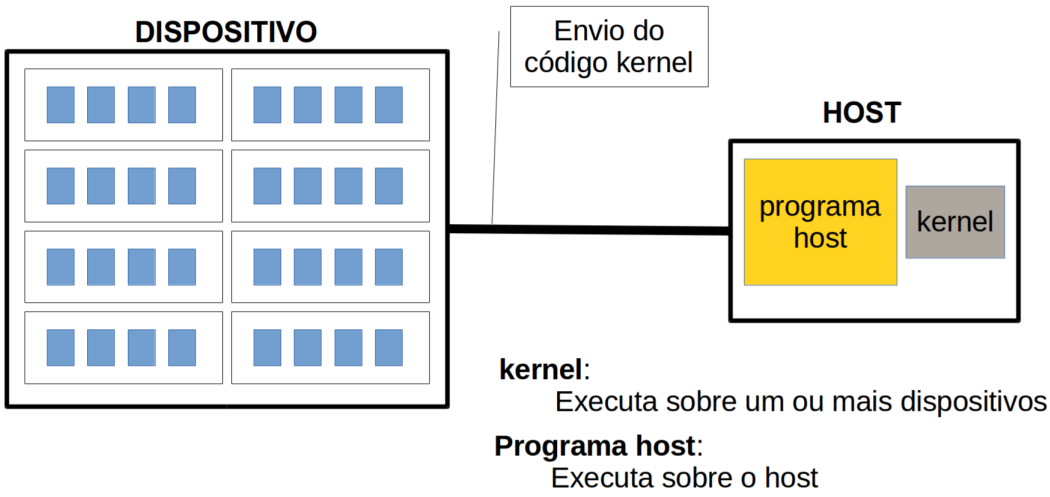


Figura 2 – Modelo de execução OpenCL.

O modelo de execução do padrão OpenCL define um espaço de execução chamado NDRange. Esse espaço de execução pode ser monodimensional, bidimensional ou tridimensional. Cada dimensão é definida por um índice e cada ponto neste espaço representa um *kernel* em excução. Um NDRange é definido por três vetores de inteiros de tamanho um para o NDRanger monodimensional, dois para o bidimensional ou três para o tridimensional. Esses vetores são:

- ***global size***: Dimensões externas do espaço de índices em cada dimensão;
- ***offset***: Indica o valor inicial dos índices em cada dimensão (zero por *default*);
- ***local size***: Dimensões de um *work-group* em cada dimensão.

A Figura 3 exibe um modelo de espaço de índice bidimensional para execução dos *work-items*.

Um objeto contexto, criado por um programa em execução no *host*, define o ambiente no qual os *kernels* irão executar. Ele gerencia os seguintes recursos:

- ***Devices***: Objeto contendo um ou mais dispositivos revelados por uma plataforma OpenCL;
- ***Kernel Objects***: Objeto contendo a função *kernel* acompanhada dos valores dos argumentos que executarão sobre um dispositivo;
- ***Program Objects***: Objeto contendo o código fonte e o executável de um *kernel*;

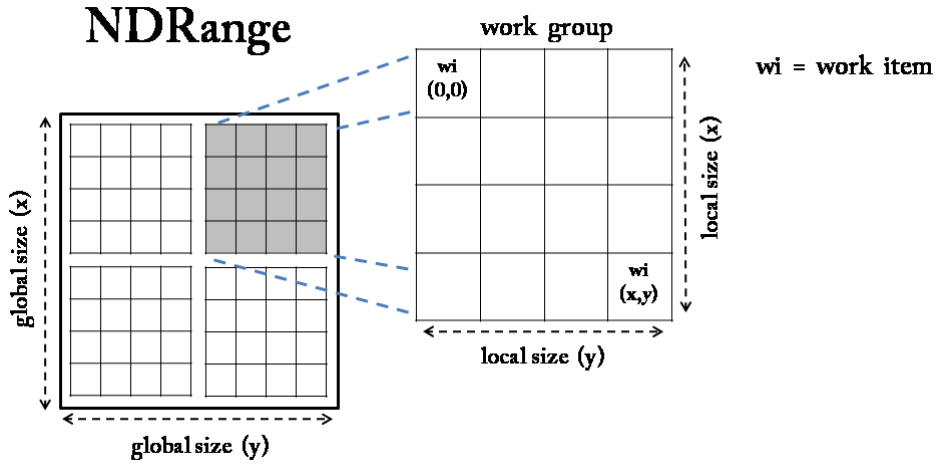


Figura 3 – O espaço de índice NDRange do OpenCL.

Fonte: Khronos (2015b)

- ***Memory Objects***: Objeto contendo variáveis visíveis para o *host* e para os dispositivos. Kernels em execução operam sobre estas variáveis.

Um programa em execução no *host* usa a API OpenCL para criar e gerenciar o contexto. Funções dentro da API OpenCL habilitam o *host* para interagir com os dispositivos através de um *command-queue*. Cada *command-queue* é associado com um único dispositivo. Os comandos dentro de um *command-queue* podem ser de três tipos:

- ***Kernel-enqueue***: Um comando contendo um *Kernel Objects* mais todas as configurações necessárias para executar sobre um dispositivo;
- ***Memory***: Transfere dados entre a memória do *host* e a do dispositivo;
- ***Synchronization***: Explícito ponto de sincronização que define ordem entre os comandos.

1.1.3 Modelo de memória

O modelo de memória do OpenCL descreve a estrutura e o comportamento das memórias disponibilizadas por uma plataforma OpenCL. As memórias em OpenCL são divididas em duas partes:

- ***Memória do host***: Memória disponível para o *host*. *Memory Objects* são movidos entre o *host* e os dispositivos através de funções dentro da API OpenCL;
- ***Memória dos dispositivos***: Memória disponível para os *kernels* que executam sobre dispositivos.

As memórias nos dispositivos consistem em quatro espaços de endereçamentos ou regiões de memórias:

- ***Memória global***: Essa região de memória permite acessos de leitura/escrita para todos *work-items* em todos os *work-groups* que executam em algum dispositivo. As leituras/escritas em uma memória global podem ser em uma memória *cache* dependendo da disponibilidade desse recurso no dispositivo;
- ***Memória constante***: Uma região da memória global que permanece constante durante toda a execução de um comando *Kernel-enqueue* em um dispositivo. O *host* aloca e inicializa essa região;
- ***Memória local***: Região de memória local para um *work-group*. Essa região de memória é usada para alocar variáveis que são compartilhadas por todos *work-items* pertencentes a um *work-group*;
- ***Memória privada***: Região de memória privada para um *work-item*. Variáveis definidas na memória privada de um *work-item* não são visíveis por outro *work-item*.

As regiões de memórias e seus relacionamentos com o modelo de plataforma OpenCL são resumidos na Figura 4. As memórias local e privada são sempre associadas com um específico dispositivo. As memórias global e constante, entretanto, são compartilhadas entre todos os dispositivos dentro de um mesmo contexto. Um dispositivo pode incluir uma *cache* para oferecer acesso eficiente para essas memórias compartilhadas.

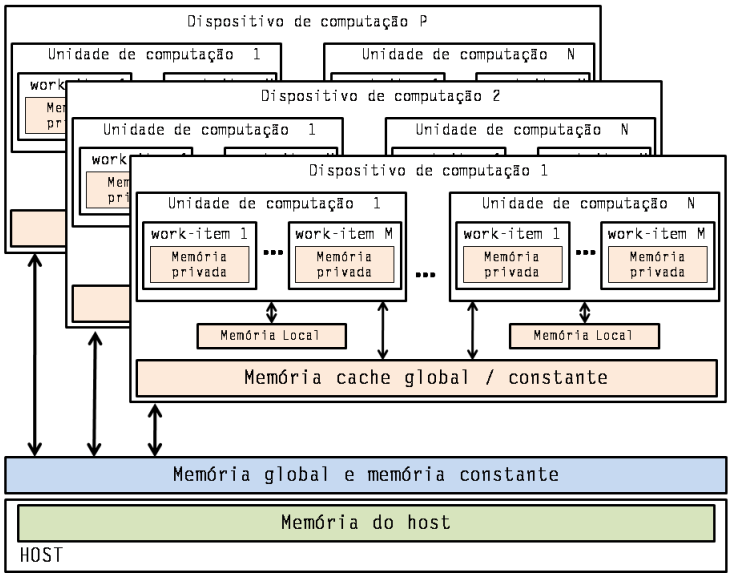


Figura 4 – Modelo de memória do OpenCL.

Fonte: Khronos (2015b)

1.1.4 Compilação *Online* e *Offline*

O desenvolvedor OpenCL necessita das duas seguintes ferramentas: um compilador OpenCL e uma biblioteca (API OpenCL). Um programador, que deseja executar seu código em um dispositivo usando a linguagem OpenCL, deve primeiro compilar o código com um compilador OpenCL desenvolvido para o ambiente de execução. Uma vez o código compilado, ele pode, então, ser enviado para execução no dispositivo. Um programa em execução no *host*, chamado **programa *host***, é responsável por enviar o código, chamado de ***kernel***, para o dispositivo.

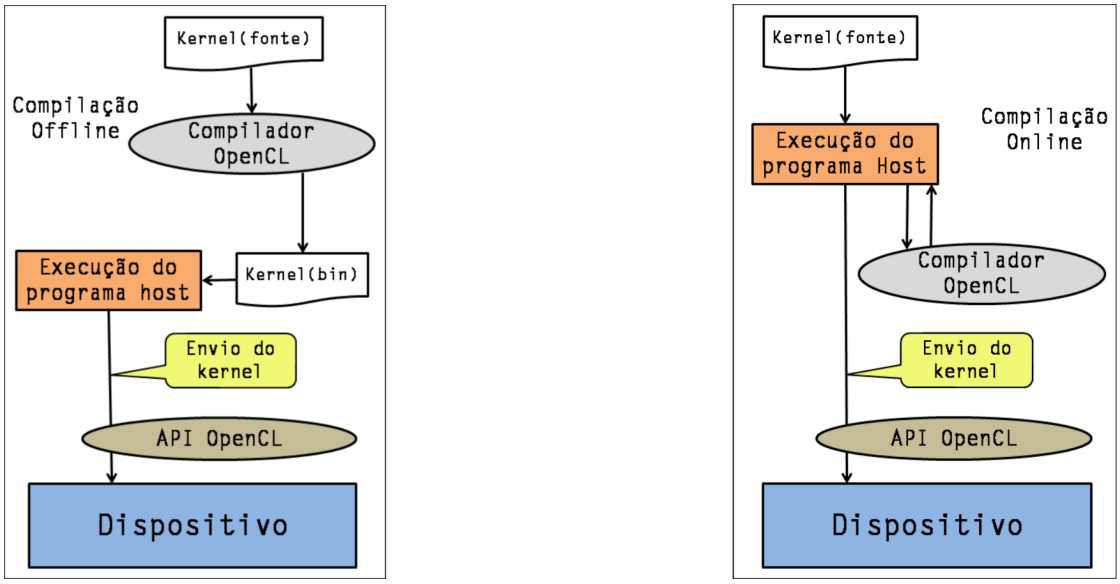
O procedimento de compilação do código fonte do programa *host* segue o procedimento normal de um código escrito em C++. O compilador pode ser um compilador C++ munido da biblioteca OpenCL.

Um *kernel* pode ser compilado de duas formas: *offline* ou *online*. As características dos dois métodos são apresentadas a seguir:

- ***Offline***: O desenvolvedor compila o código fonte do *kernel* utilizando um compilador OpenCL. Esse procedimento pode ser cross-plataforma, ou seja, em um ambiente diferente do local de execução do *software* desenvolvido. Assim, o desenvolvedor disponibiliza o código compilado para o programa *host*. O programa *host* inicia a execução, realiza as configuração necessárias e, então, carrega o arquivo *kernel* previamente compilado. Assim, o código compilado é enviado ao dispositivo apropriado. O desenvolvedor deve ter cuidado em compilar o *kernel* com o compilador OpenCL desenvolvido para o dispositivo de destino. A desvantagem desse método está na execução do *software* sobre vários dispositivos distintos. Um binário do *kernel* deve ser adicionado para cada dispositivo, o que resulta no aumento do tamanho do *software*. Entretanto, ele é adequado para um sistema que dispõe de poucos recursos de *hardware* (TSUCHIYAMA et al., 2010). A Figura 5, lado 5a, descreve uma compilação do *kernel* utilizando o método *offline*.

- ***Online***:

  O código fonte do *kernel* é compilado durante a execução do programa *host*. O programa *host*, utilizando a API OpenCL, carrega o código fonte do *kernel* e o compila com um compilador OpenCL desenvolvido para o dispositivo alvo. Assim, também utilizando a API, envia o código compilado para processamento no dispositivo apropriado. A vantagem desse método está na portabilidade do binário do programa *host* e do arquivo fonte do *kernel*. Esse método deixa o *software* independente do dispositivo alvo, pois durante a compilação o *kernel* se adapta de acordo com o dispositivo de destino. Entretanto, não é adequado para um sistema que dispõe de poucos recursos de *hardware* (TSUCHIYAMA et al., 2010). A Figura 5, lado 5b, descreve uma compilação do *kernel* utilizando o método *online*.



(a) Compilação *offline*. (b) Compilação *online*.

Figura 5 – Formas de compilação do código *kernel*.

1.2 As plataformas alvo

As duas plataformas serão apresentadas em conjunto. Primeiramente o que existe em comum e, em seguida, as particularidades serão explanadas. As plataformas METAL e ArachNoC são *manycores* com múltiplos nós de processamento conectados por uma rede em *chip* com topologia em malha 2D. Essa rede em *chip*, chamada SoCIN (ZEFERINO; SUSIN, 2003), é composta por nove roteadores dotados de cinco canais de comunicação ponto a ponto. Cada roteador possui uma conexão local com um nó de processamento. São oito nós de processamento *multicores*, chamados nós escravos, e um nó mestre dotado de um único núcleo de processamento. A Figura 6 mostra uma visão geral das duas arquiteturas com os nove roteadores interconectados.

No modelo de plataforma OpenCL, o *host* é representado pelo nó mestre. Um dispositivo é representado por todos os nós escravos. Uma unidade de computação é representada por um nó escravo. Cada núcleo de processamento, dentro do nó escravo, representa um elemento de processamento. A Figura 7 exibe a relação entre o modelo de plataforma OpenCL e os componentes das plataformas METAL e ArachNoC.

1.2.1 A Plataforma METAL

METAL é uma arquitetura paralela de propósito geral, homogênea, equipada com recursos que facilitem a execução SIMD (*Single Instruction Multiple Data*) baseada na linguagem OpenCL.

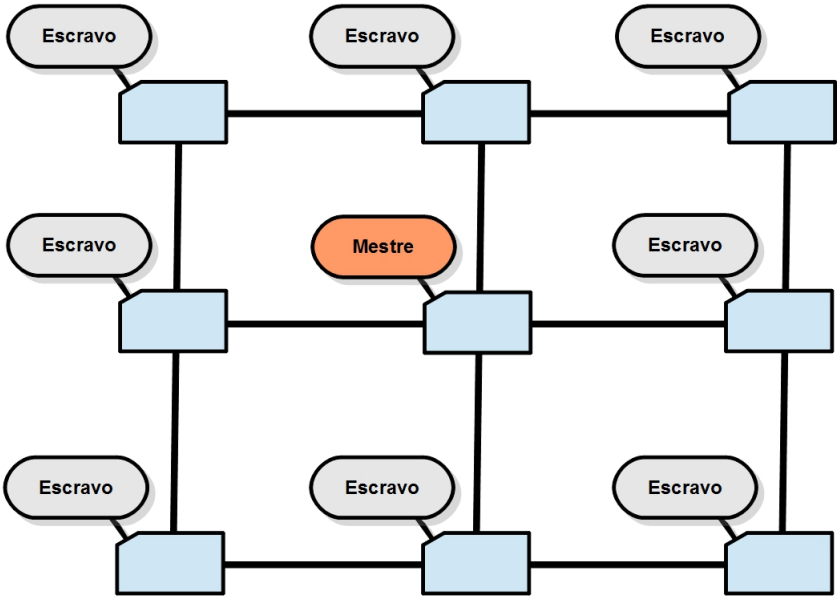


Figura 6 – Visão geral das plataformas METAL e ArachNoC.

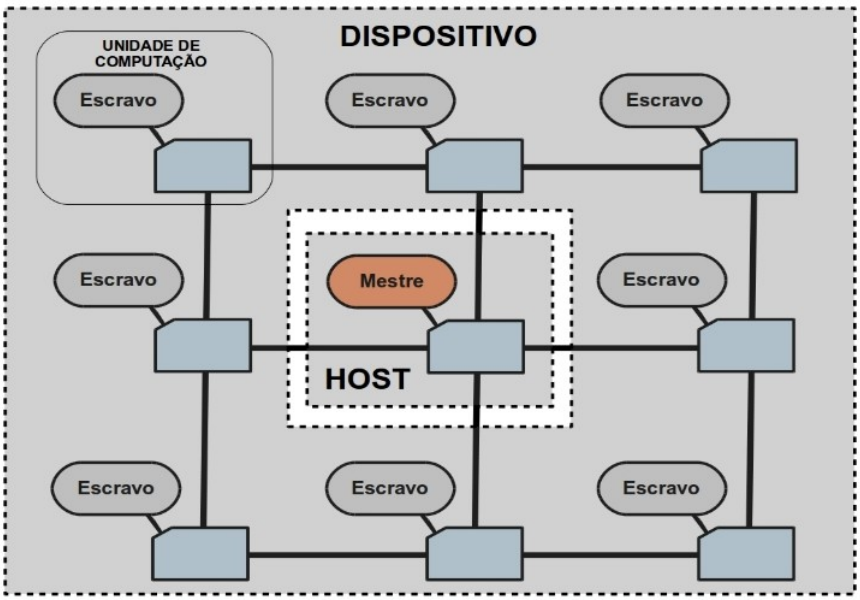


Figura 7 – Papeis no Modelo de Plataforma OpenCL.

O modo de funcionamento de METAL é baseado no padrão mestre/escravo, onde, uma *thread* mestre configura o problema, depois cria várias *threads* (escravas) e então as envia para serem executadas concorrentemente. Após todas computarem, a *thread* mestre coleta os resultados e termina o programa.

A seguir, são listadas as funções e as partes que compõem os dois tipos de nós da rede em *chip*.

O Nó Mestre

A gerência das execuções em toda a plataforma fica a cargo do nó mestre. Ele coordena o envio de tarefas aos núcleos escravos e trata as interrupções lançadas por algum nó escravo. No modelo de plataforma OpenCL, o nó mestre representa o *host* e é composto pelas seguintes partes:

- ***Memórias***: Existe uma para dados e outra para instruções, ambas, são memórias privadas do nó mestre com tamanho de 256 KB. Uma aplicação alocada nas memórias do mestre (dados/instruções) envia a tarefa para computação em paralelos nos nós escravos. Os dados computados nos nós escravos são coletados pelo mestre e alocados na memória de dados. O *software*, em execução, envia comandos ao *hardware* informando o que deve ser enviado e retornado dos núcleos escravos;
- ***Núcleo de execução***: É baseado na implementação do MIPS (HENNESSY et al., 1982), mas, separa os dados e as instruções em duas memórias;
- ***Gerência de Memória***: Controla envio de dados e instruções aos nós escravos, também, recebe dados computados nos nós escravos e guarda-os na memória de dados;
- ***Gerência de Tarefas***: A criação, configuração e envio de tarefas aos nós escravos é função do nó mestre; O Gerenciador de Tarefas atua no escalonamento de tarefas que são enviadas para computação nos nós escravos.
- ***Gerência de Interrupções***: Enfileira as interrupções lançadas por algum nó escravo e administra o tratamento junto ao núcleo mestre de execução;
- ***Interface de rede***: Controla a comunicação com a rede em *chip*.

Uma representação gráfica do nó mestre é mostrada na Figura 8.

O Nó Escravo

Os nós escravos são responsáveis por realizar a computação massiva das aplicações. Cada nó escravo representa uma unidade de computação no modelo de plataforma OpenCL.

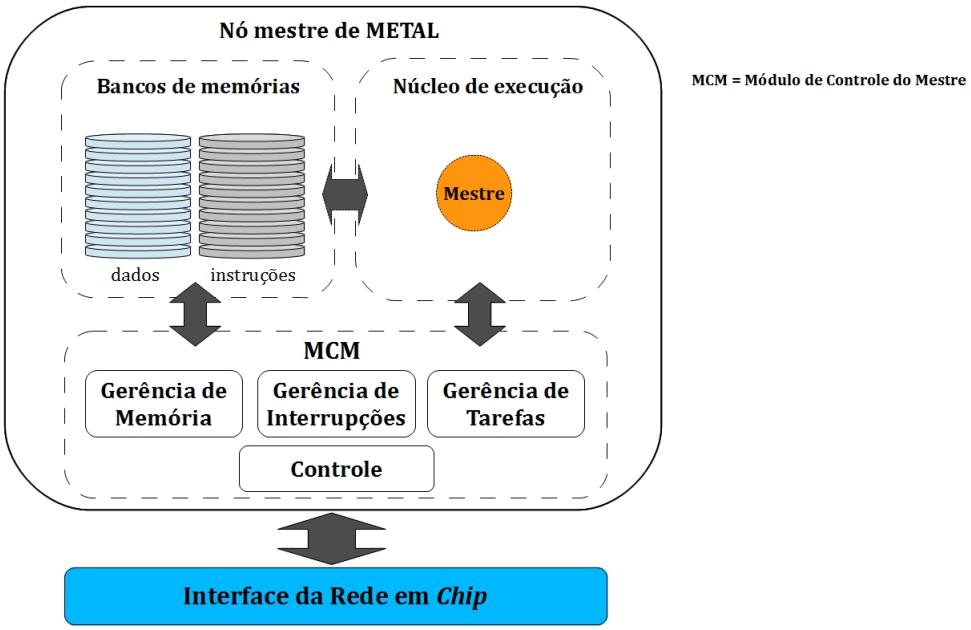


Figura 8 – Representação gráfica do nó mestre da plataforma METAL.

Dentro de um nó escravo, cada núcleo de execução representa um elemento de processamento do modelo OpenCL. Assim, um *work-group* de uma aplicação é enviado para um nó escravo computar e os work-*items*, dentro do grupo, são escalonados para computação nos núcleos de execução dentro do nó escravo.

A seguir, são descritas as partes que compõem um nó escravo.

- ***Memórias***: O nó escravo possui, para alocação de dados, uma memória de 256 KB que é compartilhada entre os núcleos de execução. Essa memória é utilizada para alocar dados necessários à computação. Os dados são transferidos a partir da memória de dados do mestre e, após todas as computações, os resultados são enviados para memória de dados do mestre. Esse procedimento é realizado pelos módulos de gerência de memória existente em cada nó e esse módulos são configurados, previamente, pelo nó mestre. Existe uma memória de instrução para cada núcleo de execução. As instruções, para cada núcleo, são enviadas pelo nó mestre.

- ***Núcleos de execução***: Realizam a computação no nó escravo.

- ***Gerência de Memória***: Módulo responsável pelo controle da transferência de dados entre um nó escravo e o nó mestre.

- ***Escalonador de Tarefas***: O nó mestre pode atribuir um grupo de tarefas para um nó escravo. Essa atribuição consiste em configurar o escalonador de tarefas do nó escravo, além de transferir dados e instruções. O escalonador fica responsável por escalonar as tarefas nos núcleos de execução.

- ***Gerência de Interrupções***: Coordena as interrupções geradas nos núcleos de execução e as respostas de cada solicitação. Por exemplo, o *mutex* (do inglês, *mutual exclusion*) é uma interrupção que solicita acesso a um recurso ou região crítica. Assim, a gerência de interrupções controla os envios e as respostas das interrupções em um nó escravo.

- ***Interface de Rede***: Controla o acesso à rede em *chip*.

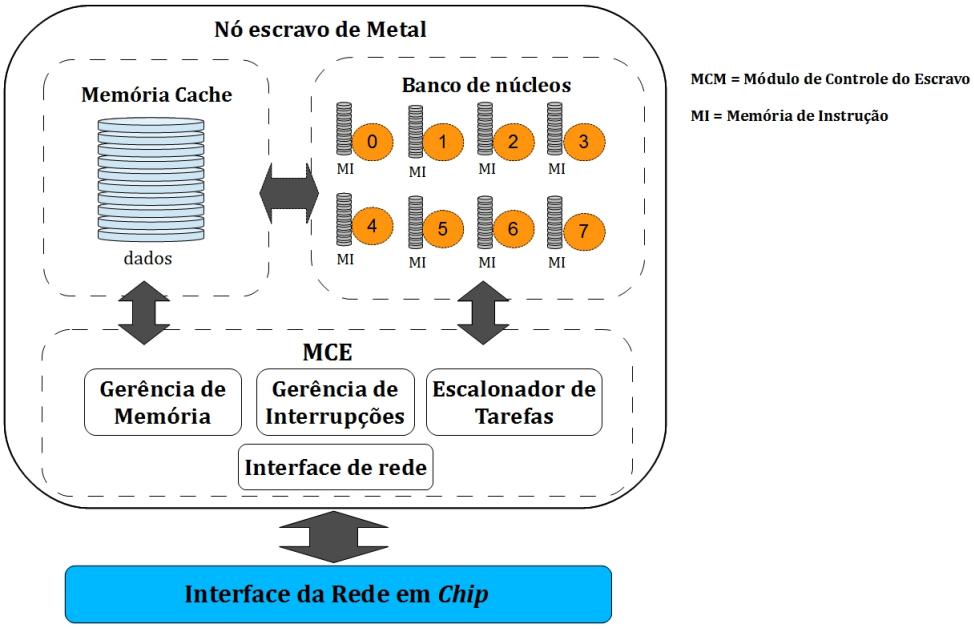


Figura 9 – Representação gráfica de um nó escravo da plataforma METAL.

1.2.2 A Plataforma ArachNoC

ArachNoC possui todos os núcleos de processamento baseados na arquitetura MIPS32 que foi descrita em Hennessy e Patterson (2011), entretanto, com algumas alterações. Uma alteração é a inserção de registradores para guardar dados de interrupções e endereços de rotinas de tratamento dessas interrupções.

O nó mestre, dotado de um único núcleo de processamento, é responsável por coordenar e enviar tarefas para os nós escravos computarem. Ele, também, trata interrupções lançadas por algum nó escravo. Já os nós escravos possuem oito núcleos de processamento.

Tanto o nó mestre quanto os escravos possuem duas memórias, sendo uma de instruções e outra de dados. Em cada nó, existe um MGTI (*Módulo de Gerenciamento de Tarefas e Interrupções*) que é responsável por gerenciar o envio/recebimento de tarefas e interrupções através da rede em *chip*. Existe uma diferença entre o MGTI do nó mestre e dos nós escravos. Enquanto, no nó mestre, ele controla a saída de tarefas e a chegada de interrupções, nos nós escravos, controla a chegada de tarefas e a saída de interrupções.

A comunicação entre os nós escravos e o nó mestre ocorre por meio da troca direta de mensagens. A comunicação entre tarefas em execução, dentro de um nó escravo, ocorre por meio do compartilhamento de variáveis na memória de dados do mesmo.

A seguir, são listadas as particularidades do nó mestre e dos nós escravos.

O Nó Mestre

O nó mestre é responsável por enviar tarefas para os nós escravos e tratar interrupções vindas dos mesmos. Ele é dotado de dois *buffers*, um para tarefas e outro para interrupções. O MGTI coordena o armazenamento de tarefas prontas para envio no *buffer* de tarefas e, assim que possível, as envia aos nós escravos. Ele, também, coordena o recebimento de interrupções e o armazenamento no buffer apropriado. Existindo interrupção no *buffer*, o MGTI coordena o envio das interrupções para o núcleo de execução tratar.

O núcleo de execução, ao receber uma interrupção, salta para a rotina de tratamento correspondente armazenada em uma memória dedicada. Após o tratamento da interrupção, ele restaura o contexto e continua a computação.

A Figura 10 mostra o nó mestre com o único núcleo de execução ligado à duas memórias. O MGTI gerencia o núcleo e os *buffers*, ele é responsável, também, pela comunicação com a interface de rede. As memórias dedicadas estão representadas ao lado do núcleo de execução.

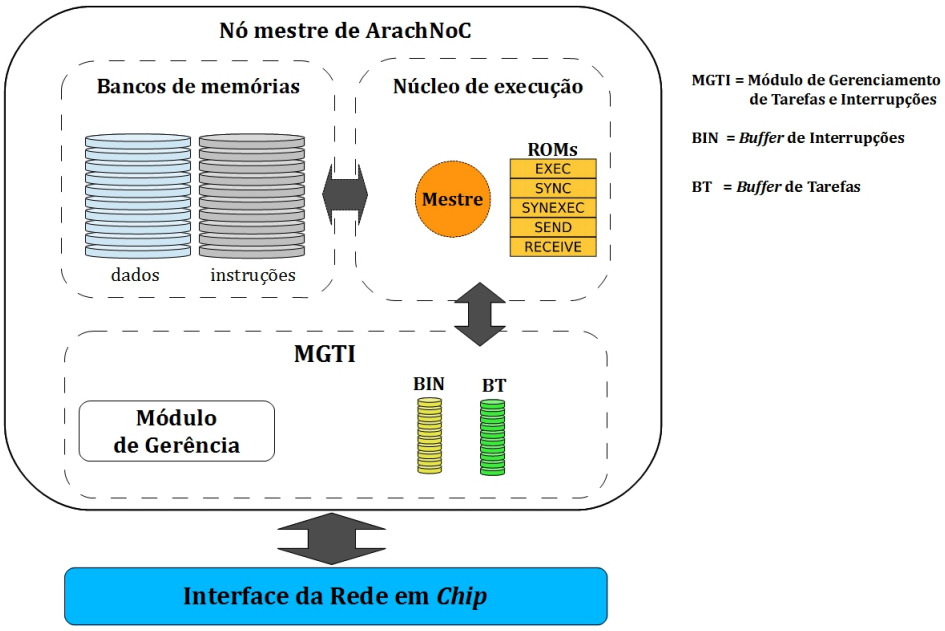


Figura 10 – Representação gráfica do nó mestre da plataforma ArachNoc.

O Nó Escravo

A principal função dos nós escravos é executar tarefas. Quando, durante a execução de uma tarefa, um dos oito núcleos de processamento encontrar uma instrução de sincronização ou comunicação, o mesmo envia uma interrupção ao MGTI, que armazena-a em um *buffer* e, assim que possível, a envia para o nó mestre. Existe um *buffer* para armazenamento de interrupções e outros oito para tarefas. Cada núcleo de execução é responsável por computar o conteúdo de um *buffer* de tarefas. O MGTI usa os oito *buffers* de tarefas para realizar a distribuição entre os núcleos. Assim, chegando uma tarefa ao nó escravo, o MGTI guarda-a no *buffer* de tarefas que possui menos tarefas.

A Figura 11 mostra um nó escravo com os oito núcleos de execução ligados as duas memórias; o MGTI gerencia os núcleos, os *buffers* e comunica-se com a interface de rede.

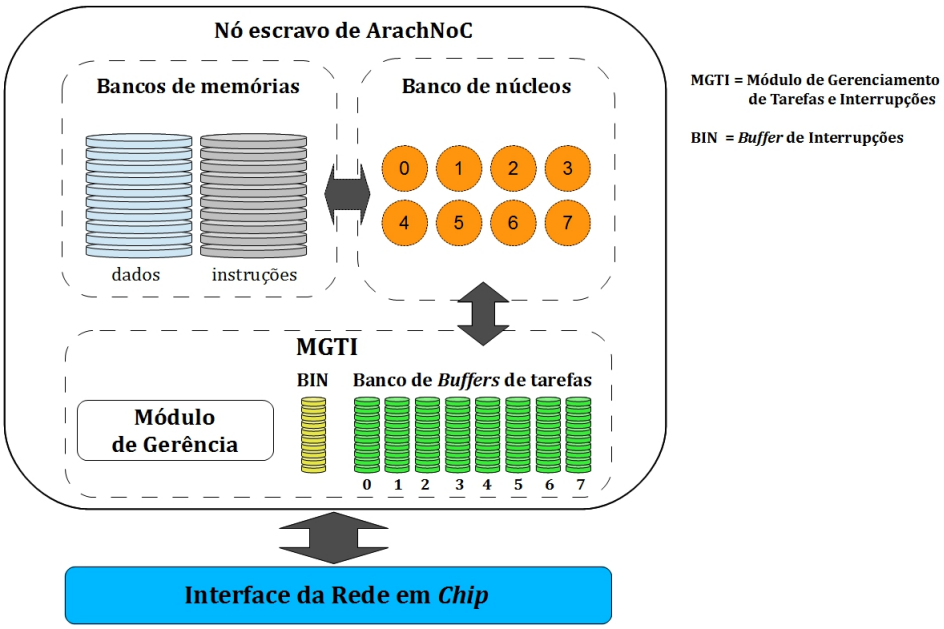


Figura 11 – Representação gráfica de um nó escravo da plataforma ArachNoc.

A seguir, são descritos os espaços de endereçamentos, o padrão de programação e o conjunto de instruções de sincronização e comunicação.

- ***Espaços de Endereçamentos***: Cada nó, possui um espaço de endereçamento privado e compartilhado entre os núcleos internos ao nó. Esse espaço de endereçamento consiste de duas memórias, sendo uma de instruções e outra de dados. As duas possuem 512 KB e, cada uma, são constituídas por 64 blocos de memória RAM.

- ***Instruções de Sincronização e Comunicação***: O sistema de comunicação e sincronização de ArachNoC permite a troca de dados e a sincronização. Esse sistema é composto pelas seguintes instruções:

  - ***Exec***: Instrução de sincronização assíncrona entre tarefas. A tarefa, que receber este sinal, está pronta para execução;
  - ***Sync***: Sinal de sincronização. A tarefa sinaliza que alcançou o ponto de sincronização;
  - ***SynExec***: Instrução de sincronização síncrona. A tarefa, que receber este sinal, deve aguardar um ou mais sinais de sincronização emitido por outras tarefas;
  - ***Send***: Instrução utilizada no envio de dados;
  - ***Receive***: Instrução utilizada no recebimento de dados.

- ***Padrão de Programação***: Esse padrão é constituído de uma tarefa inicializadora que é dividida (*fork*) em várias tarefas paralelas. No final da aplicação, esse padrão converge para uma única tarefa (*join*). O padrão de programação *Fork-Join* é ilustrado na Figura 12.

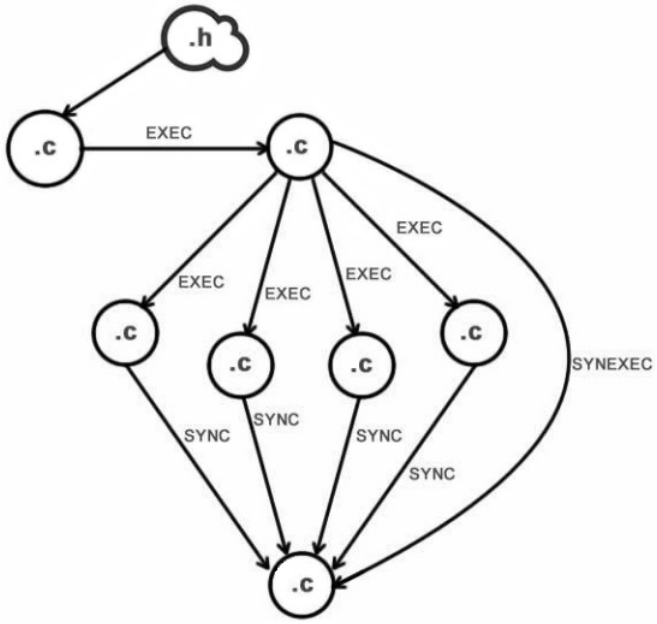


Figura 12 – Representação do padrão de programação para ArachNoC.

ArachNoC possui um ambiente de programação paralela, biblioteca para o gerenciamento de *threads* e interrupções; e um copilador cruzado *GCC to ArachNoC*. Entretanto, nesse ambiente de programação, o desenvolvedor deve dominar detalhes do *hardware*, por exemplo, definir endereços de *threads* e manejar instruções de baixo-nível como os sinais de sincronização.

1.3 Infra-estrutura do compilador LLVM

LLVM é uma infra-instrutura de compilador *open-source* escrita em C++. Iniciado como um projeto de pesquisa da Universidade de Illinois EUA. Hoje conta com um

número significativo de contribuições internacionais. Inicialmente denominado *Low Level Virtual Machine*, consiste de módulos reutilizáveis para implementação de compiladores e de ferramenta de análise e otimização de código executável (LLVM, 2015a)(LATTNER, 2008a).

A Figura 13 exibe os principais módulos do projeto LLVM. Esses módulos implementam as três fases clássicas de um compilador: *frontend*, otimização e *backend*. A divisão em três fases torna possível a criação de um novo módulo sem ter que reescrever todo o compilador.

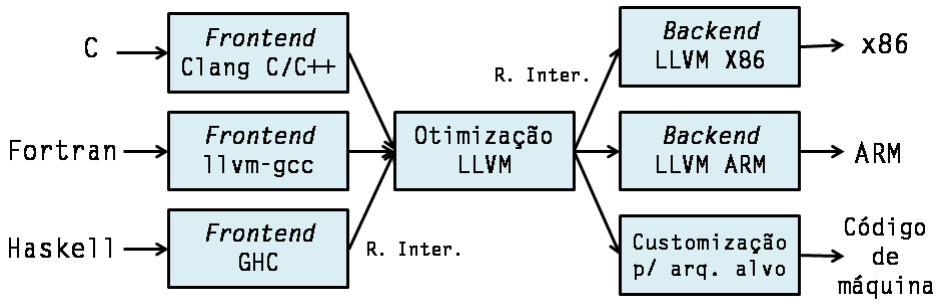


Figura 13 – Estrutura do LLVM com as três fases clássicas de um compilador.

Fonte: Kolek et al. (2013)

1.3.1 *Frontend* Clang

Clang (LATTNER, 2008b) é um *frontend* do projeto LLVM para as linguagens C, C++, Objective-C e Objective-C++. Ele possui um analisador de código fonte chamado *Clang Static Analyzer*, responsável por encontrar erros sintáticos e semânticos em programas durante a compilação. Clang foi concebido como uma plataforma *open-source* permitindo a comunidade estender o *frontend* para novas linguagens e novos recursos. Os desenvolvedores dessas extensões, podem propor a integração dos seus trabalhos com o projeto Clang/LLVM, beneficiando toda a comunidade.

1.3.2 Representação Intermediária

O compilador LLVM trabalha com uma representação intermediária do código fonte compilado. Com a adoção da representação intermediária, busca-se independência entre o *frontend* e o *backend*. Busca-se, também, suportar diversos tipos de análises e transformações.

A representação intermediária é uma linguagem parecida com uma linguagem *assembly*. *Assembly* é uma notação legível para humanos que representa um código de máquina.

1.3.3 *Backend* MIPS

A representação intermediária (RI) é a entrada do módulo *backend* LLVM que possui os componentes responsáveis por transformar a RI em código assembly ou código de máquina. A geração de código por um *backend* é um processo que consiste em muitas fases como: seleção de instruções, alocação de registradores e emissão de código.

A estrutura principal que o *backend* LLVM opera é um *directed acyclic graph* (DAG). A RI é transformada em um DAG correspondente, contendo as informações sobre as dependências entre funções e instruções da representação intermediária.

A ferramenta mais importante na criação de um *backend* LLVM é o TableGen (LLVM, 2015b) que é fornecida com o núcleo LLVM. Essa ferramenta analisa os arquivos com extensão .td, criados pelo desenvolvedor do *backend*. Estes arquivos são uma descrição da arquitetura alvo, onde o desenvolvedor instancia declarações como as descrições dos registradores e instruções da máquina alvo. Também, nestes arquivos .td, é possível realizar a definição de uma conversão de instruções da RI que não são suportadas na máquina alvo, em um conjunto de instruções equivalentes e suportadas. TableGen processa os arquivos .td e o resultado são arquivos em código C++, prontos para compilação. Esses arquivos gerados são utilizados pelas bibliotecas LLVM para geração da saída do *backend*. A Figura 14 exibe o processo (simplificado) de compilação da ferramenta TableGen para a arquitetura MIPS.

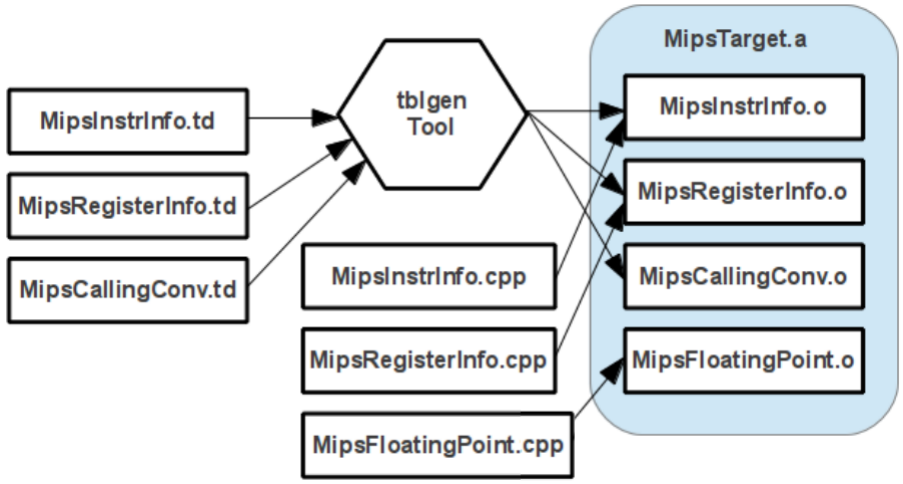


Figura 14 – Definição simplificada do alvo com TableGen.

Fonte: Kolek et al. (2013)

A Figura 15 mostra um trecho de um arquivo .td para o *backend* MIPS. Duas peças chaves na criação destes arquivos são: *class* e *def*. Ambos são tratados por TableGen como "registros". Cada registro deve ter um nome único, uma lista de parâmetros e podem possuir uma lista de super-classes. Por exemplo: *ArithLogicI* na linha 20 é uma super classe da definição ADDIU. Da linha 1 até a linha 6, são os parâmetros da classe *ArithLogicI*.

A conversão entre instruções da RI para instruções da máquina alvo também pode ser observada na Figura 15. Onde um nó em DAG, formado por uma instrução *add* em RI, é traduzido em outro nó DAG que representa uma instrução *addui* da máquina MIPS.

```
class ArithLogicI< string opstr,
                   Operand Od,
                   RegisterOperand RO,
                   InstrItinClass Itin = NoItinerary,
                   SDPatternOperator imm_type = null_frag,
                   SDPatternOperator OpNode = null_frag
                 >
    : InstSE< (outs RO:$rt),
              (ins RO:$rs, Od:$imm16),
              !strconcat(opstr, "\t$rt, $rs, $imm16"),
              [(set RO:$rt,(OpNode RO:$rs, imm_type:$imm16))],
              Itin,
              FrmI,
              opstr
            > {
        let isReMaterializable = 1;
        let TwoOperandAliasConstraint = "$rs = $rt";
}

def ADDIU : ArithLogicI<"addiu", simm16, GPR32Opnd, IIArith, immSExt16, add>,
                        ADDI_FM<0x9>;
```

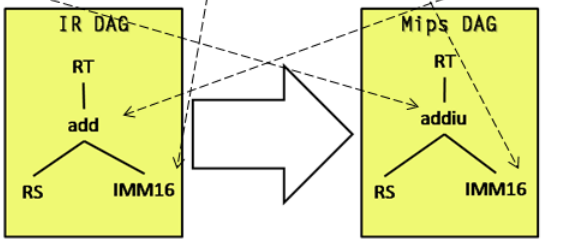


Figura 15 – Exemplo de um arquivo .td.

2 Estado da Arte

Este capítulo apresenta o que há de recente na criação de compiladores com a ferramenta LLVM. Antes desta dissertação, não existem compiladores OpenCL voltados para o desenvolvimento de código para as arquiteturas Metal e ArachNoC. As arquiteturas METAL e ArachNoC foram, também, desenvolvidas no escopo de duas dissertações de mestrado, iniciadas concomitantemente com esta. Deste modo, os compiladores são disponibilizados juntamente com os modelos simuláveis das arquiteturas. Buscou-se na literatura trabalhos voltados para o desenvolvimento de *backends* LLVM e o estado da arte no desenvolvimento de APIs (*Application Programming Interface*) para multicores.

2.1 *Backends* LLVM

Nesta seção apresenta-se o estado da arte no desenvolvimento de *backend* LLVM. Quando citadas pelos autores, discute-se as vantagens e desvantagens da utilização da infra-estrutura LLVM.

Trescastro em (Lopez Trescastro et al., 2015) apresenta os principais aspectos da concepção, validação e desenvolvimento de um *backend* LLVM para a próxima geração de processadores LEON (Cobham Gaisler, 2016). LEON é um processador RISC (do inglês, *Reduced Instruction Set Computer*). Cobham (Cobham, 2016) desenvolveu as arquiteturas LEON3 e LEON4, essas são modelos em VHDL de um processador de 32 bits compatível com a arquitetura SPARC V8. O modelo é altamente configurável e adequado para *system-on-chip* (SoC). Segundo os autores, o LLVM tem evoluído ao longo dos anos, tornando-se um compilador robusto, utilizado em projetos comerciais e acadêmicos. LLVM possui uma estrutura modular que permite a adaptação e reutilização dos módulos que o compõe. Nessa pequisa, os autores reutilizaram, do projeto LLVM, os módulos *frontend* e o de otimização, em conjunto com o novo *backend* desenvolvido. A Figura 16 mostra o novo *backend* adicionado ao projeto LLVM.

Em (KOLEK et al., 2013) é descrita a implementação de uma nova extensão do *backend* LLVM para a arquitetura microMIPS (Imagination Technologies LTD, 2016). MicroMIPS é uma das arquiteturas pertencentes à família de arquitetura MIPS. O processador microMIPS vem com conjunto de instruções de 16 e 32 bits. MicroMIPS é, como todas as arquiteturas de família MIPS, uma arquitetura RISC. A infraestrutura LLVM tornou-se popular devido ao seu projeto moderno e prático conjunto de ferramentas. Os autores ressaltam a reutilização de código do *backend* MIPS, onde, após a fase de seleção de instrução pelo *backend* MIPS, substituíram as instruções que o microMIPS não suporta por outras suportadas, em outras palavras, os códigos de operação (*opcode*) são substituídos.

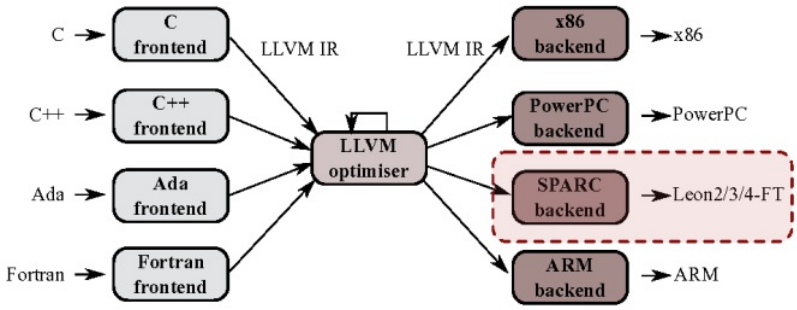


Figura 16 – Novo *backend* LEON.

Fonte: Lopez Trescastro et al. (2015)

Um novo *backend* para o compilador HiPE (*High Performance Erlang*), chamado ErLLVM, foi implementado utilizando a infraestrutura do projeto LLVM (SAGONAS; STAVRAKAKIS; TSIOURIS, 2012). HiPE é compilador da linguagem Erlang (ERICSSON, 2016). O HiPE possui *backends* para uma variedade de plataformas e cada uma requer manutenção e evolução em uma base de código de tamanho significativo. Sagonas, Stavrakakis e Tsiouris implementaram um *backend* utilizando uma infraestrutura mais moderna e popular. Os resultados indicam que o código gerado por ErLLVM é significativamente mais rápido, e em x86 e x86_64 atinge o mesmo desempenho que o *backend* existente do HiPE. Eles relatam duas desvantagens: maior tempo de compilação e a necessidade de uma nova versão do LLVM.

Stripf, Koenig, Rieder e Becker desenvolveram um *backend* LLVM (STRIPF et al., 2012) para a arquitetura reconfigurável KAHRISMA (KOENIG et al., 2010). A arquitetura KAHRISMA possui um *run-time* que configura a quantidade de recursos e a ISA (do inglês, *Instruction Set Architecture*), utilizadas pelo *software*. Esse *run-time*, chamado *Run-Time Scalable Issue-Width* (RSIW), muda as configurações da arquitetura com a finalidade de otimizar o uso de recursos, a performance e a energia consumida.

Terei e Chakravarty desenvolveram um novo *backend* para o *Glasgow Haskell Compiler* (GHC) (haskell.org, 2016) utilizando a infra-estrutura LLVM (TEREI; CHAKRAVARTY, 2010). GHC é um compilador de código aberto para a linguagem de programação Haskell (Haskell, 2016). GHC possuía dois *backends*: o primeiro traduz código *Spineless Tagless G-machine* (STG-machine) (JONES; SALKILD, 1989) para C e utiliza a infra-estrutura GNU C para gerar código de máquina; O segundo gera diretamente código assembly apenas para as arquitetura x86 e SPARC.

Terei e Chakravarty criaram, em termos de conceito e linhas de código, um *backend* mais simples do que os outros dois *backends*. Os resultados mostram que, além de ter um infra-estrutura moderna, o *backend* LLVM gera código com uma vantagem de desempenho com relação ao *backend* para as arquiteturas x86 e SPARC. A desvantagem está no tempo de compilação em comparação com o *backend* x86 e SPARC. Isso é, em parte, esperado com

o *backend* LLVM, pois esse realiza mais otimizações de código. A adoção da infraestrutura LLVM tornou possível uma evolução continua do *backend* GHC através comunidade LLVM.

LLVM é uma infraestrutura de compilador modera, prática e reutilizável. Adotar esta estrutura significa, além da redução do esforço no desenvolvimento, obter o apoio de toda a comunidade LLVM. A padronização do projeto LLVM permite que um módulo futuramente desenvolvido pela comunidade, possam ser adotados por outros projetos. Por exemplo, um novo *frontend* pode ser utilizado por *backends* pré-existentes.

2.2 Implementações da API OpenCL

A *Portable Computing Language* (POCL) foi implementada com o objetivo de fornecer uma implementação do padrão OpenCL, sendo uma API portável, tanto em termos de arquitetura quanto em otimizações (JääSKELäINEN et al., 2015). O objetivo, além de fornecer uma implementação da API OpenCL, é explorar o paralelismos de dados/instruções sobre diversas plataformas que possuem as mais variadas particularidades. Para avaliar o desempenho da implementação em POCL, foi utilizado o conjunto de exemplos de aplicações disponíveis no *AMD Accelerated Parallel Processing Software Development Kit* (AMD, 2012). O benchmark foi executado em várias plataformas suportadas por POCL. A fim de comparar POCL com a melhor implementação conhecida para o *hardware*, também foi executado, sem modificar o benchmark, na implementação da API OpenCL disponibilizada pelo fornecedor da plataforma em teste. Os resultados mostram que a maioria das aplicações, quando compilada usando POCL, foram mais rápidas ou tão quanto a API OpenCL disponibilizada pelo fornecedor da plataforma em questão.

Em (NAH et al., 2013) são apresentadas técnicas de otimizações para um compilador que tem como arquitetura alvo um processador reconfigurável. A arquitetura consiste em um processador de propósito geral, e um acelerador reconfigurável, embarcado e com unidades vetoriais. O acelerador é capaz de alternar sua arquitetura entre os modos VLIW (*Very Large Instruction Word*) e CGRA (*Coarse Grained Reconfigurable Array*). Um grande problema desta arquitetura é a dificuldade de programação, e OpenCL apresentou-se como uma boa alternativa para resolver esse problema. Entretanto, uma vez que OpenCL não garante portabilidade de desempenho, as otimizações particulares a cada arquitetura ainda são necessárias. Portanto, os autores desenvolveram um *framework* de compilação OpenCL que explora a capacidade que a arquitetura alvo tem de alternar entre os modos VLIW e CGRA, além de explorar também a unidade vetorial.

Outro *framework* para compilação OpenCL é apresentado em Li et al. (2012). Os autores possuem como arquitetura alvo um sistema DSP (*Digital Signal Processing*) *multicore* e embarcado, que consistem em um processador ARM e um subsistema com múltiplos DSPs. Para gerar código eficiente para tais DSP, o compilador tem que con-

siderar os acessos irregulares ao banco de registradores em muitas fases de otimização. Os experimentos apresentados reportam uma média de 29% de melhora no desempenho, utilizando um das técnicas de otimização, e uma aceleração de 2x quando comparado o uso de duas DSPs e um único ARM.

Em (LEE et al., 2011) é apresentado o projeto de um *framework* OpenCL (compilador e sistema de tempo de execução) para arquiteturas *manycore* sem coerência de *cache* implementada em *hardware*. Como arquitetura alvo, os autores utilizaram um processador da Intel chamado de *Single-chip Cloud Computer* (SCC). Esse é um processador experimental, criado pela *Intel Labs*, que contém 48 núcleos em um único *chip* cada um com sua própria *cache* L1 e L2 sem suporte em *hardware* para coerência. O processador permite o uso de uma memória externa de no máximo 64GB, que pode ser acessada por todos os núcleos e cada núcleo mapeia esta memória externa para seu próprio espaço de endereçamento. Os autores argumentam que a consistência e coerência do **modelo memória OpenCL** coincide com a arquitetura SCC. O fraco modelo de memória do OpenCL requer uma relativa pequena quantidade de mensagem e ações de coerência para garantir coerência e consistência entre os blocos de memória do SCC. O mecanismo de mapeamento de memória dinâmico permite que o *framework* apresentado preserve a semântica nas operações sobre os objetos *buffers* do OpenCL sem muita sobrecarga.

O padrão OpenCL foi criado para unificar em um único paradigma e conjunto de ferramentas, o desenvolvimento de soluções de computação paralela para dispositivos de naturezas distintas. Uma das vantagens da API OpenCL é oferecer uma interface que abstraia as particularidades de cada arquitetura. Entretanto, OpenCL não garante a portabilidade de desempenho, a eficiência depende da implementação de cada fornecedor de *hardware*.

3 Compiladores Clem e Ocean

Nos últimos anos, a evolução dos processadores têm levado os sistemas computacionais a possuir uma estrutura cada vez mais heterogênea (NUGTEREN; CORPORAAL, 2014). Já são comuns os sistemas computacionais heterogêneos dotados de paralelismo. A utilização de CPUs *multicore* em conjunto com GPUs leva o desenvolvedor a possuir conhecimento de uma variedade maior de arquiteturas para produzir códigos *multi-threads* eficientes. Neste contexto, é fundamental a evolução das linguagens e ferramentas de desenvolvimento que apoiem os desenvolvedores de softwares, pois elas (linguagens e ferramentas) propiciam maior produtividade e qualidade. Assim, neste trabalho de dissertação, desenvolveu-se duas ferramentas (distintas) para compilação de códigos escritos em linguagem de alto nível (*OpenCL/C*), para as arquiteturas METAL e ArachNoC. Tais arquiteturas foram desenvolvidas no escopo das pesquisas desenvolvidas no grupo de pesquisa em circuitos e sistemas embarcados (do inglês, CESLa, *Circuits and Embedded Systems Laboratory*), da Universidade Federal do Piauí (UFPI). Como apresentado na Seção 1.2, a principal característica destas arquiteturas é o paralelismo massivo, obtido da integração de múltiplos nós *multicore* de processamento.

Desenvolver aplicações paralelas é um desafio para os desenvolvedores de software (NUGTEREN; CORPORAAL, 2014), não somente pela necessidade de lidar com uma variedade maior de hardware e suas instruções de baixo nível, mas também pela necessidade de otimizações no gerenciamento de suas *threads* e recursos de hardware. Daí a importância da evolução dos compiladores em conjunto com os ambientes de execução para gerenciar de forma eficiente a execução e alocação dos recursos de *hardware* (PAONE, 2014).

Este trabalho descreve a construção de compiladores para compilação de código em linguagem de alto nível. Espera-se que com os compiladores, as arquiteturas alvo ganhem maior visibilidade como uma opção viável para o desenvolvimento de sistemas computacionais.

Uma versão anterior do compilador CLEM e da arquitetura METAL foi publicada em Nepomuceno et al. (2015). Neste capítulo descreve-se as ferramentas CLEM e OCEAN de modo conjunto, uma vez que apresentam similaridades em suas estruturas, pois utilizam a infraestrutura do LLVM. O que diferenciam os dois compiladores são o conjunto de definições de baixo nível e a estrutura da hierarquia de memória que as duas arquiteturas apresentam. Durante o texto, a medida do necessário, serão destacadas as peculiaridades de cada arquitetura e o impacto no *backend* do respectivo compilador.

3.1 Arquitetura do compilador

Inicialmente, antes que se apresentem os compiladores, faz-se necessário explicar que as arquiteturas alvo, apresentam-se atualmente como plataformas virtuais. Isto quer dizer que a execução dos códigos compilados é simulado sobre uma descrição funcional do hardware. Esta descrição foi implementada utilizando a biblioteca SystemC (Institute of Electrical and Electronics Engineers Inc., 2006). SystemC é um conjunto de classes em C++ usada para descrição de *hardware*. A biblioteca possui macros que fornecem uma interface de simulação orientada a eventos.

O processo de compilação pode ser observado na Figura 17. O desenvolvedor cria suas aplicações com o *framework OpenCL*, cuja API foi desenvolvida no escopo deste trabalho. Essas aplicações são compostas de um arquivo com a função *kernel*, para execução nos núcleos escravos, e um ou mais arquivos escritos em C/C++, para execução no núcleo mestre. Os blocos *Clang*, *Optimizer* e *MIPS Backend* são módulos da ferramenta LLVM, respectivamente, um *frontend*, módulos de otimizações e um *backend MIPS*. Esses são módulos já implementados e disponibilizados na ferramenta LLVM.

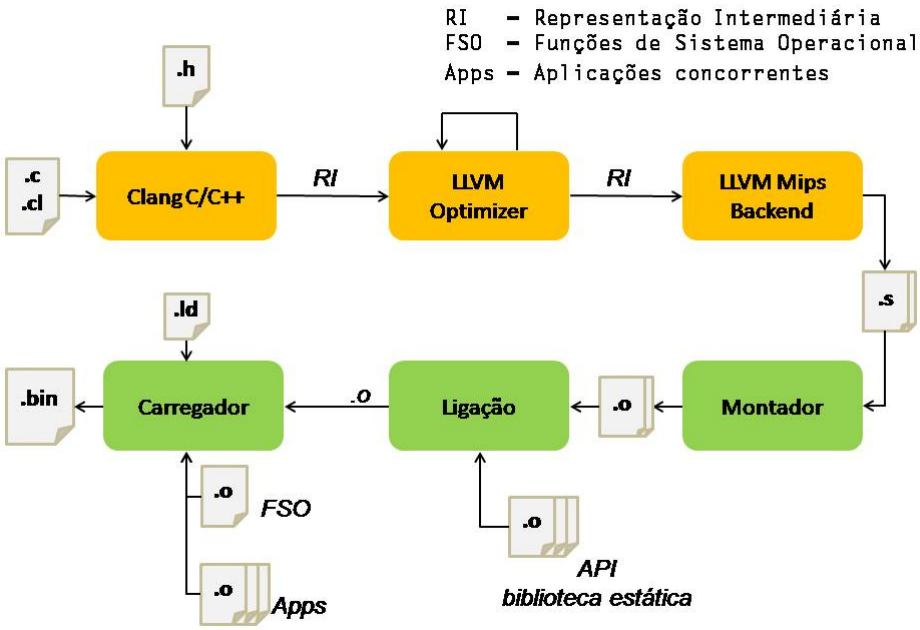


Figura 17 – Processo de compilação.

O desenvolvedor implementa o *kernel* de sua aplicação em um arquivo separado com a extensão .cl, isso se deve ao fato de ser utilizado a compilação *offline* (TSUCHIYAMA et al., 2010). A compilação *online* necessita de um compilador em tempo de execução para a função *kernel*, o que, no escopo de desenvolvimento do projeto no qual esta dissertação se insere, não é possível devido a inexistência de um ambiente de simulação apropriado. Assim, é adotada a compilação *offline*. Esse método de compilação foi incorporado à especificação

do OpenCL para dar suporte aos sistemas embarcados que não dispõem de muitos recursos de *hardware*. Como já explicado, as arquiteturas propostas possuem recursos de *hardware* suficientes para compilação *online*, entretanto, com o desenvolvimento destas arquitetura em dissertações de mestrado que acontecem paralelamente a esta, não foi possível optar pela compilação *online*.

O bloco Montador traduz um código fonte escrito em linguagem de montagem ou *assembly* para um arquivo objeto contendo binários da máquina alvo. A Figura 18 exibe um exemplo de um arquivo *assembly* para uma arquitetura baseada no MIPS (FARQUHAR; BUNCE, 1993).

```
    .text
    .globl  vecAdd
    .align  2
    .type   vecAdd,@function
    .ent    vecAdd
vecAdd:                                 # @vecAdd
    addiu   $sp, $sp, -48
    sw  $ra, 44($sp)            # 4-byte Folded Spill
    sw  $fp, 40($sp)            # 4-byte Folded Spill
    move     $fp, $sp
    sw  $4, 36($fp)
    sw  $5, 32($fp)
    addiu   $4, $zero, 0
    jal get_global_id
    nop
    sw  $2, 20($fp)
    lw  $4, 24($fp)
    sltu    $2, $2, $4
    move     $sp, $fp
    lw  $fp, 40($sp)            # 4-byte Folded Reload
    lw  $ra, 44($sp)            # 4-byte Folded Reload
    addiu   $sp, $sp, 48
    jr  $ra
    nop
    .set    at
    .set    macro
    .set    reorder
    .end    vecAdd
$func_end1:
    .size   vecAdd, ($func_end1)-vecAdd
    .ident  "clang version 3.7.1 (tags/RELEASE_371/final)"
    .ident  "clang version 3.7.1 (tags/RELEASE_371/final)"
    .section    ".note.GNU-stack","",@progbits
    .text
```

Figura 18 – Exemplo de um arquivo *assembly*.

Já o bloco Ligação realiza o processo de *link* entre um conjunto de arquivos objetos e produz como resultado um arquivo objeto. Esses arquivos objetos podem ser, além da aplicação desenvolvida, uma biblioteca que contém funções para serem adicionadas ao código.

A biblioteca estática contém as funções da *API OpenCL* implementadas para a arquitetura alvo. O desenvolvedor cria sua aplicação chamando métodos da *API OpenCL*, a fim de abstrair as particularidades da máquina alvo. A implementação da API para a arquitetura alvo é descrita na Seção 3.9.

O bloco Carregador é responsável por alocar a aplicação na memória para execução. Em uma máquina real, quando um arquivo executável é passado para o carregador, existe uma comunicação entre o sistema operacional (SO) e o *software* carregador, com a finalidade de buscar um espaço de endereçamento para a nova aplicação (TANENBAUM, 2003). A implementação do bloco Carregador tem uma pequena alteração devido as características dos ambientes de execução das duas arquiteturas alvo. O carregador desenvolvido também aloca na memória de execução as funções de um SO e as aplicações desenvolvidas para execução.

Um *software* SO pode ser definido como uma máquina estendida e um gerenciador de recursos, essas são suas duas principais funções. Assim, a primeira função esconde toda complexidade do *hardware*, enquanto a outra gerencia a alocação de memória, núcleos de execução, tempo de execução, entre outras gerências. O SO desenvolvido consiste em um pequeno conjunto de funções, desenvolvidas na linguagem C, que viabiliza a execução de aplicações concorrentes. Dessa forma permite o controle da alocação de núcleos de execução; a criação e finalização de processos; e a alocação de memória em tempo de execução.

A saída do Carregador são arquivos que representam os conteúdos das memórias das arquiteturas alvos. A simulação em SystemC na arquitetura METAL tem como entrada dois arquivos: um com o conteúdo da memória de dados e outro da memória de instrução. Já a arquitetura ArachNoC pode requerer a geração de até 64 arquivos de instruções e 64 arquivos de dados. Entretanto, nem todo processo de compilação requer todos estes arquivos, em uma compilação são gerados apenas os arquivos correspondentes às memórias que alocaram dados ou instruções no processo de compilação.

O arquivo .ld é um *scrip* que descreve os arquivos de saída, as partições de memórias e as seções de memória. A Figura 19 exibe um exemplo desse arquivo. As três principais partes do *script* são:

- **MEMORY**: A *tag* MEMORY, na linha 1, define as regiões de memória da arquitetura. Pode ser definida uma memória por completo, parte de uma ou com subdivisões. Por exemplo, considere uma memória de 4096 palavras, o desenvolvedor pode definir duas regiões: uma iniciando em zero com tamanho 1024 e a outra iniciando em 1024 com tamanho 3072. ORIGIN (linhas 2 e 3) defini-se o endereço de início dessa representação de memória, ou seja, quando for zero o arquivo de saída representa um segmento memória que começa no endereço de memória zero. LENGTH (linha 2

```
1  MEMORY {
2      INST_BLOCK    : ORIGIN = 0,      LENGTH = 2048;
3      DATA_BLOCK    : ORIGIN = 0,      LENGTH = 2048;
4  }
5
6  SECTIONS {
7      INST_BLOCK < {
8      *(.bootSO)
9      *(.launchProcess)
10     *(.kernelProgram)
11     *(.text)
12     }
13     DATA_BLOCK < {
14     *(.data)
15     *(.rodata)
16     *(.bss)
17     *(COMMON)
18     *(.scommon)
19     }
20 }
21
22 FILE [Instruction.mif]( ORIGIN = 0, LENGTH = 4096) {
23     INST_BLOCK;
24 }
25 FILE [Data.mif]( ORIGIN = 0, LENGTH = 4096) {
26     DATA_BLOCK;
27 }
```

Figura 19 – Exemplo de um *script* de configuração do Carregador.

e 3) recebe como parâmetro a quantidade máxima de palavras suportada por essa representação de memória. O *script* deve conter apenas uma única *tag* MEMORY.

- **SECTIONS**: Define em qual região de memória as seções devem ser alocadas. O desenvolvedor cria novas seções invocando o comando *.section* (BARR; MASSA, 2006). Assim, durante a fase de ligação é criada uma nova seção, que pode ser realocada no arquivo de saída do Carregador. Também é obrigatório definir apenas uma SECTIONS.

- **FILE**: Deve existir ao menos uma *tag* FILE. Um FILE registra o conteúdo e as informações de um único arquivo de saída. O nome do arquivo é definido no parâmetro entre colchetes (linhas 22 e 25). As funções de ORIGIN e LENGTH (linha 22 e 25) são as mesmas citadas em MEMORY. Entre chaves (linhas 23 e 26) são listadas as MEMORY's que serão destinadas ao arquivo de saída definido.

O Carregador foi criado de forma genérica para atender as duas arquiteturas METAL e ArachNoC. O *script* de configuração permite configurar a saída do Carregador de forma a atender as particularidades de cada arquitetura.

3.2 Sistema em camadas

Um sistema modularizado em camadas consiste na separação de funcionalidades em camadas. Cada camada é formada por um conjunto de funções que provê recursos para a camada de cima e utiliza as funções da camada inferior. A Figura 20 apresenta o sistema desenvolvido em camadas. A base é composta pelo ***Hardware*** (Metal ou ArachNoC). Acima da camada de *hardware* está a camada **Funções de *Hardware***. Ela possui funções que abstraem as particularidades do *hardware* para a camada superior. A camada seguinte é composta das funções do **Sistema de Tempo de Execução** e **Funções de Sistema Operacional**. A implementação da **API Opencl** encontra-se na camada seguinte. A última camada é composta pelo código OpenCL criado por um programador OpenCL.

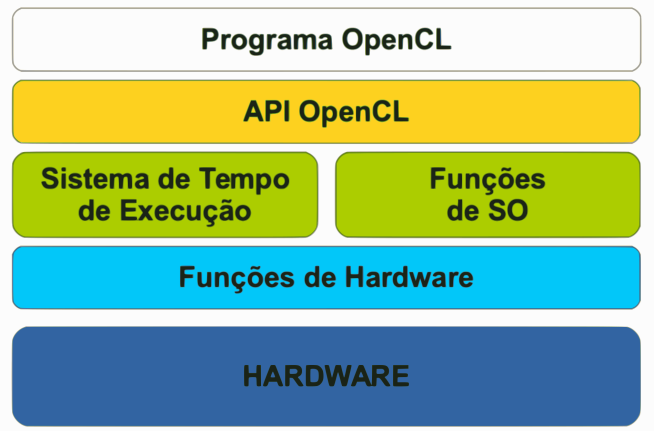


Figura 20 – Sistema em camadas.

As funções de hardware, funções de SO e funções do sistema de tempo de execução serão descritas nas seções seguintes.

3.3 Funções de *hardware*

Nesta seção serão apresentadas as funções desenvolvidas para desempenharem o pepel de interface do *hardware*. Toda comunicação com o *hardware* ocorre através destas funções. Foram desenvolvidas para abstrair/simplificar a comunicação do *software* em execução com a plataforma de hardware. Em alguns casos, agruparam-se em uma só função várias instruções da plataforma.

Plataforma METAL

São listadas as funções de *hardware* da plataforma METAL. As seguintes funções são executadas apenas pelo nó mestre de Metal.

- *___setMemWriteReadBuffer()*: Esta função configura o envio de dados entre o nó mestre e os nós escravos. Ela recebe como parâmetro o endereço de origem, o de

destino, a quantidade dados e a direção da transferência de dados (mestre para escravo ou escravo para mestre).

- *___setKernelSlave*: Comanda o envio de código executável para a memória de execução dos nós escravos. Transfere a partir da memória de dados do nó mestre. Ela recebe como parâmetro o endereço de origem e a quantidade de dados.

- *___createNDRange()*: É utilizada para configurar o módulo gerenciador de tarefas no nó mestre. Ela recebe como parâmetros a quantidade de *work-items* total da aplicação, a quantidade de *work-group*, a quantidade de *work-items* dentro de *work-group* e as dimensão do espaço de endereçamento da aplicação. A plataforma METAL, após ser autorizada pela aplicação em execução, envia um *work-group* para um nó escravo (que esteja livre) processar, até que todos *work-groups* sejam enviados.

- *___beginExec()*: Dispara a execução na plataforma. A execução ocorre seguindo as informações passadas ao *hardware*.

Foram implementadas as funções de *hardware* que são executadas apenas pelos nós escravos de Metal. Essas funções são o meio pelo qual as tarefas obtêm seus ids. O módulo escalonador detêm o controle e os ids das tarefas em execução em cada nó escravo e, da mesma forma, em cada um dos 8 núcleos de execução dentro de um nó escravo. Assim, para solicitar seus ids, as tarefas em execução utilizam as seguintes funções de *harware*.

- *___sys_get_global_id()*: Considerando o espaço de índices (NDRange) do modelo de execução OpenCL, essa função retorna o id global da tarefa. Esse id refere-se à posição da tarefa no espaço de índices global.

- *___sys_get_local_id()*: Retorna o id local da tarefa. Esse id refere-se à posição da tarefa no espaço de índices dentro do grupo.

- *___sys_get_group_id()*: Uma tarefa pode solicitar o id do grupo dentro do espaço de índices. Essa função retorna esses ids.

- *___sys_barrier()*: Informa ao escalonador que a tarefa atingiu uma barreira. O escalonador atribui, caso exista, outra tarefa para execução. Após toda as tarefas em um grupo atingirem a barreira, o escalonador re-escalona todas as tarefas do grupo para execução.

Plataforma ArachNoC

Todas as seguintes funções geram uma interrupção para o mestre tratar, exceto a função *___rfe()*. São descritas as funções de *hardware* para ArachNoC.

- *___target_node()*: Usada para definir para qual nó escravo uma tarefa deverá ser encaminhada para computar.

- *___exec()*: Função utilizada para envio de um comando de execução. Ela recebe quatro parâmetros: o id da aplicação origem, o id da tarefa origem, o id da aplicação destino e o id da tarefa destino.

- *___synexec()*: Sinaliza para uma tarefa que ela esta apta para executar, mas ainda deve aguardar todos os sinais de sinc pre-definidos. Ela recebe quatro parâmetros: o id da aplicação origem, o id da tarefa origem, o id da aplicação destino e o id da tarefa destino.

- *___sync()*: Sinaliza para uma tarefa que já atingiu o ponto de sincronização. Os parâmetros dessa função são: o id da aplicação origem, o id da tarefa origem, o id da aplicação destino e o id da tarefa destino.

- *___send()*: Função utilizada para envio de dados entre tarefas. Os parâmetros são: o id da aplicação destino, o id da tarefa destino, o endereço dos dados e tamanho dos dados.

- *___receive()*: Todo send possui um receive correspondente. Essa função sinaliza a espera de um dado. Quando uma tarefa executa um receive, ela é bloqueada até a chegada do send correspondente. As funções send's e receive's geram interrupções. O nó mestre coordena a transferência de dados e realiza o desbloqueio da tarefa que executou o receive.

- *___rfe()*: Função executada por todas as tarefas. Ela sinaliza o término da tarefa para o Módulo de Gerenciamento de Tarefas e Interrupções (MGTI), localizado dentro do mesmo nó da tarefa.

3.4 Funções de SO

Esta seção discute sobre a implementação de funções para viabilizar a execução concorrente de aplicações nas plataformas.

Plataforma METAL

A plataforma METAL, conforme discutido na Seção 3.3, recebe da aplicação uma configuração e, em seguida, um comando para iniciar a execução. Nesse momento, o *hardware* bloqueia o nó mestre até o término das execuções nos nós escravos, ou seja, uma aplicação ocupa o nó mestre durante todo o seu processamento. Em futura versão da plataforma METAL será oferecido suporte às execuções de aplicações concorrentemente.

A implementação de funções para gerenciamento de múltiplas aplicações será realizada em trabalhos futuros. As funções de SO para ArachNoC, descritas a seguir, podem ser adaptadas para o gerenciamento de múltiplas aplicações em METAL.

Plataforma ArachNoC

As funções a seguir estão disponíveis apenas para o nó mestre. São responsáveis por coordenar o início das aplicações, criar novas tarefas e alocar memória em tempo de execução.

- *loop_so()*: Essa função coordena o início das execuções das aplicações. Ela busca, em um descritor de aplicações, uma aplicação pronta para executar e, em seguida, disponibiliza o núcleo de execução para a aplicação. Quando o processo inicial da aplicação finalizar, o fluxo de execução volta para loop_so() que, por sua vez, volta a consultar o descritor.

- *create_process()*: Uma aplicação em execução cria tarefas utilizando essa função. Essa função recebe três parâmetros: o endereço inicial da tarefa, o tamanho do código da tarefa e um ponteiro para uma lista de sync's. Esse ponteiro sendo nulo, significa que a tarefa não aguardará sinais de sync's.

- *malloc_bytes()*: Utilizada na locação de memória em tempo de execução.

3.5 Sistema de Tempo de Execução

Aqui serão descritas as funções de gerenciamento de memória e passagem de parâmetros das tarefas em execução.

Plataforma METAL e ArachNoC

- ***Passagem de parâmetros para tarefas***: Foi implementada a função *set_Param()*. Ela é utilizada pelo nó mestre para definir os parâmetros do *kernel*.

  Kernel é o trecho de código executável que deve ser enviado aos escravos. A *set_Param()* adiciona linhas de código no início do *kernel*, que definem os parâmetros necessários. Assim, o nó mestre envia o código executável ao nós escravos e os parâmetro junto ao código.

- ***Gerenciamento de memórias***: Foram desenvolvidas as funções *metal_NDRange()* e *arach_NDRange()* para gerenciamento de memória nas plataformas METAL e ArachNoC, respectivamente.

Em cada nó escravo, tanto em METAL quanto em ArachNoC, existe uma memória de dados compartilhada. Essa memória deve ser dividida em espaços de endereçamentos global, local e privado. Esses espaços são o modelo de memória OpenCL definidos na Seção 1.1.3.

A divisão ocorre da seguinte forma: a primeira quarta parte da memória é para endereçamento global, a segunda para endereçamentos locais e as outras duas partes são para endereçamento privado.

Existe ainda uma outra divisão dessa seção de memória privada. O endereçamento privado é dividido entre os *work-items* de um *work-group*. Deve-se garantir que todos os *work-items* de um *work-group* sejam enviados para o mesmo nó escravo. A Figura 21 exibe a divisão da memória de dados em global, local e privado. A região privada é subdividida em seções de memórias privadas que, cada seção, pertence a um *work-item*. Esses *work-items* estão dentro de um único *work-group*.

Cada *work-item* utiliza sua fatia de memória privada para alocar sua pilha de execução. As funções *metal_NDRange()* e *arach_NDRange()*, após a divisão da memória privada, definem o endereço base da pilha de execução de cada *work-item*. Essas funções adicionam linhas de código no início do *kernel* que atualiza apontador de pilha para a fatia de endereçamento privado do work-item em execução. O nó mestre deve chamar essas funções antes de enviar o *kernel* ao nós escravos.

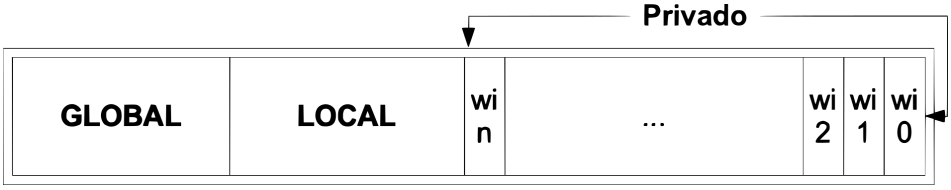


Figura 21 – Divisão da memória de dados em tempo de execução.

- ***Retorno dos resultados da computação***: Em ArachNoC, cria-se uma tarefa extra para cada *work-group*. Essa tarefa é responsável pelo retorno dos resultados computados em cada nó escravo. Ela inicia sua execução após receber um sinal de sync de todas as tarefas do *work-group*.

3.6 Chamadas de funções

As hierarquias nas chamadas de funções, em Metal e ArachNoC, são apresentadas nas Figuras 22 e 23 respectivamente. Na API OpenCL, a função **clFinish()** é responsável pela chamadas das funções do sistema de tempo de execução e das funções de *hardware*. Ela (*clFinish()*) detêm toda configuração da aplicação OpenCL e fica responsável por passar essas configurações para as funções abaixo na hierarquia no sistema em camadas.

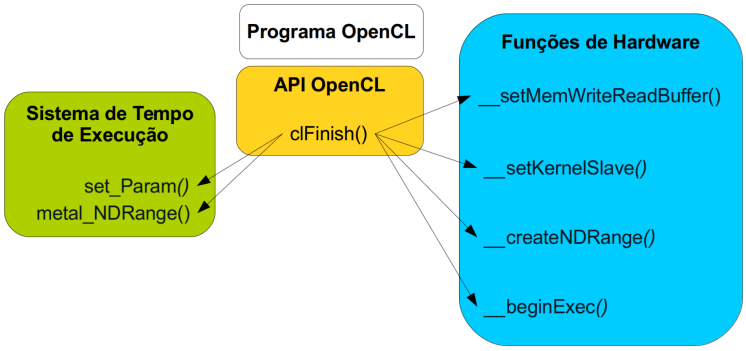


Figura 22 – Chamadas de funções na plataforma Metal.

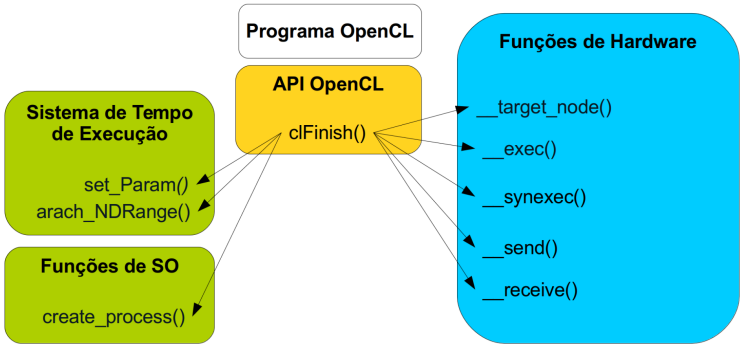


Figura 23 – Chamada de funções na plataforma ArachNoC.

3.7 Execução de uma aplicação na plataforma Metal

Para execução na plataforma METAL, a aplicação deve realizar os seguintes passos:

1. Configurar o envio dos dados necessários para computação nos nós escravos;
2. Configurar o envio do código executável aos nós escravos;
3. Configurar o gerenciador de tarefas e, finalmente, dispara a execução;
4. Após finalizarem todas as execuções, a aplicação solicita a volta dos resultados utilizando a função *__setMemWriteReadBuffer()*;
5. Fim da execução.

A aplicação não tem controle de quais nós escravos vão ser utilizados, esse controle é realizado pela plataforma. A Figura 24 mostra uma aplicação OpenCL em execução na plataforma METAL. Essa possui um processo *host* no nó mestre e dois *work-group* ocupando dois nós escravos. Os id's são fornecidos pela plataforma ao software em execução.

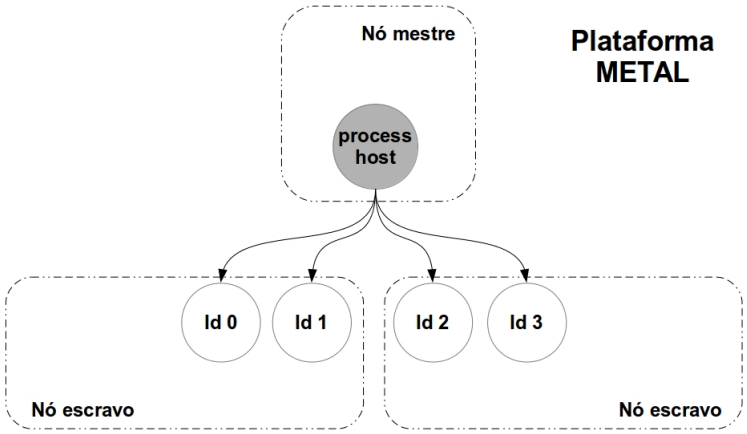


Figura 24 – Execução na plataforma METAL.

3.8 Execução de uma aplicação na plataforma ArachNoC

Para execução na plataforma ArachNoC, a aplicação, baseada em OpenCL, deve realizar os seguintes passos:

1. O nó mestre cria as tarefas para computação nos nós escravos;
2. Executa **send's** para que o nó mestre transfira dados aos nós escravos;
3. Dispara as execuções das tarefas utilizando a função **exec**;
4. Executa **receive** para os resultados nos escravos;
5. Emite um **synexec** para sincronizar a volta dos dados em cada nó escravo. Esse sinal é direcionado à tarefa que possui a função de retornar os resultados em cada nó escravo;
6. A tarefa inicial da aplicação finaliza, mas as demais continuam em execução nos nós escravos;
7. Quando todas as interrupções estiverem tratadas e as tarefas finalizarem, então, ocorre o fim da aplicação.

A Figura 25 exibe o exemplo de uma aplicação, baseada em OpenCL, mapeada para a plataforma ArachNoC. Os id's são os valores fornecidos pela plataforma ao *software* em execução. A função *loop_so()* chama o processo inicial de uma aplicação. Essa aplicação possui dois *work-groups* com três *work-items*. Cada *work-group* é enviado para um nó escravo computar.

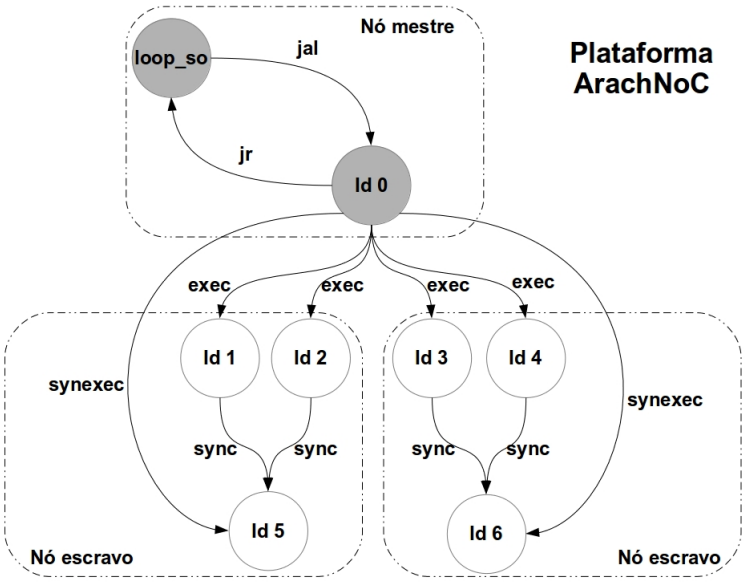


Figura 25 – Execução na plataforma ArachNoC.

3.9 API OpenCL

A *API OpenCL* é uma implementação da especificação do padrão OpenCL. Essa especificação é mantida pelo *Khronos Group* que cria padrões para criação e aceleração de gráficos, mídia interativa e computação paralela, em uma grande variedade de plataformas. Algumas características do padrão OpenCL são: suporte a modelos de programação paralela baseada em tarefas e dados; utiliza um subconjunto da ISO C99 (open-std.org, 2003) com extensões para paralelização; e define configurações para dispositivos embarcados. O grupo *Khronos* fornece, além da especificação do núcleo da API, uma descrição da linguagem C suportada pelo kernel OpenCL. Os benefícios da padronização proposto pelo padrão OpenCL são claros, os fornecedores de *hardware* podem oferecer suporte ao desenvolvimento de aplicações que seguem esse padrão, portanto, reduz o esforço em portabilidade do código fonte (JääSKELäINEN et al., 2015).

Serão apresentadas as funcionalidades implementadas da especificação da API OpenCL para as duas arquiteturas METAL e ArachNOC. São duas implementações distintas do mesmo subconjunto da API OpenCL. À medida que as funcionalidades forem apresentadas, será destacada as particularidade de cada arquitetura no contexto da implementação, também os problemas e soluções desenvolvidas.

3.9.1 API's para METAL e ArachNoC

As funções foram escolhidas com objetivo de abranger as principais funcionalidades da especificação do *OpenCL*. A Figura 26 descreve, de forma simples, a especificação *OpenCL* como um diagrama de classes usando a notação Unified Modeling Language

(UML). Cada classe nesse diagrama é uma representação de um objeto OpenCL, ou seja, uma estrutura que guarda dados/informação. Por exemplo, *platform* é um objeto (estrutura de dado) que guarda informações sobre uma plataforma disponível no ambiente de execução.

São listadas as funções implementadas. Nesta lista, buscou-se agrupar os métodos que manipulam o mesmo tipo de objeto OpenCL. A intenção é destacar os principais **objetos** da especificação e demonstrar a importância das funções escolhidas para implementação.

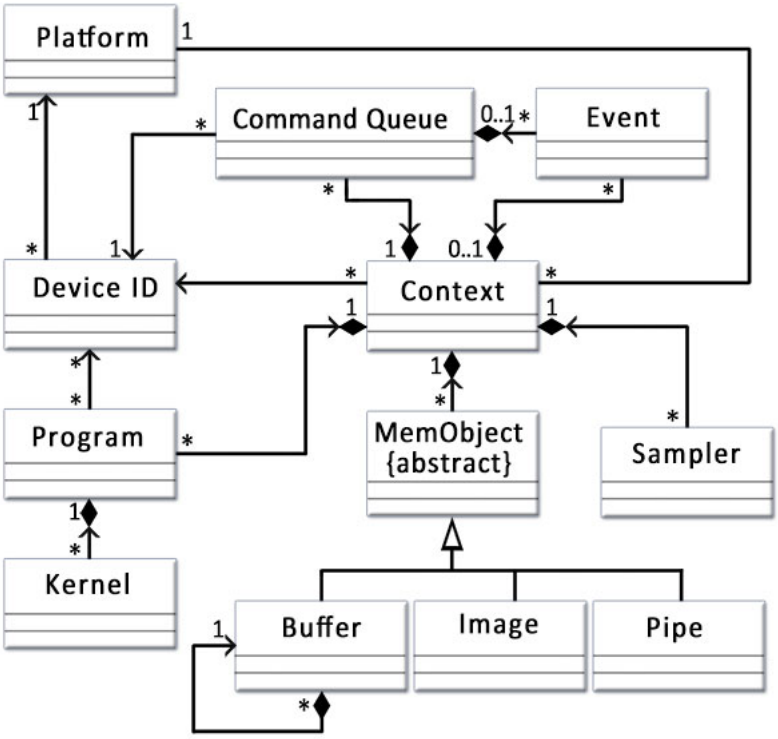


Figura 26 – Diagrama de classes da especificação *OpenCL*.

Fonte: Khronos (2015b)

- **Platform**: A implementação padrão da API OpenCL inclui um conjunto de funções que permite ao programador obter informações sobre as plataformas disponíveis no ambiente de execução. A API das arquiteturas METAL e ArachNoC implementam as funções descrita a seguir, diferindo apenas nas funções básicas referente a cada arquitetura.

  – *clGetPlatformIDs*: O desenvolvedor usa esta função para lista as plataformas disponíveis no ambiente de execução. Em METAL e ArachNoC o desenvolvedor vai receber de retorno um objeto que representa uma das plataformas. É uma estrutura de dados com informações da plataforma.

  - *clGetPlatformInfo*: Essa função retorna informações sobre uma determinada plataforma, por exemplo o nome, a versão, fornecedor, etc.

  Para identificar a informação requisitada pelo desenvolvedor, a função *clGetPlatformInfo* possui uma enumeração com seis itens possíveis. As funções desenvolvidas oferecem suporte a todas enumerações.

- **Device**: O desenvolvedor pode obter uma lista de dispositivos presentes em uma determinada plataforma e informações sobre um determinado dispositivo. Aplicações podem consultar particularidades sobre um dispositivo, a fim de determinar, entre os disponíveis, qual dispositivo utilizar e a forma desse uso.

  As duas arquiteturas, METAL e ArachNoC, possuem estas funções e suas implementações são semelhantes.

  - *clGetDeviceIDs*: Utilizado para retornar uma lista de dispositivos disponíveis em uma plataforma.
  - *clGetDeviceInfo*: É usada para obter informações sobre um determinado dispositivo, como por exemplo as informações:
    1. Quantidade de *compute units* disponíveis. *Compute units* é a abstração, em OpenCL, que fica responsável pelo processamento de um *work-group*. Tanto em METAL como em ArachNoC, um *work-group* é enviado para processamento em um nó da rede em chip;
    2. Dimensão máxima de espaço de execução: monodimensional, bidimensional ou tridimensional. METAL e ArachNoC suportam todas as definições de espaços;
    3. Limites nos números de *work-tems* que podem ser definidos em cada dimensão do espaço de execução;

  A função *clGetDeviceInfo* possui uma enumeração com noventa e dois itens possíveis para identificar a informação requisitada pelo desenvolvedor. Desses, tanto para METAL como ArachNoC, foram implementados nove itens. Os demais não foram implementados porque não se aplicam ao escopo deste trabalho de dissertação de mestrado. A Tabela 1 exibe os itens implementados.

- **Context**: Os contextos são utilizados, em tempo de execução, para gerenciamento de fila de comandos, memórias, programas e *kernels*. Também é utilizado no controle da execução dos *kernels* sobre um ou mais dispositivos. As implementações de METAL e ArachNoC são semelhantes para as seguintes funções.

  - *clCreateContext*: Cria um objeto contexto.

Tabela 1 – Lista de tipos enumerados implementados para *clGetDeviceInfo*.

Tipo enumerado	**Tipo de retorno**
CL_DEVICE_TYPE	cl_device_type
CL_DEVICE_MAX_COMPUTE_UNITS	cl_uint
CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS	cl_uint
CL_DEVICE_MAX_WORK_ITEM_SIZES	size_t [ ]
CL_DEVICE_MAX_WORK_GROUP_SIZE	size_t
CL_DEVICE_AVAILABLE	cl_bool
CL_DEVICE_PLATFORM	cl_platform_id
CL_DEVICE_NAME	char [ ]
CL_DEVICE_VENDOR	char [ ]

  - *clGetContextInfo*: Retorna informações sobre um determinado objeto contexto.

As implementações dessas funções, para METAL e ArachNoC, seguem todas as especificações OpenCL.

- **Command Queues**: Os *command_queues* são utilizados para criar filas de comandos para gerenciamento de objetos OpenCL como os *kernels* e, também, para objetos de leitura e escrita em memória. As funções que serão implementadas para esta funcionalidade e foram desenvolvidas para as duas arquiteturas METAL e ArachNoC.

  - *clCreateCommandQueueWithProperties*: Cria um objeto *command_queues*.
  - *clGetCommandQueueInfo*: Retorna informações sobre um *command_queues*.

  As implementações dessas funções seguem todas as especificações OpenCL.

  A especificação OpenCL descreve uma função que é responsável pelo envio dos comandos previamente associados com uma fila de comandos (*command_queue*). A função *Finish* foi desenvolvida para METAL e ArachNoC.

  - *clFinish*: Envia para execução todos os comandos associados em uma fila de comando para serem executados em um dispositivo definidos nessa fila. Só retorna quando todos os comandos finalizarem suas execuções, ou seja, bloqueia o *host* até o final da execução de todos os comandos.

- **Buffer Objects**:

  Um *buffer* é utilizado para armazenar uma coleção de elementos. Os elementos podem ser um tipo escalar (como int, float), vetor ou uma estrutura definida pelo usuário. Para manipulação e criação de um *buffer*, as seguintes funções foram desenvolvidas para as arquiteturas METAL e ArachNoC.

  - *clCreateBuffer*: Essa função cria um objeto *buffer*.

  - *clEnqueueWriteBuffer*: Coloca em uma fila de comandos (*command_queues*) um comando de transferência de dados tendo como destino um *buffer*.
  - *clEnqueueReadBuffer*: Coloca em uma fila de comandos (*command_queues*) um comando de cópia de um *buffer* para a memória do *host*.
  - *clEnqueueCopyBuffer*: Coloca em uma fila de comandos (*command_queues*) um comando para copiar um *buffer* para outro *buffer*.

  As implementações dessas funções, também, seguem todas as especificações OpenCL.

- **Program**: Um objeto *program* consiste em uma ou mais funções identificadas com o qualificador *__kernels* no código fonte. As seguintes funções foram implementadas para as duas arquiteturas e seguem todas as especificações OpenCL.

  - *clCreateProgramWithBinary*: Cria um objeto programa. Um de seus parâmetros é o binário da função *kernel* previamente compilada.
  - *clGetProgramInfo*: Retorna informações sobre um determinado programa.

- **Kernel**: Um objeto *kernel* encapsula uma função identificada por um qualificador *__kernel* e os valores dos argumentos dessa função. As seguintes funções foram implementadas para as duas arquiteturas e seguem todas as especificações OpenCL.

  - *clCreateKernel*: Cria um objeto *kernel*.
  - *clSetKernelArg*: Função usada para definir o valor de todos os argumentos de uma função *kernel*.
  - *clGetKernelInfo*: Retorna informações sobre um objeto *kernel*.

  O desenvolvedor define o envio do *kernel* para execução utilizando as funções *clEnqueueNDRangeKernel* ou *clEnqueueNativeKernel*. Duas implementações de *clEnqueueNDRangeKernel* foram desenvolvidas uma para METAL e outra para ArachNoC. A função *clEnqueueNativeKernel* configura o envio de uma função escrita em C/C++ e não compilada com um compilador OpenCL. Uma chamada de *clEnqueueNativeKernel* configura apenas uma execução da função C/C++.

  - *clEnqueueNDRangeKernel*: Coloca na fila de comandos (*command_queues*) um comando para execução múltipla de um *kernel* em um dispositivo.

3.9.2 Especificações OpenCL não implementadas

Nesta seção, serão descritas as especificações do padrão OpenCL que não foram desenvolvidas para nenhuma arquitetura alvo. Essa descrição não será exaustiva, apenas abrangerá as principais funcionalidades do padrão OpenCL. A seguinte lista é dividida

em funcionalidades do padrão OpenCL e, em cada item, são citadas as funções não desenvolvidas. O principal motivo do não desenvolvimento de toda especificação do padrão OpenCL é o tempo exigido para implementação e validação.

- ***Device***: São várias as funções que lidam com os dispositivos, dentre as funções que não foram implementadas são relacionadas três:
  - *clCreateSubDevices*: O desenvolvedor, utilizando a função *clCreateSubDevices*, pode criar sub-dispositivo para envio dos *kernels* a uma sub-divisão de um dispositivos.
  - *clRetainDevice*: Incrementa um contador que referencia dispositivos criados.
  - *clReleaseDevice*: Decrementa um contador que referencia dispositivos criados.
- ***Image Objects***: Não foram desenvolvidas nenhuma das funções que lidam com imagens. São listadas a seguir:
  - *clCreateImage*: Criar um objeto imagem.
  - *clGetSupportedImageFormats*: Retorna uma lista de formatos de imagens suportadas pela implementação OpenCL.
  - *clEnqueueWriteImage*: Coloca em uma fila de comandos (*command_queues*) um comando de transferência de uma imagem, a partir do *host*, tendo como destino um *buffer*.
  - *clEnqueueReadImage*: Coloca em uma fila de comandos (*command_queues*) um comando de cópia de uma *imagem* para a memória do *host*.
  - *clEnqueueCopyImage*: Coloca em uma fila de comandos (*command_queues*) um comando para copiar um objeto de imagem para outro.
- ***Pipes***: Um *pipes* é uma estrutura que guarda dados organizados como uma FIFO. Não foram desenvolvidas nenhuma das funções que lidam com *Pipes*. As funções relacionadas ao objeto *Pipes* são listadas abaixo.
  - *clCreatePipe*: Cria um objeto *Pipes*.
  - *clGetPipeInfo*: Retorna informações sobre um objeto *Pipes*.
- ***Sampler Objects***: Não é oferecido suporte a tipo de dado *Sampler*. Sampler é usado para controle de como elementos de um objeto de imagem são lidos. São listadas as funções que lidam com esse tipo de dado.
  - *clCreateSamplerWithProperties*: Cria um *Sampler*.
  - *clGetSamplerInfo*: Retorna informações sobre um determinado *Sampler*.

- ***Program Objects***: Uma função relacionada com programa não foi implementada.
  - *clCreateProgramWithSource*: Cria um objetos programa e carrega o código fonte (não compilado) do kernel.
- ***Building Program***: A especificação OpenCL define uma função para compilar o *kernel* em tempo de execução para todos os dispositivos relacionados em um objeto programa.
  - *clBuildProgram*: Realiza, em tempo de execução, a compilação e ligação de um código fonte kernel.
  - *clCompileProgram*: Realiza, em tempo de execução, a compilação de um código fonte kernel.
  - *clLinkProgram*: Realiza, em tempo de execução, a ligação de um código compilado e cria um executável.

3.10 Considerações Finais

O desenvolvimento de aplicações paralelas tendo como alvo um sistema multiprocessado é uma tarefa complexa por diversas razões. Um dos fatores é a evolução dos supercomputadores paralelos, com milhares de núcleos de processamento, cada um com suas particularidades. As ferramentas apresentadas oferecem suporte ao desenvolvimento de aplicações paralelas. Essas aplicações possuem códigos fontes portáveis para diversão arquiteturas. Quem garante isso é o padrão OpenCL adotado. O próximo capítulo apresenta as validações realizadas com os compiladores CLEM e OCEAN.

4 Validações e Resultados

Este capítulo avalia a implementação dos compiladores desenvolvidos para METAL e ArachNoC. A implementação dos compiladores é completa, pode-se compilar uma grande variedade de código fonte escrito em C/C++, pois todos os requisitos básicos estão presentes. É oferecido suporte às operações aritméticas e lógicas sobre os tipos de dados comuns da linguagem C.

4.1 Validações

Todo novo módulo de *software* desenvolvido pode ser analisado comparando-se com um produto equivalente e estável. Para um compilador, três relevantes critérios podem ser analisados: a velocidade de compilação, o tamanho do código de máquina emitido e o desempenho do programa compilado.

As aplicações Dijkstra e Susan, do *Benchmark* ParMiBench (IQBAL; LIANG; GRAHN, 2010), foram utilizadas para avaliar o tempo de compilação e a densidade de código dos dois compiladores, isto sem alterar o código original do *benchmark*. Para a avaliação do tempo de execução foi utilizada uma implementação do algoritmo Dijkstra. Não foi utilizada a implementação original da aplicação Dijkstra do ParMiBench, mas foi desenvolvida uma implementação OpenCL para essa avaliação.

O Dijkstra é um algoritmo utilizado para calcular o caminho de custo mínimo entre vértices de um grafo. O algoritmo inicia a partir de um vértice denominado raiz e este algoritmo calcula o custo mínimo do vértice raiz para todos os vértices do grafo. O Susan é um algoritmo para detecção de bordas em imagem. O algoritmo percorre a imagem com uma máscara circular. O brilho de cada *pixel*, dentro da área da máscara, é comparando com o brilho do *pixel* central. Uma borda é encontrada quando os brilhos não são semelhantes.

A versão do GCC utilizada é a 4.8.4, enquanto os compiladores CLEM e OCEAN foram construídos com a versão 3.7.1 do *framework* LLVM.

4.1.1 Comparação: LLVM vs GCC

- ***Tempo de Compilação***: Este teste mensura o tempo necessário para compilar o código fonte e gerar o código da máquina alvo. Para realização dos testes, utilizou-se um processador Intel core i3. Retirou-se o menor tempo entre 30 medições realizadas para cada compilador, dessa forma, buscou-se garantir o aquecimento da máquina

utilizada nos testes. Esse processo foi realizado com os quatro níveis de otimização: O0, O1, O2 e O3 (diretivas de otimização) (gcc.gnu.org, 2016). Esses módulos de otimização estão disponíveis tanto no LLVM quanto no GCC.

Os testes realizados nos dois compiladores, tanto para o Dijkstra quanto o Susan, apresentaram tempos semelhantes. Assim, será mostrado uma só figura para os dois compiladores. Isso era esperado, pois os mesmos foram desenvolvidos reutilizando a infra-estrutura LLVM e geram códigos para arquiteturas baseadas em MIPS.

As Figuras 27 e 28 exibem, respectivamente, os resultados para as aplicações Dijkstra e Susan.

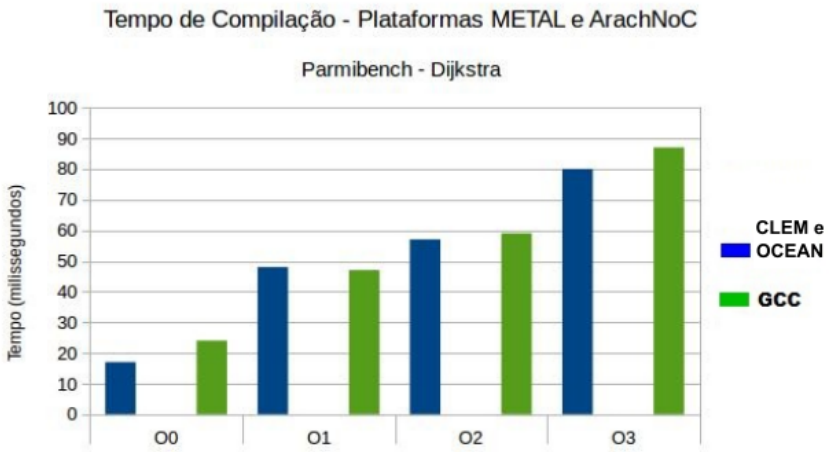


Figura 27 – Tempo de Compilação para METAL e ArachNoC com o algoritmo Dijkstra.

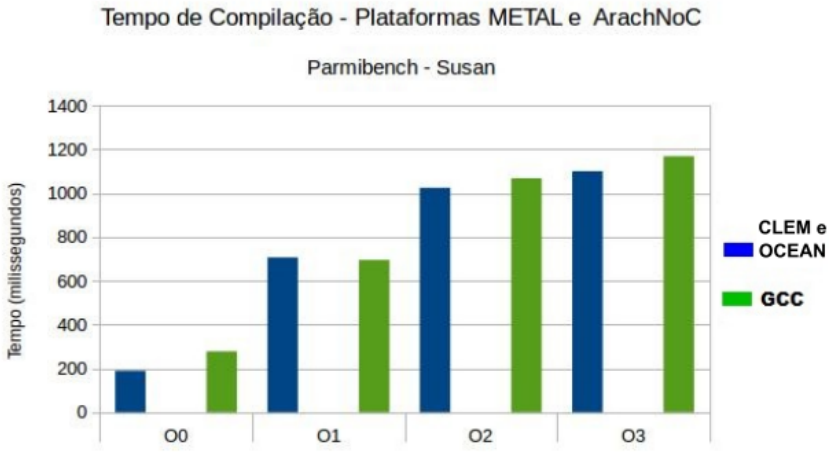


Figura 28 – Tempo de Compilação para METAL e ArachNoC com o algoritmo Susan.

Todos os compiladores apresentam uma pequena adição no tempo quando aumenta-se o nível de otimização utilizada. Isso se deve ao acréscimo de transformações e análises, o que reflete no tempo total de compilação.

CLEM e OCEAN levaram um tempo 30% menor com relação ao GCC. Eles demoraram um pouco mais compilando em O1 (de 2% a 3%), mas foram superiores em O2 e O3 (entre 5% a 6%).

- ***Densidade de Código***: Este teste mensura o número de instruções de máquina geradas. O arquivo de saída dos compiladores, para esta avaliação, é o código de máquina (*assembly*). Esse teste também analisa os quatro níveis de otimização O0, O1, O2 e O3. Os resultados também são similares e é mostrada apenas um gráfico para os dois compiladores CLEM e OCEAN.

  As Figuras 29 e 30 exibem, respectivamente, os resultados para as aplicações Dijkstra e Susan.

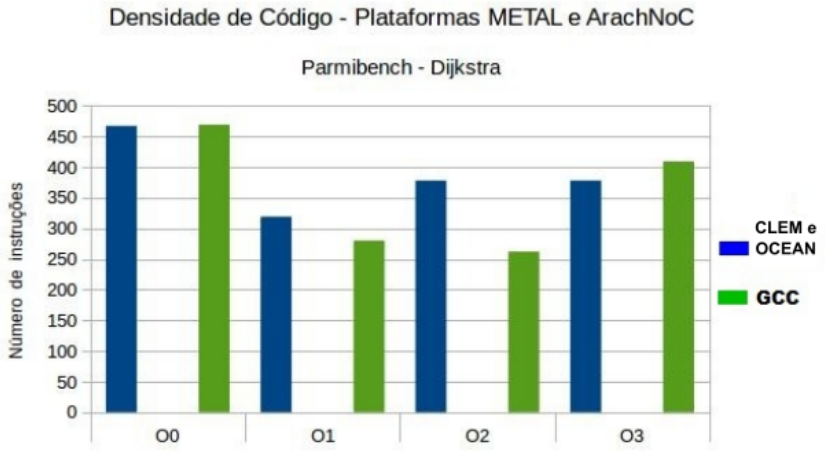


Figura 29 – Densidade de código para METAL e ArachNoC com o algoritmo Dijkstra.

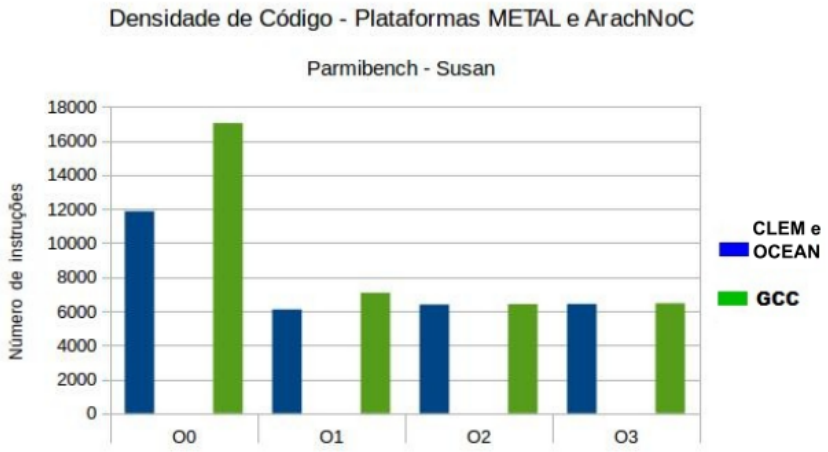


Figura 30 – Densidade de código para METAL e ArachNoC com o algoritmo Susan.

Os resultados do algoritmo Dijkstra, na Figura 29, mostram um empate no teste sem otimização (O0). Em O1 e O2, os compiladores CLEM e OCEAN geraram, respectivamente, um código 12% e 30% maior. Os compiladores desenvolvidos geram um código 7% menor em O3.

O algoritmo Susan, na Figura 30, apresenta um empate em O2 e O3. Nos testes com O1 e O2, os compiladores geraram, respectivamente, um código 30% e 14% menor.

- ***Tempo de Execução***: Esta avaliação compara o desempenho do código gerado. O código gerado pelos compiladores CLEM e OCEAN foram, respectivamente, executados em suas arquiteturas Metal e ArachNoC. Foi avaliado o desempenho do código compilado sem otimização (O0).

  Foram desenvolvidas duas aplicações Dijkstra, uma para CLEM e outra para OCEAN. Para a plataforma METAL, a aplicação Dijkstra desenvolvida possui as seguintes características: a entrada do algoritmo foi um grafo com 32 vértices; foram criadas 32 tarefas, uma para cada vértice do grafo; dividiu-se as 32 tarefas em 8 grupos e, finalmente, cada grupo foi enviado para um único nó escravo.

  Já a aplicação Dijkstra foi implementada da seguinte maneira: a entrada do algoritmo foi um grafo com 9 vértices; foram criadas 9 tarefas pertencentes a um grupo; esse grupo foi enviado para computar em um nó escravo.

  As Figuras 31 e 32 exibem, respectivamente, os tempos das execuções em ciclos de *clock* para METAL e ArachNoC.

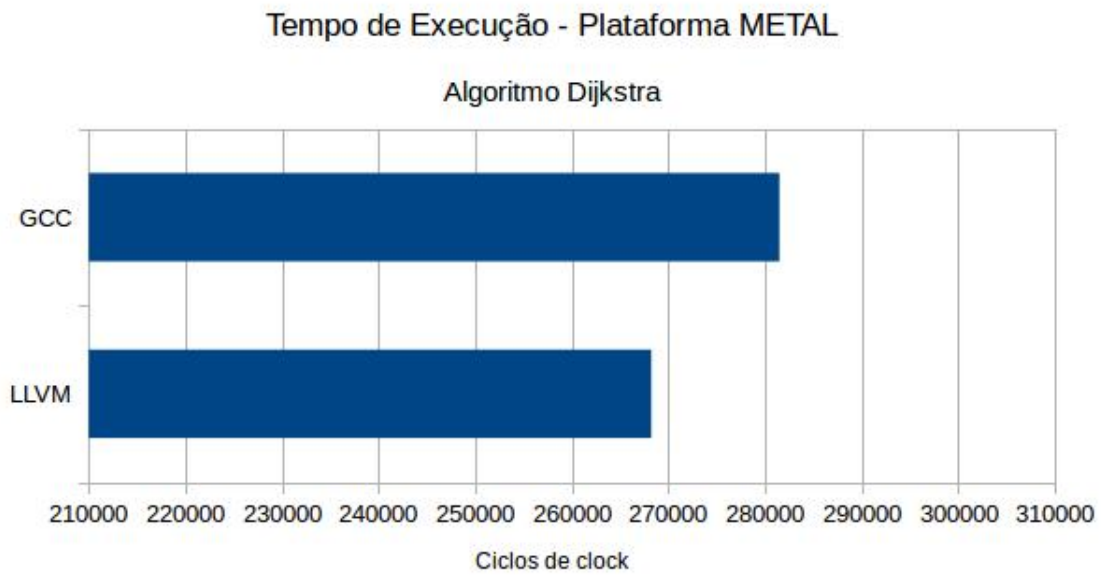


Figura 31 – Tempo de Execução para METAL.

Os resultados mostram que o tempo de execução dos códigos gerados é 4,5% menor que o gerado pelo GCC.

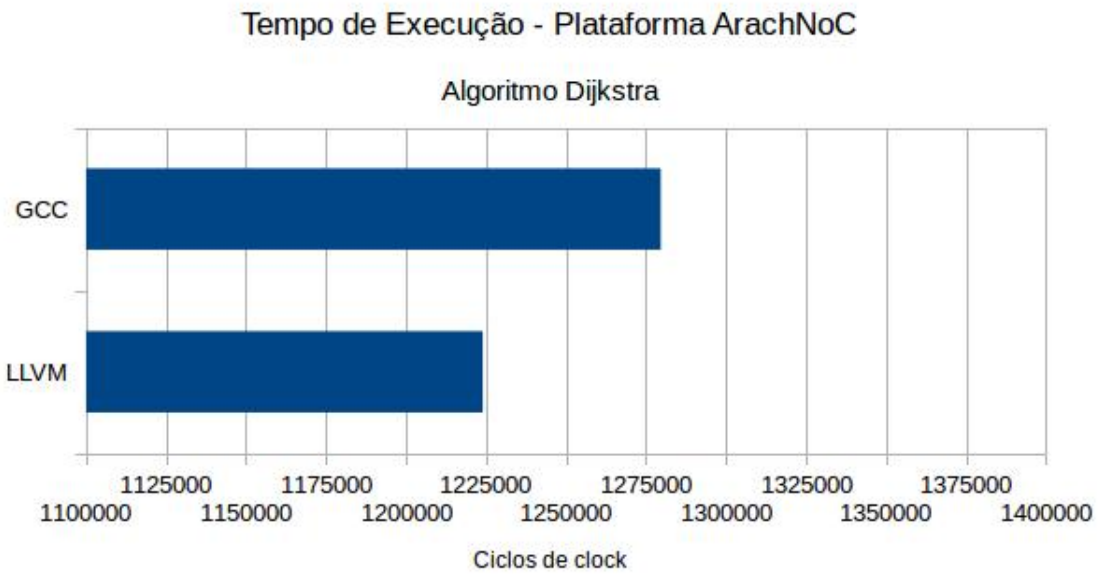


Figura 32 – Tempo de Execução para ArachNoC.

5 Conclusões

5.1 Conclusões

A utilização de CPUs *multicore* em conjunto com GPUs leva o desenvolvedor a possuir conhecimento de uma variedade maior de arquiteturas para produzir códigos *multi-threads* eficientes. Neste contexto, é fundamental a evolução das linguagens e ferramentas de desenvolvimento que apoiem os desenvolvedores de softwares, pois elas (linguagens e ferramentas) propiciam maior produtividade e qualidade.

Neste trabalho de dissertação, desenvolveu-se dois compiladores CLEM e OCEAN para compilação de códigos escritos em linguagem de alto nível (*OpenCL/C*), para as arquiteturas METAL e ArachNoC respectivamente. Tais arquiteturas foram, também, desenvolvidas no escopo de duas dissertações de mestrado, iniciadas concomitantemente com esta no grupo de pesquisa em circuitos e sistemas embarcados (do inglês, CESLa, *Circuits and Embedded Systems Laboratory*), da Universidade Federal do Piauí (UFPI). Como apresentado na Seção 1.2, a principal característica dessas arquiteturas é o paralelismo massivo, obtido da integração de múltiplos nós *multicores* de processamento.

A implementação dos compiladores é completa, pode-se compilar uma grande variedade de código fonte escrito em C/C++, pois todos os requisitos básicos estão presentes. É oferecido suporte às operações aritméticas e lógicas sobre os tipos de dados comuns da linguagem C.

O Capítulo 4 apresenta um comparativo entre os compiladores desenvolvidos em relação ao compilador GNU GCC. Comparou-se o tempo de compilação, a densidade de código gerado na compilação e o tempo de execução. As aplicações Dijkstra e Susan, do *Benchmark* ParMiBench, foram utilizadas para avaliar o tempo de compilação e a densidade de código nos dois compiladores. Para o tempo de execução, foi utilizada uma implementação própria do algoritmo Dijkstra. Os resultados apresentados mostram um melhor desempenho dos compiladores CLEM e OCEAN. Esses ganham no tempo de compilação e execução. Enquanto na densidade de código gerado, pose-se considerar um empate com o GCC.

5.2 Trabalhos futuros

Para trabalhos futuros, pretende-se realizar um comparativo entre as plataformas METAL e ArachNoC. Desenvolver um *runtime* para gerenciamento de nós escravos. Esse *runtime* poderá ligar ou desligar um nó dependendo das informações cedidas pelo *hardware*.

Por exemplo, desligar um nó caso esteja superaquecendo. Pretende-se criar um depurador de código. Essa ferramenta vai permitir encontrar erros nos códigos desenvolvidos para as plataformas.

Referências

AHO, A.; SETHI, R.; LAM, S. *Compiladores: princípios, técnicas e ferramentas*. [S.l.]: LONGMAN DO BRASIL, 2008. ISBN 9788588639249. Citado na página 1.

AMD. *Advanced Micro Devices Inc: Accelerated parallel processing (APP) software development kit (SDK)*. 2012. V2.8. Disponível em: <http://developer.amd.com/>. Citado na página 25.

BARR, M.; MASSA, A. *Programming Embedded Systems: With C and GNU Development Tools*. [S.l.]: O'Reilly Media, Inc., 2006. ISBN 0596009836. Citado na página 31.

Cobham. *Site Oficial Cobham*. 2016. Disponível em: <http://www.gaisler.com/>. Acesso em: 14 ago. 2016. Citado na página 23.

Cobham Gaisler. *LEON4 Processor*. 2016. Disponível em: <http://www.gaisler.com/index.php/products/processors/leon4>. Acesso em: 14 ago. 2016. Citado na página 23.

ERICSSON. *Erlang programming language*. 2016. Disponível em: <https://www.erlang.org/>. Acesso em: 14 ago. 2016. Citado na página 24.

FANG, J.; VARBANESCU, A. L.; SIPS, H. A comprehensive performance comparison of cuda and opencl. IEEE, p. 216 – 225, 2011. ISSN 1544-3566. Citado na página 2.

FARQUHAR, E.; BUNCE, P. *The Mips Programmer's Handbook*. 1st. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993. ISBN 1558602976. Citado na página 29.

gcc.gnu.org. *GNU GCC: Optimize-Options*. 2016. Disponível em: <https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html#Optimize-Options>. Acesso em: 14 ago. 2016. Citado na página 48.

Haskell. *Site Oficial Haskell*. 2016. Disponível em: <https://www.haskell.org/>. Acesso em: 14 ago. 2016. Citado na página 24.

haskell.org. *Site Oficial GHC*. 2016. Disponível em: <https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/>. Acesso em: 14 ago. 2016. Citado na página 24.

HENNESSY, J.; JOUPPI, N.; PRZYBYLSKI, S.; ROWEN, C.; GROSS, T.; BASKETT, F.; GILL, J. Mips: A microprocessor architecture. *SIGMICRO Newsl.*, ACM, New York, NY, USA, v. 13, n. 4, p. 17–22, out. 1982. ISSN 1050-916X. Disponível em: <http://dl.acm.org/citation.cfm?id=1014194.800930>. Citado na página 13.

HENNESSY, J. L.; PATTERSON, D. A. *Computer Architecture: A Quantitative Approach*. [S.l.: s.n.], 2011. Citado na página 15.

Imagination Technologies LTD. *MIPS® Architecture for Programmers Volume II-B: microMIPS32™ Instruction Set*. [S.l.], 2016. Citado na página 23.

Institute of Electrical and Electronics Engineers Inc. *1666-2005 - IEEE Standard System C Language Reference Manual*. New York, NY, USA, 2006. 0–423 p. Citado na página 28.

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 47.

JääSKELäINEN, P.; de La Lama, C. S.; SCHNETTER, E.; RAISKILA, K.; TAKALA, J.; BERG, H. pocl: A performance-portable opencl implementation. *International Journal of Parallel Programming*, Springer, New York, USA, v. 43, p. 752–785, 2015. ISSN 1573-7640. Citado 2 vezes nas páginas 25 e 39.

JONES, S. L. P.; SALKILD, J. The spineless tagless g-machine. In: *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*. New York, NY, USA: ACM, 1989. (FPCA '89), p. 184–201. ISBN 0-89791-328-0. Disponível em: <http://doi.acm.org/10.1145/99370.99385>. Citado na página 24.

KARIMI, K.; DICKSON, N. G.; HAMZE, F. A performance comparison of CUDA and opencl. *CoRR*, abs/1005.2581, 2010. Disponível em: <http://arxiv.org/abs/1005.2581>. Citado na página 1.

KHRONOS. *The open standard for parallel programming of heterogeneous systems*. 2015. Disponível em: <https://www.khronos.org/opencl/>. Acesso em: 6 nov. 2015. Citado na página 5.

KHRONOS. *The OpenCL Specification*. [S.l.], 2015. Disponível em: <https://www.khronos.org/registry/cl/specs/opencl-2.0.pdf>. Acesso em: 6 nov. 2015. Citado 5 vezes nas páginas 5, 6, 8, 9 e 40.

KIRK, D.; HWU, W. *Programming Massively Parallel Processors: A Hands-on Approach*. [S.l.]: Morgan Kaufmann, 2010. ISBN 978-0-12-415992-1. Citado na página 1.

KOENIG, R.; BAUER, L.; STRIPF, T.; SHAFIQUE, M.; AHMED, W.; BECKER, J.; HENKEL, J. Kahrisma: A novel hypermorphic reconfigurable-instruction-set multi-grained-array architecture. In: *Proceedings of the Conference on Design, Automation and Test in Europe*. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2010. (DATE '10), p. 819–824. ISBN 978-3-9810801-6-2. Disponível em: <http://dl.acm.org/citation.cfm?id=1870926.1871127>. Citado na página 24.

KOLEK, J.; JOVANOVIć, Z.; ŠLJIVIć, N.; NARANčIć, D. Adding micromips backend to the llvm compiler infrastructure. In: *Telecommunications Forum (TELFOR), 2013 21st*. [S.l.: s.n.], 2013. p. 1015–1018. Citado 3 vezes nas páginas 19, 20 e 23.

LATTNER, C. Introduction to the llvm compiler system. In: *Proceedings of International Workshop on Advanced Computing and Analysis Techniques in Physics Research, Erice, Sicily, Italy*. [S.l.: s.n.], 2008. Citado na página 19.

LATTNER, C. Llvm and clang: Next generation compiler technology. In: *The BSD Conference*. [S.l.: s.n.], 2008. Citado na página 19.

LEE, J.; KIM, J.; KIM, J.; SEO, S.; LEE, J. An opencl framework for homogeneous manycores with no hardware cache coherence. In: *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*. [S.l.: s.n.], 2011. p. 56–67. ISSN 1089-795X. Citado na página 26.

LI, J. J.; KUAN, C. B.; WU, T. Y.; LEE, J. K. Enabling an opencl compiler for embedded multicore dsp systems. In: *2012 41st International Conference on Parallel Processing Workshops*. [S.l.: s.n.], 2012. p. 545–552. ISSN 0190-3918. Citado na página 25.

LLVM. *The LLVM Compiler Infrastructure*. 2015. Disponível em: <http://llvm.org/>. Acesso em: 6 nov. 2015. Citado na página 19.

LLVM. *TableGen*. 2015. Disponível em: <http://llvm.org/docs/TableGen/index.html>. Acesso em: 6 nov. 2015. Citado na página 20.

Lopez Trescastro, J.; Vassev, E.; Hellstrom, D.; Cederman, D. An LLVM Backend for LEON Processors. In: *DASIA 2015 - DAta Systems in Aerospace*. [S.l.: s.n.], 2015. (ESA Special Publication, v. 732), p. 42. Citado 2 vezes nas páginas 23 e 24.

MAJETI, D.; SARKAR, V. Heterogeneous habanero-c (h2c): A portable programming model for heterogeneous processors. In: . Hyderabad, Telangana, Índia: IEEE, 2015. p. 708 – 717. ISBN 978-1-4673-7684-6/15. Citado na página 1.

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 5.

NAH, J.; LEE, J.; KIM, H.; LEE, J.; HWANG, S. J.; YOO, D.; LEE, J. An opencl optimizing compiler for reconfigurable processors. In: *Field-Programmable Technology (FPT), 2013 International Conference on*. [S.l.: s.n.], 2013. p. 184–191. Citado na página 25.

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 na página 27.

NUGTEREN, C.; CORPORAAL, H. Bones: An automatic skeleton-based c-to-cuda compiler for gpus. *ACM Trans. Archit. Code Optim.*, ACM, New York, NY, USA, v. 11, n. 4, p. 35:1–35:25, dez. 2014. ISSN 1544-3566. Disponível em: <http://doi.acm.org/10.1145/2665079>. Citado na página 27.

open-std.org. *Rationale for International Standard—Programming Languages—C*. [S.l.], 2003. Disponível em: <http://www.open-std.org/jtc1/sc22/wg14/www/C99RationaleV5.10.pdf>. Citado na página 39.

OPENMP. 2016. Disponível em: <http://www.openmp.org>. Acesso em: 14 ago. 2016. Citado na página 2.

PAONE, E. *Design space exploration of openCL applications on heterogeneous parallel platforms*. Tese (Doutorado) — Politecnico Di Milano, 2014. Citado 2 vezes nas páginas 2 e 27.

PIPS project. 2016. Disponível em: <http://pips4u.org/>. Acesso em: 14 ago. 2016. Citado na página 1.

SAGONAS, K.; STAVRAKAKIS, C.; TSIOURIS, Y. Erllvm: An llvm backend for erlang. In: . Copenhagen, Denmark: ACM, 2012. ISBN 978-1-4503-1575-3. Citado na página 24.

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 2.

SPEAR-DE interface. 2016. Disponível em: <http://pips4u.org/projects-using-pips/pips-spear-interface>. Acesso em: 14 ago. 2016. Citado na página 1.

STONE, J. E.; GOHARA, D.; SHI, G. Opencl: A parallel programming standard for heterogeneous computing systems. IEEE, p. 66 – 73, 2010. ISSN 1521-9615. Citado na página 2.

STRIPF, T.; KöNIG, R.; RIEDER, P.; BECKER, J. A compiler back-end for reconfigurable, mixed-isa processors with clustered register files. In: *IPDPS Workshops*. IEEE Computer Society, 2012. p. 462–469. ISBN 978-1-4673-0974-5. Disponível em: <http://dblp.uni-trier.de/db/conf/ipps/ipdps2012w.html#StripfKRB12>. Citado na página 24.

TANENBAUM, A. *Sistemas operacionais modernos*. 2st. ed. [S.l.]: Prentice-Hall do Brasil, 2003. ISBN 9788587918574. Citado na página 30.

TEREI, D. A.; CHAKRAVARTY, M. M. An llvm backend for ghc. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 45, n. 11, p. 109–120, set. 2010. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/2088456.1863538>. Citado na página 24.

The Potland Group. 2016. Disponível em: <http://www.pgroup.com/index.htmf>. Acesso em: 14 ago. 2016. Citado na página 1.

TSUCHIYAMA, R.; NAKAMURA, T.; IIZUKA, T.; ASAHARA, A.; MIKI, S. *The OpenCL Programming Book*. [S.l.]: Fixstars Corporation, 2010. Citado 4 vezes nas páginas 1, 5, 10 e 28.

YANG, C.-T.; CHANG, T.-C.; WANG, H.-Y.; CHU, W. C. C.; CHANG, C.-H. Performance comparison with openmp parallelization for multi-core systems. In: . Busan, Coréia do Sul: IEEE, 2011. p. 232 – 237. ISBN (978-0-7695-4428-1),(978-1-4577-0391-1). ISSN 2158-9178. Citado na página 1.

ZEFERINO, C.; SUSIN, A. Socin: a parametric and scalable network-on-chip. In: *Integrated Circuits and Systems Design, 2003. SBCCI 2003. Proceedings. 16th Symposium on*. [S.l.: s.n.], 2003. p. 169–174. Citado na página 11.