Universidade Federal de Pernambuco

Centro de Informática

Aplicando *Model-Driven Development* à Plataforma GPGPU

Trabalho de Graduação

Aluno: Ademir José de Carvalho Junior (ajcj@cin.ufpe.br)

Orientadora: Judith Kelner (jk@cin.ufpe.br)

Co-Orientador: Thiago Farias (tsmcf@cin.ufpe.br)

Recife, Dezembro de 2008

Universidade Federal de Pernambuco

Centro de Informática

Aplicando *Model-Driven Development* à Plataforma GPGPU

Trabalho de Graduação

Monografia apresentada ao Centro de Informática da Universidade Federal de Pernambuco, como requisito para obtenção do Grau de Bacharel em Ciência da Computação.

Orientadora: *Prof.ª* Judith Kelner
Co-orientador: Thiago Farias

Recife, Dezembro de 2008

A minha família.

Agradecimentos

Agradeço primeiramente à Deus, por ter me dado força e coragem de lutar pelos meus objetivos, e por ter guiado meus caminhos para que eu pudesse viver este momento.

Aos meus pais Sr. Ademir e D. Nilza, por todo amor, paciência e por nunca permitirem que eu desistisse do meu sonho, sendo exemplos de esforço e dedicação ao trabalho. Pela ótima base educacional que me proporcionaram, e pelos valores e princípios ensinados que me acompanharão pelo resto de minha vida. Não menos importante aos meus irmãos, Denílson e Elma, que sempre me apoiaram em momentos difíceis.

A todos os amigos da turma de Ciência da Computação 2004.2, em especial a turma do Projetão: André (Debão), Bruno (Pigmeu), Chico (Priquito), David (Buchecha), Felipe (Moxinho), Fernando (Japa), Filipe (Gravatá), Henrique (Rick Seabra), Rilter (Rilt Seabra) e Thiago (Xeroso), e também à amigos próximos da turma como Rebeka (Galega), Jesus (Aranha), Rodrigo (Digão), que sempre estiveram comigo durante todos esses anos sem nunca perder o bom humor e pelos bons momentos vividos. Aos amigos em geral, que souberam entender a minha ausência tantas vezes e que também me proporcionaram muitos momentos de alegria.

À minha orientadora, Judith Kelner, por todos os ensinamentos e apoio, e pela oportunidade da realização deste trabalho de graduação. E ao pessoal da GPRT, o grupo de pesquisa que me proporcionou a oportunidade de iniciar minha carreira profissional.

Não menos importante, a aqueles não citados, mas que contribuíram de alguma forma para a realização deste trabalho, ou que contribuíram para que eu chegasse até este momento.

Meus sinceros agradecimentos.

Resumo

A GPU (*Graphics Processing Unit*) é um dispositivo gráfico que vem ganhando destaque nos últimos anos pela sua eficiência em processamento paralelo. Principalmente, após o surgimento da plataforma de programação denominada CUDA (*Compute Unified Device Architecture*). A GPU vem sendo utilizada por programadores que desejam usufruir dos benefícios das GPUs em suas aplicações gráficas.

Neste contexto, o termo GPGPU (*General-Purpose computation on GPU*) é um novo conceito que visa explorar as vantagens das GPUs em áreas não necessariamente relacionadas a processamento gráfico. O desafio neste escopo é identificar os problemas que podem ser solucionados, e como mapear estes problemas para a plataforma, já que ela foi projetada originalmente para processamento gráfico.

A abordagem utilizada neste trabalho de graduação para lidar com esse desafio consiste no desenvolvimento baseado em modelos (ou *Model-Driven Development*). Esta é uma filosofia de desenvolvimento de software que transfere o foco de desenvolvimento do uso das linguagens de programação para a elaboração de modelos, expressos em alguma linguagem como UML (*Unified Modeling Language*), por exemplo.

O objetivo principal deste trabalho de graduação é o desenvolvimento de uma aplicação, o CudaMDA, que a partir de modelos é capaz de gerar código para a plataforma CUDA. Os modelos elaborados por esta aplicação refletem características do contexto de GPUs, mas que podem ser consideradas flexíveis ao ponto de serem aplicadas em diversos domínios.

Palavras-Chave: GPGPU, MDD, Engenharia de Software.

Sumário

1. INTRODUÇÃO 9
2. BACKGROUND 11
   - 2.1 GPU 11
     - 2.1.1 Arquitetura da GPU 12
   - 2.2 GPGPU 14
   - 2.3 MDD 15
   - 2.4 MDA 16
     - 2.4.1 Modelos 16
     - 2.4.2 Tipos de Modelos 17
3. ESTADO DA ARTE 19
   - 3.1 COSA 19
   - 3.2 Array-OL 20
   - 3.3 Gaspard2 21
   - 3.4 Linguagens para GPGPU 22
   - 3.5 CUDA 23
   - 3.6 Ferramentas para MDD 26
4. DESENVOLVIMENTO 27
   - 4.1 Requisitos Especificados 27
   - 4.2 Design do Software 29
   - 4.3 Implementação 31
     - 4.3.1 Perfil UML 31
     - 4.3.2 Componente de Manipulação de Modelos 32
     - 4.3.3 Componente de Geração de Código 34
     - 4.3.4 Componente de Interface Gráfica 35
5. APLICAÇÃO 36
6. RESULTADOS 39

6.1 Descrição do Cenário 39
6.2 Versão desenvolvida através da Ferramenta 40
6.3 Discussão 42
7. CONCLUSÃO E TRABALHOS FUTUROS 43
8. REFERÊNCIAS 44
APÊNDICE A – MODELOS GMF 47
APÊNDICE B – TEMPLATES JET 49

Índice de Ilustrações

Figura 2.1 - Comparação GPU/CPU em termos de GFLOPS.....12
Figura 2.2 - Disposição dos Transistores na CPU e na GPU. 13
Figura 2.3 - Arquitetura projetada para o *pipeline* gráfico.....13
Figura 3.1 - Exemplo de *looping* em COSA. 20
Figura 3.2 - Modelo global do Array-OL. 21
Figura 3.3 - Modelo local do Array-OL. 21
Figura 3.4 - Modelo de execução de CUDA. 24
Figura 3.5 - Modelo de Aplicações CUDA. 25
Figura 4.1 - Fluxo do CudaMDA. 29
Figura 4.2 - Arquitetura do CudaMDA. 30
Figura 5.1 - Interface gráfica do CudaMDA. 37
Figura 5.2 – Visualização de um Arquivo CUDA produzido pela ferramenta. 38
Figura 6.1 - Modelo do Algoritmo elaborado na Ferramenta. 40

1. INTRODUÇÃO

Engenharia de Software é uma área da computação que possui técnicas e ferramentas que podem ser aplicadas a diversos domínios relacionados ao processo de desenvolvimento de software. A principal motivação desta área é o aumento de produtividade, que os seus recursos são capazes de fornecer, neste processo.

Neste trabalho de graduação, o MDD (*Model-Driven Development*) [3], uma abordagem de Engenharia de Software, é aplicado ao desenvolvimento de softwares na área de Processamento Gráfico, mais especificamente no contexto de GPGPU (*General Purpose computation on GPU*) [1] [2], com a finalidade de aumentar a produtividade. A abordagem MDD visa produzir um ambiente mais adequado para a construção de aplicações na plataforma GPGPU.

A GPU (*Graphics Processing Unit*) [4] [5], é um dispositivo que vem ganhando destaque nos últimos anos, principalmente pela sua eficiência em processamento. Esta eficiência é obtida através de sua arquitetura, que contém um alto grau de paralelismo, característica necessária para a execução de sistemas gráficos. Para este dispositivo, existe uma plataforma de programação desenvolvida pela NVIDIA, denominada CUDA (*Compute Unified Device Architecture*) [6], utilizada por programadores (que desejam usufruir dos benefícios das GPUs existentes em placas gráficas da própria NVIDIA) em suas aplicações. CUDA utiliza uma extensão da linguagem C para desenvolver suas aplicações. Por ser uma plataforma recente, ainda não existem muitas opções de ferramentas CASE (*Computer Aided Software Engineering*) que auxiliem na produção de aplicações. Esta plataforma se destaca das demais pelo fato de possuir mecanismos que abstraem os detalhes de uma GPU para o programador, tornando mais fácil o seu uso para GPGPU.

GPGPU é um conceito que visa explorar as vantagens das GPUs em áreas não necessariamente relacionadas a processamento gráfico. O desafio é identificar os problemas que podem ser solucionados, e como mapear estes problemas para a plataforma, já que ela foi especificada para aplicações gráficas.

A abordagem utilizada neste trabalho consiste no desenvolvimento de

aplicações baseado em modelos, o MDD. Esta é uma filosofia de desenvolvimento de software que transfere o foco de desenvolvimento do uso das linguagens de programação para a elaboração de modelos, expressos em alguma linguagem como UML (*Unified Modeling Language*) [7], por exemplo.

O MDD é uma abordagem promissora, pois permite ao desenvolvedor, lidar com aspectos relacionados ao domínio em questão, ao invés de tratar aspectos ligados às linguagens de programação [8]. Porém, esta ainda não é uma abordagem bastante difundida, existem poucas iniciativas que a usam. Uma destas iniciativas é o MDA (*Model-Driven Architecture*) [9] que provê um conjunto de padrões para o desenvolvimento baseado em modelos.

Este trabalho de graduação objetiva principalmente o desenvolvimento de uma ferramenta, o CudaMDA, que a partir de modelos, é capaz de gerar código na plataforma CUDA. Os modelos elaborados por essa ferramenta refletem características do contexto de GPUs, mas que são flexíveis ao ponto de serem aplicadas em diversos domínios.

Este trabalho está estruturado da seguinte forma: o Capítulo 2 introduz os conceitos relacionados aos temas abordados, o Capítulo 3 aborda o estado da arte, citando alguns trabalhos relacionados e ferramentas existentes na área, o Capítulo 4 descreve o processo de desenvolvimento da aplicação, expondo os requisitos, arquitetura e implementação do software, o Capítulo 5 apresenta o resultado final do desenvolvimento, ou seja, o CudaMDA em funcionamento. O Capítulo 6 demonstra o uso da aplicação em um cenário específico. E os Capítulos 7 e 8 são respectivamente, a conclusão e as referências do trabalho.

2. BACKGROUND

Este capítulo visa introduzir alguns conceitos para fundamentar o restante do trabalho.

2.1 GPU

GPU [4] [5] é um tipo de microprocessador especializado no processamento de gráficos. Este dispositivo é encontrado em *videogames*, computadores pessoais e estações de trabalho, e pode estar situado na placa de vídeo ou integrado diretamente à placa-mãe.

De acordo com [6], as recentes arquiteturas das GPUs proporcionam, além de um vasto poder computacional como ilustra a Figura 2.1, uma grande velocidade no barramento de dados, sendo superiores as CPUs comuns nestes aspectos.

Apenas para exemplificar, podemos comparar o Pentium IV 3GHz, que possui picos de 6 GFLOPS e 6GB/s em seu desempenho, enquanto o GeForceFX 6800 possui picos de 53GFLOPS e 34GB/s. É importante lembrar que estes desempenhos foram obtidos em situações ideais. Existem casos em que o desempenho de uma CPU pode ser superior ao desempenho de uma GPU. O desempenho da GPU está relacionado à existência de paralelismo em uma aplicação.

Originalmente, GPUs foram concebidos para realizar *processamento gráfico*, incluindo iluminação, rasterização, e outras tarefas de sintetização de imagens. Este processo de sintetização é possível através de uma seqüência de estágios bem definidos chamado *pipeline* gráfico. Cada estágio é responsável por uma tarefa como: transformação de coordenadas; cálculos de câmera e iluminação; entre outras tarefas, maiores detalhes podem ser encontrados em [10].

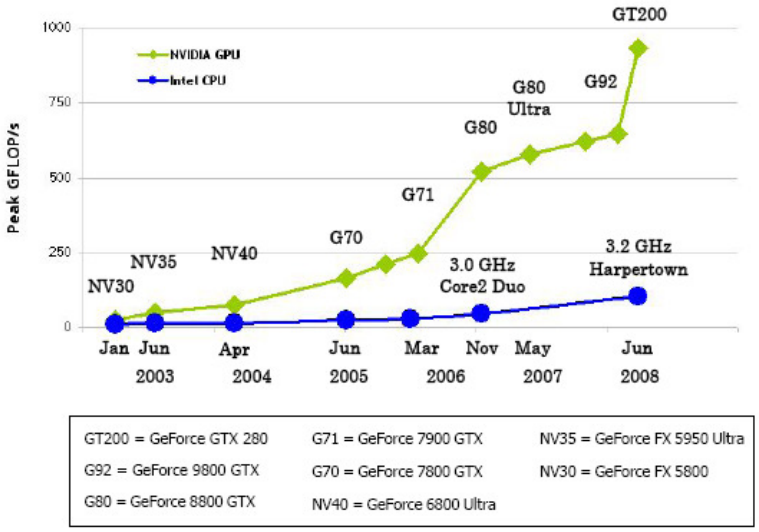


Figura 2.1 - Comparação GPU/CPU em termos de GFLOPS[1].

2.1.1 Arquitetura da GPU

GPUs são processadores orientados para a execução paralela de instruções. Eles são otimizados para processar operações sobre vetores, executando no modo SIMD (*Simple Instruction, Multiple Data*)[2].

A arquitetura das GPUs foi projetada de modo a incluir mais transistores dedicados ao processamento de dados, em detrimento da realização do *caching* de dados e do fluxo de controle, quando comparado a uma CPU comum. Isto é ilustrado na Figura 2.2.

[1]FLOPS (***fl****oating point* ***op****erations per* ***s****econd*) consiste em uma métrica para a velocidade de processamento de microprocessadores.

[2] SIMD é uma técnica aplicada para obtenção de paralelismo em processadores. No SIMD, uma mesma instrução é executada em paralelo em diferentes partes do hardware e sobre diferentes dados.

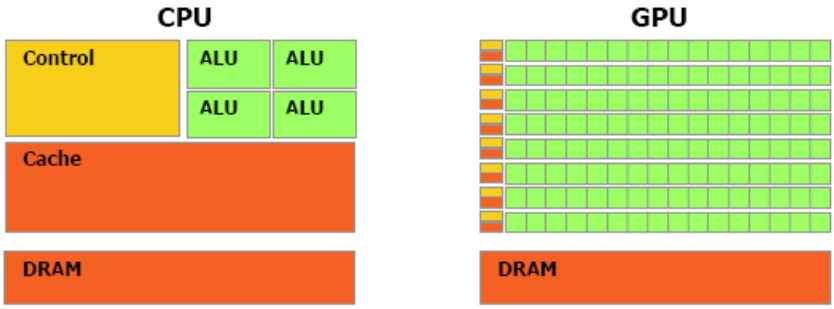


Figura 2.2 - Disposição dos Transistores na CPU e na GPU.

Desta forma, uma GPU consegue lidar melhor com problemas que possuem uma grande quantidade de operações sobre dados. Já que uma mesma operação é realizada sobre cada elemento dos dados, não existe a necessidade de um fluxo de controle sofisticado. Estas características permitem que latências no acesso a memória sejam disfarçadas através da grande vazão de cálculos.

Vale ressaltar, que a arquitetura de uma GPU, é bastante relacionada ao *pipeline* gráfico. Sua arquitetura foi projetada de modo a realizar as operações contidas no *pipeline* de forma simultânea, como ilustra a Figura 2.3.

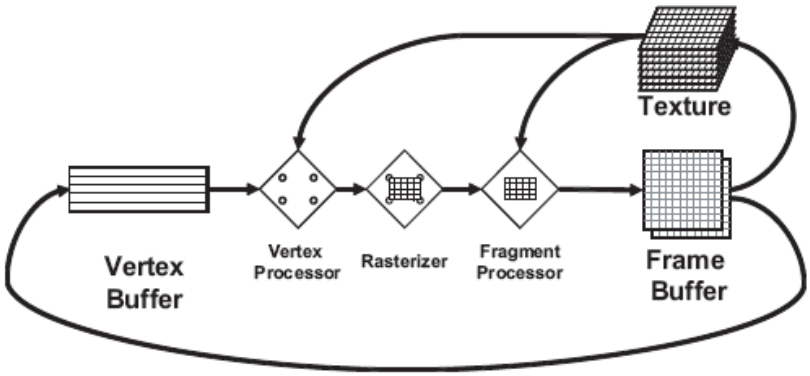


Figura 2.3 - Arquitetura projetada para o *pipeline* gráfico.

Novas funcionalidades foram adicionadas às GPUs recentemente, incluindo a habilidade de modificar o processo de renderização do *pipeline*, através dos chamados *shaders* [10]. Esta programabilidade é considerada uma característica chave da arquitetura.

Primeiramente, os *shaders* eram programados através de linguagens de baixo nível existentes nas GPUs. Apesar de estas linguagens proverem controle e flexibilidade, elas são de difícil uso. Atualmente, existem linguagens de alto nível que facilitam esta tarefa. Algumas destas linguagens são descritas no próximo capítulo.

2.2 GPGPU

Como foi mencionado, as GPUs apresentam um enorme poder computacional. Existe um conceito denominado GPGPU [1] [2], que aproveita a programabilidade destes dispositivos, para tentar estender suas vantagens para outras áreas de computação de propósito geral.

Uma das motivações desta área está relacionada ao binômio custo-desempenho. A demanda por jogos e aplicações gráficas levou a um aumento da necessidade de máquinas que atendessem aos requisitos de desempenho impostos por essas aplicações. O aumento desta demanda provocou uma redução nos custos deste tipo de equipamento, como as máquinas equipadas com GPUs (capazes de realizar milhares de operações por segundo), tornando viável sua utilização em outros contextos além de jogos e aplicações gráficas.

Nem todos os tipos de problemas se encaixam no contexto das GPUs, mas existem vários casos de aplicações que conseguiram um aumento significativo de performance através do uso de hardware gráfico. O conceito *intensidade aritmética*, mede a razão entre a quantidade de operações matemáticas e a quantidade de acessos a memória, existentes em uma aplicação. Quando mapeadas para as GPUs, aplicações que possuem uma alta intensidade aritmética são bem sucedidas.

Porém, a natureza gráfica das GPUs torna árdua a programação para outros fins, exigindo do programador conhecimentos extras, como a adaptação da aplicação ao *pipeline* gráfico. Para exemplificar, no caso de uma aplicação em GPGPU que utiliza *shaders* é necessária a renderização de alguma primitiva gráfica (por exemplo, um quadrilátero texturizado) para que o aplicativo seja inicializado, assim como a obtenção do *feedback* da aplicação, que é realizada através da leitura de texturas.

Conceitualmente, uma aplicação comum, não possui relação com o desenho de gráficos e manipulação de texturas, mas as linguagens existentes forçam os programadores a pensarem desta forma. As plataformas mais recentes tentam lidar com este problema abstraindo alguns detalhes da GPU para o programador. Algumas destas plataformas serão descritas no próximo capítulo.

2.3 MDD

MDD [3] é uma filosofia que possibilita a construção do modelo de um sistema e, a partir deste, a geração do sistema real.

Uma abordagem baseada em modelos transfere o foco de desenvolvimento das linguagens de programação de terceira geração para modelos, mais especificamente modelos descritos em UML [7] e seus *profiles*. O objetivo dessas abordagens é facilitar o desenvolvimento de aplicações, permitindo o uso de conceitos ligados ao domínio em questão ao invés dos conceitos oferecidos pelas linguagens de programação.

Porém essa não é a visão que a maioria dos desenvolvedores possui sobre modelagem, atualmente. A visão comum é de que os modelos são simples desenhos extraídos dos sistemas reais e que eles são desnecessários no processo de desenvolvimento de sistemas.

Cada modelo pode contemplar um ou mais aspectos de um sistema. Por exemplo, um sistema bancário pode ser modelado, ignorando aspectos de segurança e interface com o usuário, ou pode ser modelado realizando uma combinação entre estes domínios. Esta decisão está relacionada ao tipo de sistema que está sendo desenvolvido e as prioridades adotadas no seu desenvolvimento. Um sistema pode priorizar sua segurança, ou sua usabilidade, por exemplo.

Quanto maior o nível de abstração de um modelo, mais ele se relaciona ao vocabulário do domínio, reduzindo assim, a complexidade no manuseio do modelo por parte de um especialista no domínio.

Um dos principais desafios do MDD é a dificuldade técnica envolvida na transformação de modelos em código. Existe ainda uma descrença por parte dos

desenvolvedores sobre a geração de código eficiente, compacto, e que atenda ao propósito contido em seu modelo. Mas, podemos pensar nesta situação, como a mesma enfrentada há alguns anos atrás, com a introdução dos compiladores. Estes têm a função de transformar objetos descritos em uma linguagem de alto nível para objetos em uma linguagem de mais baixo nível (linguagem de máquina) e aspectos de eficiência e performance são considerados nesta transformação [3].

2.4 MDA

MDA [9] é um padrão definido pela OMG (*Object Management Group*)[1], um consórcio entre empresas de software, indústria, governo, e instituições acadêmicas. O MDA pode ser visto como um *framework* conceitual para a definição de um conjunto de padrões que suportam o MDD.

O MDA define uma abordagem para sistemas de TI (Tecnologia da Informação) que separa a especificação da funcionalidade de um sistema de sua implementação. Para isto ser possível, o MDA define uma arquitetura que provê um conjunto de boas práticas para estruturação destas especificações através de modelos.

2.4.1 Modelos

Em MDA, um modelo significa uma representação de parte de uma funcionalidade, estrutura ou comportamento de um sistema.

Uma especificação é considerada formal quando ela é baseada em uma linguagem que possui uma sintaxe, semântica, e possivelmente regras de análise, inferência ou prova de suas construções. A sintaxe pode ser textual ou gráfica. A semântica deve ser definida em termos do domínio em questão. As regras opcionais definem quais propriedades podem ser deduzidas a partir das declarações explicitadas em um modelo.

Em MDA, uma especificação que não segue este padrão não pode ser considerada como modelo. Um modelo MDA precisa estar relacionado, sem ambigüidades, a uma definição de sintaxe e semântica, como por exemplo, definições

[1] Para mais informações sobre o OMG visite http://www.omg.org.

elaboradas através do MOF (*Meta-Object Facility*) [12].

Note que através dessa definição, um código-fonte pode ser considerado um modelo, possuindo a característica adicional de ser executável em alguma plataforma. De forma similar, uma especificação baseada em UML é um modelo cujas propriedades podem ser expressas graficamente via diagramas.

2.4.2 Tipos de Modelos

MDA possui alguns tipos de modelos para descrição de um sistema. Estes modelos são baseados em UML. Os principais são o PIM (*Modelo Independente de Plataforma*), e o PSM (*Modelo Específico de uma Plataforma*). Existe também o CIM (*Modelo Independente de Computação*), que representa a visão de um sistema, sem levar em consideração detalhes relativos à sua computação. O CIM não revela detalhes da estrutura de um sistema, mas sim conceitos ligados ao domínio em questão, por isso, este modelo também é conhecido como modelo do domínio.

O modelo PIM fornece informações sobre a estrutura e a funcionalidade de um sistema, porém, abstraindo os detalhes técnicos. Ele descreve componentes existentes e suas interações sem relacioná-los a uma tecnologia ou plataforma específica.

O PSM funciona como uma extensão ao PIM, incluindo detalhes específicos de uma tecnologia ou plataforma. O PSM pode ser obtido manualmente, ou através de uma operação denominada transformação de modelos. Esta é uma operação que conta com o auxílio de algumas tecnologias como o XMI (*XML Metadata Interchange)*[1] [13] e o MOF [12]. Como este modelo é descrito em UML, existe uma incompatibilidade na representação dos elementos da plataforma, através dos elementos existentes em UML. Por exemplo, como representar uma função da linguagem C em UML?

Esse problema pode ser solucionado com o auxílio de decisões contidas no Perfil UML (ou *UML Profile*). Um Perfil UML consiste em um conjunto de extensões

[1] O XMI é um formato para representação de modelos derivado do XML, bastante utilizado como forma de comunicação entre diversas ferramentas de modelagem.

à linguagem UML que utiliza alguns recursos como estereótipos (*stereotypes*) e valores marcados (*tagged values*). Estes recursos são utilizados para rotular elementos e denotar que o elemento possui alguma semântica em particular. Perfis UML podem ser desenvolvidos para várias plataformas, e eles constituem a base para a construção de ferramentas de modelagem. Porém, esta tecnologia ainda não está consolidada, e faltam ferramentas apropriadas para realizar seu uso.

3. ESTADO DA ARTE

Esta seção descreve o estado da arte em relação ao tema abordado nesse trabalho. Vale observar que dois temas recentes são o foco deste trabalho, e poucos trabalhos de pesquisa os abordam isoladamente. No caso da pesquisa adotada neste trabalho a situação ainda é mais precária porque é proposta uma combinação dos temas, MDD e GPGPU. Nas próximas seções serão descritos resumidamente os principais trabalhos relacionados.

3.1 COSA

O COSA (*Complementary Objects for Software Applications*) [14], é um ambiente de construção e execução de software, baseado em sinais e reativo. Uma aplicação no ambiente COSA é tipicamente concorrente. Um dos objetivos deste projeto diz respeito à produção de aplicações livres de *bugs*. Argumenta-se que a prática tradicional de desenvolvimento de software baseada em algoritmos é não-confiável. Assim, este projeto propõe uma mudança para um modelo visual, síncrono, e baseado em sinais, na tentativa de resolver este problema.

O modelo proposto pelo COSA tem a visão de um computador não como uma simples máquina para execução de instruções, mas sim como uma coleção de objetos que interagem entre si. Um objeto consiste em uma entidade de software que reside na memória do computador e que aguarda um sinal para realizar uma operação e emitir um sinal de saída.

O desenvolvimento de softwares no COSA consiste em conectar objetos elementares utilizando uma ferramenta de composição gráfica. Não existem procedimentos, sub-rotinas ou ciclos de compilação e execução. No COSA não existe sequer a necessidade de aprender alguma linguagem de programação. A Figura 3.1 ilustra uma estrutura em *looping* desenvolvida no ambiente COSA.

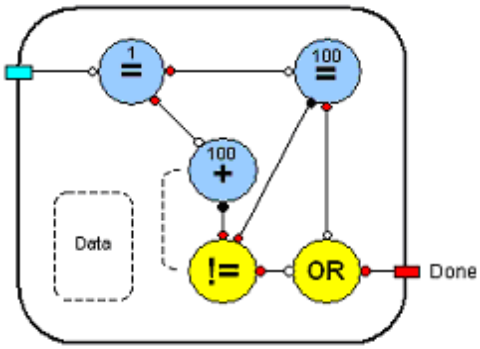


Figura 3.1 - Exemplo de *looping* em COSA.

3.2 Array-OL

O Array-OL [15] é uma linguagem para expressar aplicações paralelas, cujo foco consiste na modelagem de aplicações para processamento de sinais. Esta linguagem é baseada em um modelo que permite expressar tanto o paralelismo de tarefas (*task-level parallelism*) como o paralelismo de dados (*data-level parallelism*) de uma aplicação. Neste modelo, tarefas são conectadas umas as outras através de dependências de dados.

A descrição de uma aplicação em Array-OL utiliza dois modelos: o modelo global, que define uma seqüência entre as diferentes partes de uma aplicação, determinando assim, o paralelismo de tarefas; e o modelo local, que especifica ações elementares em cima dos dados, especificando o paralelismo de dados em cada tarefa. É importante ressaltar que o Array-OL é apenas uma linguagem de especificação, não existem regras para a execução de uma aplicação descrita nesta linguagem.

O modelo global (como ilustrado na Figura 3.2) consiste em um grafo acíclico direcionado. Cada nó neste grafo representa uma tarefa, e cada aresta representa um *array*. O número de *arrays* de entrada e de saída para um nó é ilimitado. Assim, uma tarefa pode, por exemplo, receber dois *arrays* bidimensionais como entrada e produzir um *array* tridimensional como saída. Existe apenas uma restrição, para cada *array*, pode existir apenas uma dimensão infinita, esta dimensão costuma ser utilizada para representar o tempo.

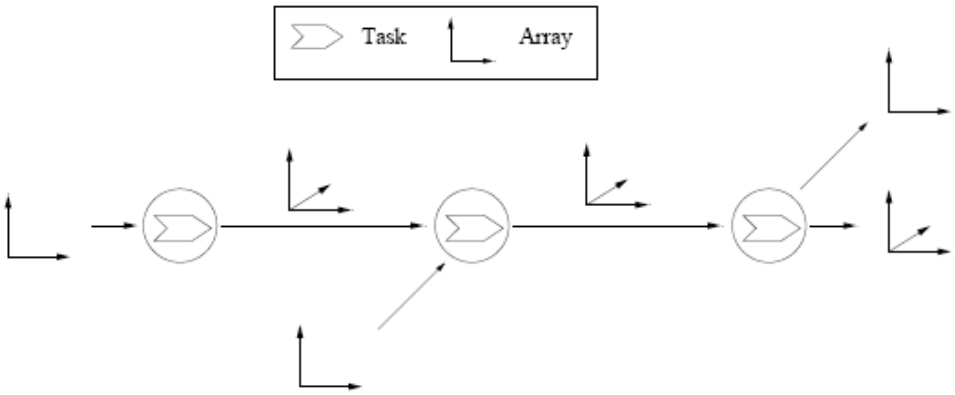


Figura 3.2 - Modelo global do Array-OL.

O modelo local (veja Figura 3.3) permite especificar como os *arrays* são consumidos e produzidos em uma dada tarefa, existindo o conceito de padrões (*patterns*), que são blocos extraídos de cada *array* de entrada, assim como os blocos resultantes da computação destes, que são armazenados nos *arrays* de saída de uma tarefa. Este processo ocorre de forma repetitiva, e o tamanho e a forma de cada padrão permanecem constantes durante todas as repetições.

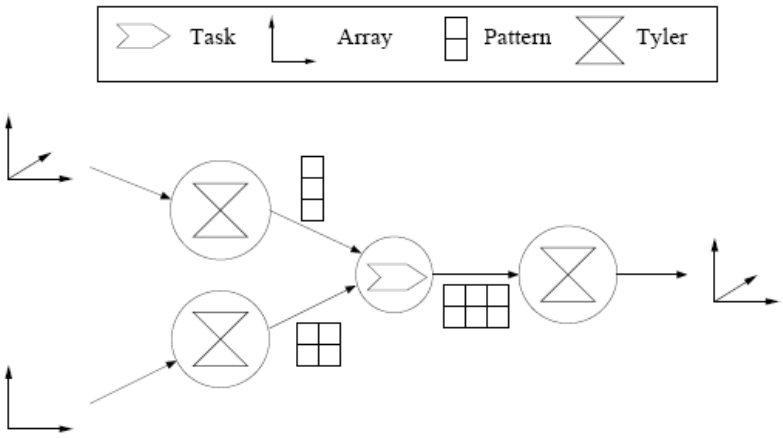


Figura 3.3 - Modelo local do Array-OL.

3.3 Gaspard2

Um dos ambientes que utilizam o Array-OL é o Gaspard2 [16], voltado para o design de sistemas SoC (*System-on-Chip*) orientado a modelos. Gaspard2 é

baseado em uma abordagem que a partir de dois modelos UML, um descrevendo a aplicação, e o outro descrevendo o hardware, é realizado um mapeamento da aplicação no hardware, através do chamado modelo de associação. Este modelo de associação é então projetado em descrições de baixo nível como modelos de simulação ou de síntese.

Assim como o Array-OL, o Gaspard2 é um ambiente dedicado a especificação de aplicações de processamento de sinal. Ele estende o Array-OL, permitindo, além da modelagem, a simulação, teste, e geração de código de aplicações SoC e arquiteturas de hardware. O Gaspard2 utiliza um Perfil UML [16] para modelagem das aplicações, e é capaz de gerar código para a plataforma SystemC [17].

3.4 Linguagens para GPGPU

Existem diversas linguagens para programação em GPUs. Estas linguagens costumam ser denominadas linguagens de programação *shading*. A seguir são descritas algumas dessas linguagens.

O Cg (*C for Graphics*) [18] é uma linguagem desenvolvida pela NVIDIA em colaboração com a Microsoft, para a programação dos *shaders*. Ela é baseada na linguagem C, com algumas modificações. Para o seu uso, existe um toolkit da própria NVIDIA, que contém diversas ferramentas, *plugins* para arte gráfica com suporte a aplicações como Maya e 3DS Max, bibliotecas, documentação e exemplos de *shaders*.

O Accelerator [19] é um sistema da Microsoft Research que visa fornecer um modelo de programação paralelo em alto nível, através de uma biblioteca acessível a partir de linguagens de programação tradicionais. O Accelerator traduz operações de dados paralelas para *shaders* em tempo real, obtendo ganhos significantes de desempenho sobre versões rodando em CPUs comuns.

O GLSL (*OpenGL Shading Language*) [20] é uma linguagem para programação de GPUs baseada em OpenGL. Nesta linguagem existem os *vertex shaders*, que são unidades que executam nos processadores de vértices de uma GPU (ou *vertex processors*), e os *fragment shaders*, que são unidades que executam nos processadores de fragmentos (ou *fragment processors*).

O HLSL (*High Level Shading Language*) [21] é uma linguagem também desenvolvida pela Microsoft, para o desenvolvimento de *shaders*. Esta linguagem funciona integrada ao DirectX, a partir da versão 9, assim, ela é voltada para a plataforma Windows.

O Brook [22] é uma extensão da linguagem ANSI C. Esta extensão contém conceitos de um modelo de programação denominado *stream*. Um *stream* conceitualmente significa um *array*, exceto pelo fato de que todos os seus elementos podem ser operados em paralelo através de funções conhecidas como *kernels*. Esta linguagem mapeia automaticamente *kernels* e *streams* em *shaders* e memórias de texturas. O Folding@Home [23], um projeto de computação distribuída para o estudo do comportamento de proteínas, é um exemplo de aplicação que utiliza o Brook.

CUDA também é um ambiente de programação que expõe os recursos de uma GPU ao programador. Pelo fato deste ser o ambiente utilizado neste trabalho, uma seção foi dedicada para descrevê-lo.

3.5 CUDA

CUDA (*Compute Unified Device Architecture*) [6] foi lançado pela NVIDIA em Novembro de 2007, para ser utilizado em conjunto com suas placas gráficas. Os primeiros processadores compatíveis com o CUDA são os processadores G80 da NVIDIA. CUDA consiste em um ambiente de programação que permite realizar computação de propósito geral utilizando a GPU.

O CUDA é composto por três componentes de software: o SDK, o Toolkit, e o *driver* gráfico. O SDK fornece exemplos de código, utilitários, e artigos para o auxílio no desenvolvimento de aplicações. O Toolkit contém o compilador, o *profiler*, e bibliotecas adicionais como o CUBLAS (*CUDA port of Basic Linear Algebra Subprograms*) e o CUFFT (*CUDA implementation of Fast Fourier Transform*).

O código CUDA de uma aplicação é armazenado em arquivos com a extensão '.cu'. Nestes arquivos são permitidos códigos similares a linguagem C, com algumas extensões, que são necessárias para lidar com detalhes relacionados à forma como uma GPU opera. Porém, a sintaxe básica da linguagem também é permitida.

Assim, aplicações CUDA podem ser facilmente integradas às aplicações já existentes, já que funções que sigam a sintaxe padrão podem ser invocadas por outras aplicações.

O modelo de programação de CUDA reflete a arquitetura de hardware descrita no Capítulo 2. Alguns conceitos que precisam ser introduzidos são *threads*, *blocks*, *grids* e *kernels* (vide Figura 3.4). *Threads* são as unidades de execução paralela em uma GPU. As *threads* são agrupadas em *blocks*, onde *threads* pertencentes a um mesmo *block* podem sincronizar sua execução e compartilhar um mesmo espaço de memória (*shared memory*). Um conjunto de *blocks* representa um *grid*. E um *kernel* consiste no código que é executado por uma *thread*. Cada chamada a um *kernel* precisa especificar uma configuração contendo o número de *blocks* em cada *grid*, o número de *threads* em cada *block* e opcionalmente a quantidade de memória compartilhada a ser alocada.

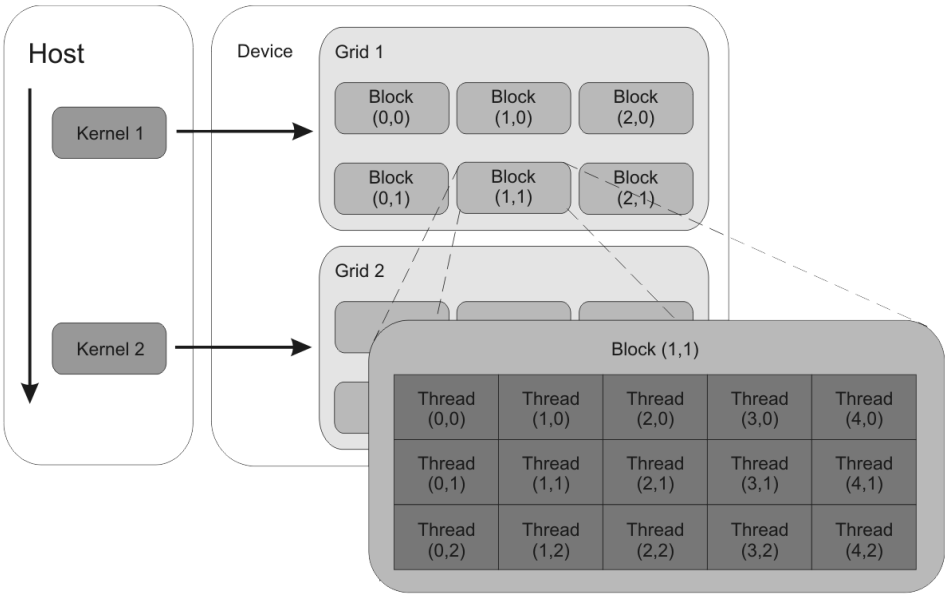


Figura 3.4 - Modelo de execução de CUDA.

Algumas palavras chaves são adicionadas a linguagem C. O conjunto dessas palavras chaves representa o escopo de uma função ou variável: `__host__`, `__device__` e `__global__`. O `__global__` é utilizado apenas para as declarações de *kernels*, porque o *kernel* é uma função que reside na GPU, mas é chamada a partir

da CPU. Os modificadores `__host__` e `__device__` são aplicados tanto a funções como a variáveis, para especificar sua localização, e de que ponto pode ser chamada (CPU ou GPU). Outra modificação na linguagem é a invocação dos *kernels*, a chamada de um *kernel* é seguida pela sua configuração entre os caracteres '<<<' e '>>>', como em:

```
kernel_name<<<gridDim, blockDim>>>(params);
```

A palavra chave `__shared__` foi introduzida para indicar que uma variável será alocada no espaço da memória compartilhada, e estará acessível apenas a *threads* de um mesmo *block*.

O modelo de software proposto pelo CUDA (veja Figura 3.5) enxerga a GPU como um co-processador de dados paralelo. Neste contexto, a GPU é considerada como o dispositivo (ou *device*) e a CPU como anfitrião (ou *host*).

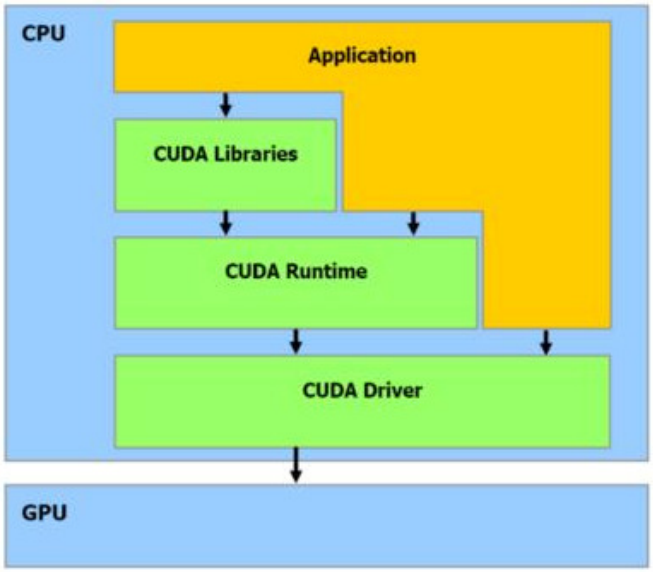


Figura 3.5 - Modelo de Aplicações CUDA.

A plataforma oferece três bibliotecas que podem ser utilizadas no desenvolvimento das aplicações. O *Common Runtime Library*, que provê alguns tipos de dados como vetores com duas, três ou quatro dimensões, e algumas funções matemáticas. O *Host Runtime Library*, que provê gerenciamento dos dispositivos e da memória, incluindo funções para alocar e liberar memória, e funções para invocar *kernels*. E o *Device Runtime Library*, que pode apenas ser utilizado pelos

dispositivos, e que provê funções específicas como sincronização de *threads*. CUDA é bastante utilizado em conjunto com o ambiente da ferramenta Microsoft Visual Studio [36], através da configuração das ferramentas do CUDA a este ambiente. A listagem a seguir ilustra um exemplo de código escrito em CUDA. O código consiste em um *kernel* para a soma de vetores.

```
1
2  __global__ void vecAdd(float* A, float* B, float* C)
3  {
4      int i = threadIdx.x;
5      C[i] = A[i] + B[i];
6  }
7
```

3.6 Ferramentas para MDD

Existem ferramentas que utilizam o conceito de orientação a modelos. Estas ferramentas costumam ser rotuladas como *MDA-Oriented Tools*, em referência ao principal processo que aplica esta forma de desenvolvimento. Os parágrafos a seguir descrevem algumas das ferramentas que utilizam este conceito.

O AndroMDA [24] é um *framework* MDA *open source*. Sua principal atividade é produzir, a partir de modelos de entrada descritos em um formato como o XMI, e de *plugins* de transformação do *framework*, alguns componentes customizados. Desde que exista o *plugin* adequado, é possível gerar componentes para qualquer plataforma com este *framework*. O AndroMDA é bastante utilizado para desenvolvimento no âmbito da plataforma J2EE [25], gerando componentes para as tecnologias Hibernate, Struts, EJB, Spring e WebServices.

O OptimalJ [26] é um produto comercial que utiliza o conceito de padrões para realizar as transformações para PSMs (tipo de modelo descrito na seção 2.4.2). Este produto possui uma ferramenta UML integrada para análise, mas utiliza uma notação estrutural diferente na parte MDA da ferramenta.

Além das ferramentas já citadas, existem aquelas que trabalham com modelagem UML em geral, como o IBM Rational Rose [27] ou o ArgoUML [28], e que também podem ser consideradas ferramentas MDA. Elas podem desempenhar

um importante papel no processo MDA, apesar de não terem sido desenvolvidas com este objetivo.

4. DESENVOLVIMENTO

Esta seção visa expor os detalhes de desenvolvimento do CudaMDA, apresentando as tecnologias utilizadas, e outras informações relevantes, como os requisitos especificados e a arquitetura da ferramenta.

4.1 Requisitos Especificados

A especificação dos requisitos de um software é de fundamental importância para que se construa um software corretamente [29]. Os requisitos elicitados para a ferramenta foram baseados no estudo das ferramentas orientadas a modelos descritas no Capítulo 3.

Os requisitos foram separados em três grupos de prioridades: o grupo dos requisitos essenciais, os quais são imprescindíveis à aplicação, sem eles a ferramenta não poderia entrar em funcionamento; o grupo dos requisitos importantes, sem eles a ferramenta consegue entrar em funcionamento, mas tem seu desempenho prejudicado; e o grupo dos requisitos desejáveis, cuja ausência não comprometeria a aplicação.

Todos os requisitos elicitados estão listados na Tabela 4.1. Os requisitos funcionais estão identificados com o prefixo RF e os requisitos não funcionais com o prefixo RNF:

Tabela 4.1 - Requisitos do CudaMDA.

Identificação	Requisito	Prioridade
RF01	Trabalhar com Modelos • O principal objeto da ferramenta são modelos visuais, a aplicação deve ser capaz de salvar, carregar, e exibir corretamente estes	Essencial

	modelos	
RF02	Modelos CUDA • Os modelos manipulados pela aplicação devem representar aplicações da plataforma CUDA	Essencial
RF03	Geração de Código Incompleto • A aplicação deve ser capaz de gerar pelo menos um esqueleto do código CUDA a partir dos modelos elaborados pela aplicação	Essencial
RF04	Geração de Código Completo • A aplicação deve ser capaz de gerar o código CUDA completo a partir dos modelos elaborados pela aplicação	Desejável
RF05	Execução do Modelo • A aplicação deve ser capaz de compilar, e executar um modelo (o código CUDA resultante) elaborado pela aplicação	Desejável
RNF01	Interface Gráfica Adequada • A interface gráfica da aplicação deve ser amigável, respeitando regras de usabilidade *(Afeta os Requisitos RF01/RF02)*	Importante
RNF02	Geração de Código Eficiente/Legível • O código gerado pela aplicação deve ser claro e não pode conter redundância *(Afeta os Requisitos RF03/RF04)*	Importante

Em resumo, os requisitos especificam o CudaMDA, como uma aplicação

para construção de aplicações CUDA que, a partir de modelos, é capaz de gerar a aplicação resultante deste modelo. Porém a geração do código completo da aplicação CUDA e a execução dos modelos são requisitos com menor prioridade, que só seriam implementados caso houvesse tempo suficiente. A implementação destes requisitos é considerada como uma futura extensão ao trabalho. A Figura 4.1 ilustra os passos no uso do CudaMDA.

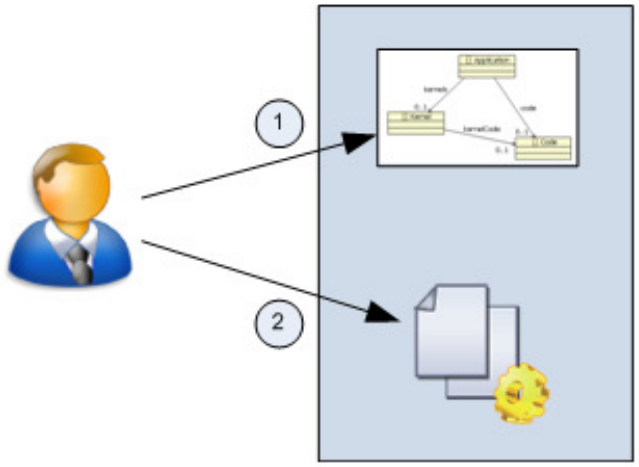


Figura 4.1 - Fluxo do CudaMDA.

No passo 1 o usuário elabora um modelo de sua aplicação, podendo opcionalmente, armazenar ou carregar um modelo no disco. Este passo equivale aos requisitos funcionais RF01 e RF02. No passo 2, o usuário gera o código, da aplicação CUDA equivalente ao modelo elaborado (RF03 e RF04).

4.2 Design do Software

Definida a especificação funcional do software, o próximo passo é a definição de uma arquitetura que atenda aos requisitos contidos nesta especificação. O objetivo desta seção é expor a arquitetura desenvolvida para o software.

A Figura 4.2 ilustra, no nível de componentes, a arquitetura desenvolvida para o CudaMDA.

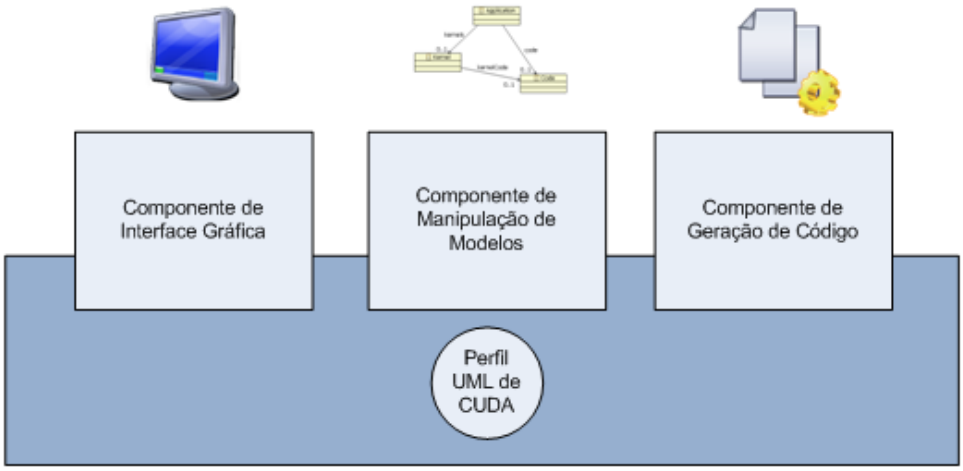


Figura 4.2 - Arquitetura do CudaMDA.

A Figura 4.2, apresenta os quatro componentes principais da ferramenta proposta:

- **Perfil UML de CUDA**: este componente da arquitetura é necessário aos demais componentes, pois especifica os elementos disponíveis nos modelos elaborados pela aplicação.

- **Componente de Interface Gráfica**: o objetivo deste componente é fornecer uma interface gráfica do software para o usuário, que exponha todas as funcionalidades da ferramenta, de forma adequada.

- **Componente de Manipulação de Modelos**: componente que contém as classes que representam os modelos, e que é capaz de armazená-los ou carregá-los do disco em algum formato apropriado.

- **Componente de Geração de Código**: sua função é receber um modelo como entrada e gerar o código da aplicação CUDA como saída.

As tecnologias utilizadas em cada componente e os detalhes de implementação serão abordados nas próximas subseções.

4.3 Implementação

Nesta subseção, é apresentada a implementação de cada componente da ferramenta. No desenvolvimento, foi utilizada a IDE de programação Eclipse [30]. O Eclipse é um ambiente de desenvolvimento projetado para programar na linguagem Java [31], porém, sua arquitetura fornece condições para que ele seja estendido a outras linguagens e funcionalidades. Isso é possível por ter sua arquitetura baseada no conceito de *plugins* (extensões que podem ser integradas ao ambiente). Os *plugins* desempenham um importante papel neste trabalho, pois alguns foram utilizados no desenvolvimento dos componentes, enquanto componentes da ferramenta foram implementados como *plugins* para o Eclipse.

4.3.1 Perfil UML

O EMF (*Eclipse Modeling Framework*) [32] é um *plugin* cuja funcionalidade consiste na definição e implementação de modelos de dados estruturados. No EMF, um modelo de dados é visto simplesmente como um conjunto de classes relacionadas que são utilizadas para manipular os dados do domínio de uma aplicação. O EMF inclui funcionalidades como: geração de código, onde todo o código necessário para manusear a informação contida em um modelo é gerado automaticamente; serialização, onde os modelos podem ser persistidos ou carregados a partir de arquivos XMI; e a geração de um editor gráfico para o modelo, incluindo recursos como tabelas de propriedades para editar as informações contidas no modelo. O resultado final do EMF é um *plugin* para o Eclipse, capaz de lidar com os modelos cuja definição foi fornecida como entrada.

O Perfil UML da ferramenta foi elaborado como um modelo do EMF. Este modelo foi baseado no modelo global da linguagem Array-OL, descrita na seção 3.1.2. A principal informação contida no modelo é o paralelismo de tarefas, característica existente nas aplicações com foco em paralelismo. Na Figura 4.3 é apresentado o modelo.

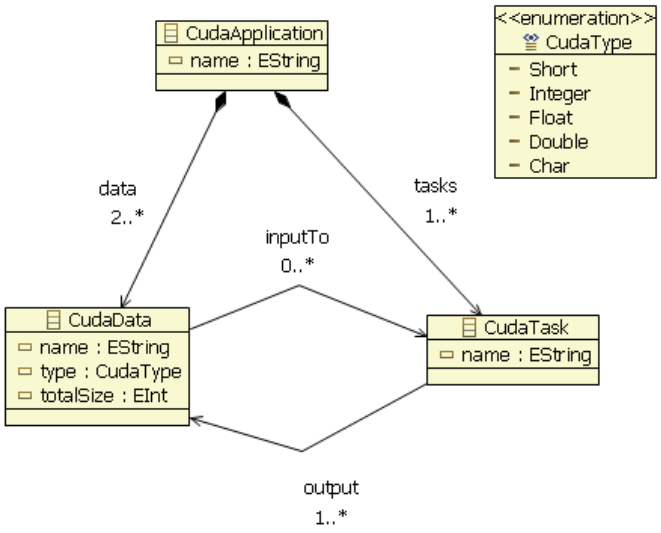


Figura 4.3 - Modelo EMF.

Os elementos básicos deste modelo são as *tarefas*, representadas pela classe `CudaTask`, que consistem em atividades que possuem potencial para serem realizadas paralelamente, e os *dados*, representados pela classe `CudaData`, que representam as informações que são manipuladas pelas tarefas. O modelo inclui os relacionamentos, `inputTo` e `output`, entre tarefas e dados, para explicitar as entradas e saídas de cada tarefa. Dependências entre tarefas podem ser representadas através do encadeamento dos dados de saída de uma tarefa para a entrada de outra tarefa. O modelo inclui ainda, para cada elemento, informações adicionais como o nome do dado ou tarefa, e no caso do dado, o seu tipo e tamanho. O conjunto destes elementos forma uma aplicação, representado pela classe `CudaApplication`, que consiste no resultado final de um modelo.

4.3.2 Componente de Manipulação de Modelos

O GMF (*Graphical Modeling Framework*) [33] é um *plugin*, contendo uma extensão ao EMF, que permite produzir editores gráficos mais complexos melhorando a qualidade da representação visual de um modelo. O GMF possui um processo de implementação bem definido, ilustrado na Figura 4.4.

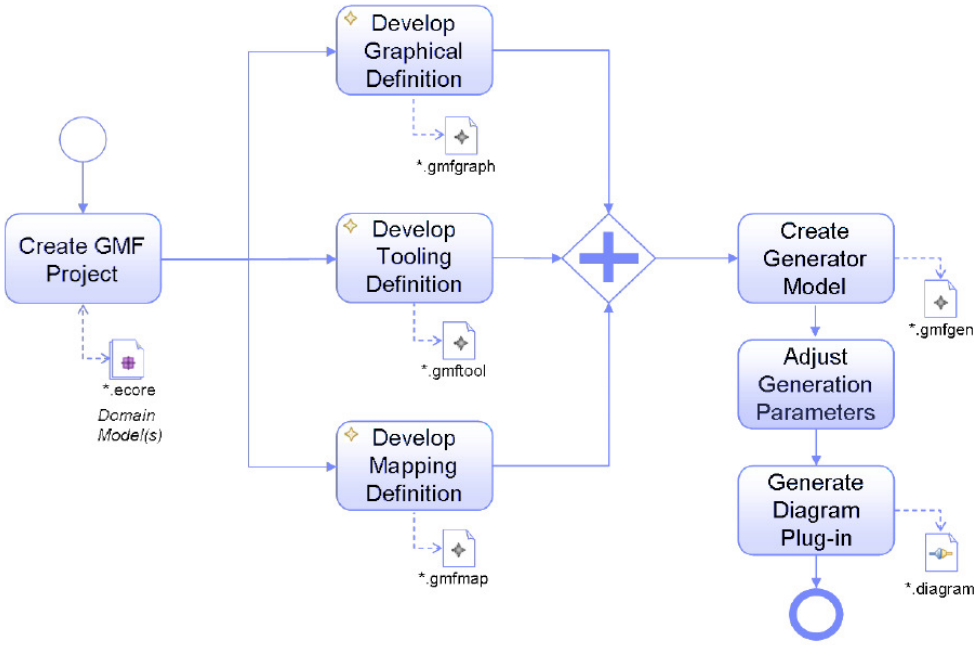


Figura 4.4- Visão geral do GMF.

O Modelo de Domínio é a definição dos elementos com os quais uma aplicação opera, um modelo EMF é válido neste caso. O Modelo de Definição Gráfica contém informações relacionadas aos elementos gráficos que estarão disponíveis no editor. O Modelo de Ferramenta é um modelo opcional em que se podem especificar opções como menus, paletas de ferramentas, botões, recursos para customizar ainda mais o editor resultante. A Definição de Mapeamento é o modelo que realiza a ligação entre os modelos definidos até então. Ele associa os elementos do domínio, os elementos gráficos, e os elementos da ferramenta. Fornecendo como entrada os modelos mencionados, o GMF é capaz de produzir um *plugin* contendo um editor visual para o Modelo de Domínio fornecido como entrada.

O Componente de Manipulação de modelos da Arquitetura, e parte do Componente de Interface Gráfica, foram implementados como um *plugin* gerado pelo GMF. O conteúdo dos modelos utilizados no processo pode ser encontrado no Apêndice A. O modelo mais relevante é o Modelo de Definição Gráfica, que define o aspecto visual dos diagramas. Para as tarefas, a imagem de um retângulo azul foi utilizada, enquanto que, para os dados, uma imagem de um círculo vermelho foi utilizada, como ilustra as Figuras 4.5 e 4.6.

Figura 4.5 - Representação de uma Tarefa no Modelo.

Figura 4.6 - Representação de um Dado no Modelo.

4.3.3 Componente de Geração de Código

O JET (*Java Emitter Templates*) [34] é um recurso contido no EMF para geração de código. No JET é utilizada uma sintaxe similar a sintaxe de JSP (*Java Server Pages*) para a criação de *templates* que especificam o código a ser gerado. A partir de um *template,* é produzida uma classe Java para representá-lo. Esta classe deve ser utilizada para a geração do código resultante. Qualquer tipo de código pode ser gerado através do JET, código SQL, XML, código Java, bastando criar o *template* apropriado.

O componente de geração de código foi implementado com cinco *templates* JET e algumas classes Java. O conteúdo dos *templates* JET pode ser encontrado no Apêndice B. As classes Java foram escritas para extrair a informação contida em um modelo elaborado no componente de manipulação de modelos e repassar essas informações aos *templates* para que o código CUDA seja gerado. As ações realizadas pelos *templates* consistem basicamente em:

- Mapeamento de cada tarefa existente no modelo para um *kernel* da linguagem CUDA.
- Geração de uma função de acesso, que encapsula a execução paralela das tarefas.

Os *kernels* gerados pelo mapeamento contêm em seu interior apenas comentários explicativos e o cálculo do índice de um elemento nos conjuntos de

dados. A idéia é que o usuário forneça o código específico de cada tarefa no corpo dos *kernels*. Mas, a lista de parâmetros dos *kernels* é gerada, de acordo com os dados de entrada e saída existentes para a tarefa no modelo.

A função de acesso gerada contém código em seu interior. Esse código realiza automaticamente as chamadas aos *kernels*, o gerenciamento de memória, a transferência de dados entre GPU e CPU, e o encadeamento da saída de um *kernel* para a entrada de outro, caso estes venham a ser dependentes.

No final deste processo, é gerado um arquivo com a extensão '.cu', contendo os *kernels* associados a cada tarefa, e a função utilizada para se executar as operações paralelas na GPU.

4.3.4 Componente de Interface Gráfica

A parte da interface gráfica relativa à edição de modelos é implementada através do *plugin* gerado pelo GMF, porém alguns recursos precisaram ser incluídos na interface para melhorar a experiência do usuário na utilização da ferramenta. Esta parte da interface foi implementada em outro *plugin*. Este *plugin* contém opções para acessar as funcionalidades da ferramenta, *wizards* para facilitar a criação de projetos e modelos dentro da ferramenta, além de customizações em alguns componentes gráficos, para diferenciá-los dos componentes comuns do Eclipse.

5. APLICAÇÃO

Como descrito no capítulo anterior, o resultado final deste trabalho de graduação foi o desenvolvimento de uma ferramenta composta por um conjunto de *plugins* para a plataforma Eclipse, o CudaMDA. Esta ferramenta contém um editor gráfico que permite que o usuário elabore modelos para representar aplicações GPGPU, e depois os transformem para código na linguagem CUDA.

Os quatro *plugins* que compõem a ferramenta são:

- `com.grvm.CudaMDA`
- `com.grvm.CudaMDA.edit`
- `com.grvm.CudaMDA.diagram`
- `com.grvm.CudaMDA.core`

O primeiro *plugin* contém os modelos utilizados como entrada no processo de geração da ferramenta através do EMF/GMF, e recursos gráficos necessários aos demais *plugins*. O segundo e o terceiro são gerados automaticamente pelo EMF/GMF e contém o código necessário para edição dos modelos. O último *plugin* contém os recursos adicionais de interface gráfica, e a parte de geração de código através dos *templates* JET.

A instalação da ferramenta segue o padrão tradicional de instalação de *plugins* no Eclipse. Copiam-se os quatro *plugins*, que consistem em arquivos '.jar' para a pasta 'plugins' localizada na pasta onde o Eclipse está instalado, e reinicia-se o Eclipse. Para começar um projeto, basta selecionar a opção para criação de um novo projeto, e ativar o *wizard* específico da ferramenta. Este *wizard* resulta na criação de um novo projeto dentro do ambiente Eclipse, contendo duas pastas: 'model' e 'output', os modelos devem ser criados na primeira pasta, e a segunda pasta é utilizada para armazenar o código gerado pela ferramenta.

Como principal funcionalidade, a maior parte da interface gráfica da ferramenta é voltada para a elaboração dos modelos. Para iniciar esta atividade, existe um *wizard* específico. Este *wizard* resulta na criação de dois arquivos: um com a extensão '.cuda' e outro com a extensão '.cuda_diagram'. O primeiro consiste na informação contida em um modelo, no formato XMI, e o segundo consiste no próprio

modelo. Para editar o modelo, basta selecionar a edição do arquivo com a extensão '.cuda_diagram'.

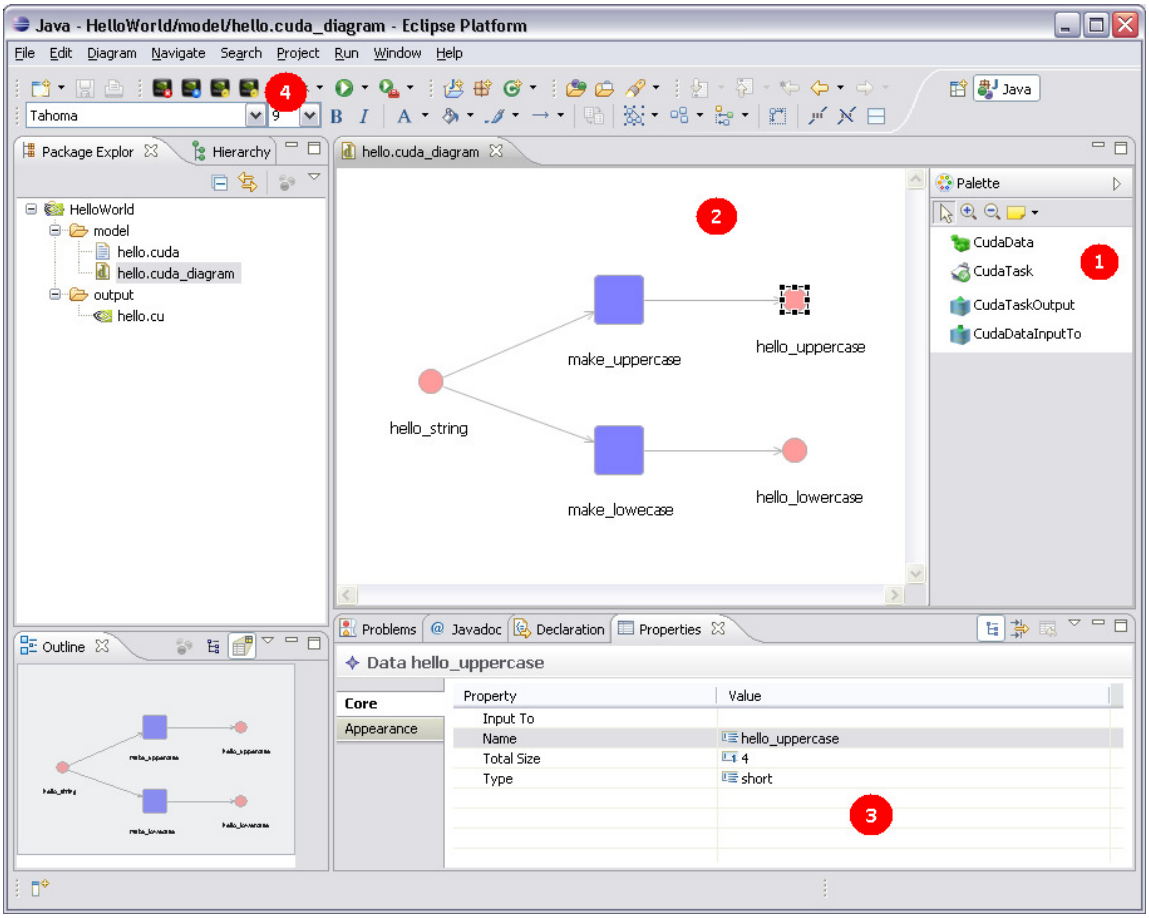


Figura 5.1 - Interface gráfica do CudaMDA.

A Figura 5.1 apresenta a interface principal do CudaMDA. No (1), está a paleta contendo as opções disponíveis para incluir em um modelo: dados, tarefas, e conexões de entrada e saída para uma tarefa. No (2), está a representação visual do modelo. Em (3), está a tabela de propriedades, que permite visualizar e alterar informações do elemento selecionado no momento. No (4), estão os botões que permitem a geração do código do modelo que está sendo editado, e ações auxiliares como a exclusão de qualquer código anteriormente gerado. Utilizando esses recursos o usuário é capaz de elaborar um modelo para sua aplicação paralela.

O próximo passo após o desenvolvimento do modelo é a geração do código. Para realizar esta atividade basta pressionar o botão associado a esta funcionalidade

na interface. O código será gerado na pasta 'output' do projeto, com o mesmo nome do modelo, e pode ser visualizado pela ferramenta, como ilustra a Figura 5.2:

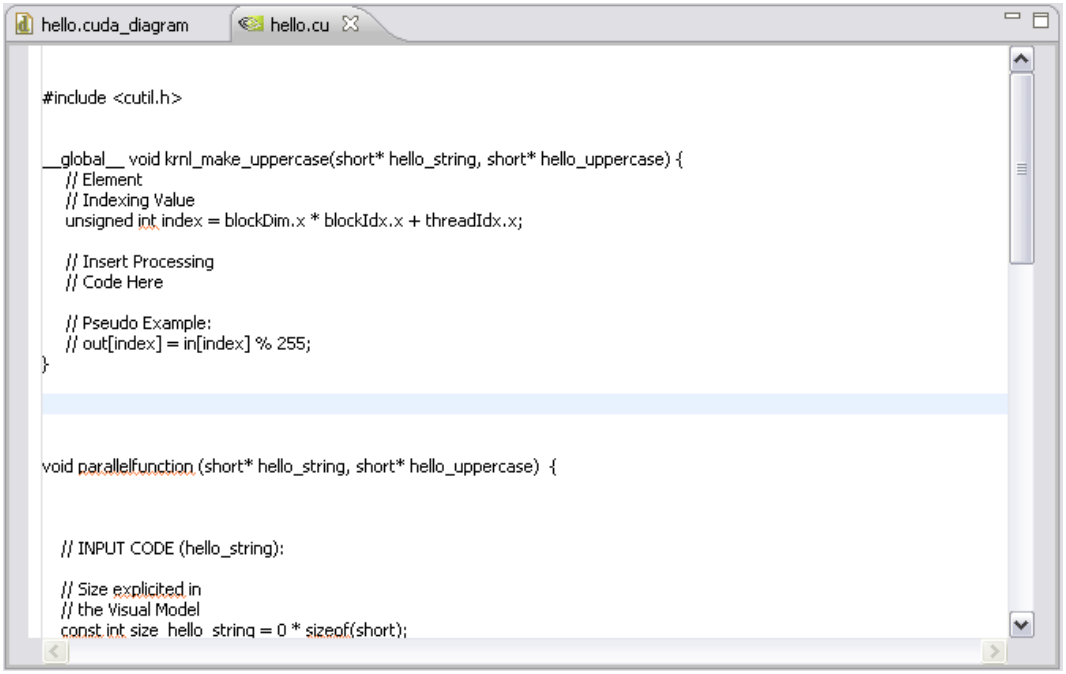


Figura 5.2 – Visualização de um Arquivo CUDA produzido pela ferramenta.

O código gerado pela ferramenta traduz a informação contida no modelo, de acordo com o mapeamento descrito na seção 4.3.3, que explica a geração de código. No próximo Capítulo, será discutida a utilização da aplicação em um cenário específico.

6. RESULTADOS

Para validar a ferramenta desenvolvida neste trabalho, é apresentado um cenário de utilização. Esse cenário compreende a construção de um software para descoberta de senhas[1], onde são comparados os esforços na construção deste software, sem o auxílio da ferramenta, e com o seu auxílio.

6.1 Descrição do Cenário

O armazenamento de senhas nos sistemas modernos utiliza um tipo de função conhecida como *função hash* [35]. Esta função é aplicada à senha do usuário para gerar outro valor que o represente. Este valor gerado é então armazenado em um banco de dados, por exemplo. A peculiaridade desta função, é que a partir desse valor produzido, não existe maneira simples de obter a senha que a gerou. Isto evita que pessoas mal intencionadas que obtenham acesso ao banco de dados das senhas, consigam descobrir senhas alheias, por exemplo.

Quando é necessário realizar a comparação entre uma senha fornecida e a senha armazenada, ela é realizada através dos seus valores obtidos através desta função. Caso eles coincidam, a senha é considerada válida. Funções eficientes fazem com que a probabilidade de duas senhas diferentes conterem o mesmo valor seja muito pequena.

Uma categoria de sistemas para descoberta de senhas funciona gerando todas as possibilidades de senha existentes, e comparando o seu valor com o da senha que se deseja descobrir. Neste tipo de sistema, assume-se que é mais fácil descobrir o valor da *função hash* de uma senha (de forma ética ou não), do que descobrir a própria senha. Este tipo de sistema é conhecido como sistema de força bruta, pois ele não emprega nenhuma heurística na busca pela senha desejada.

Para exemplificar, este sistema pode ser implementado através de um algoritmo que recebe como entrada o valor *hash* da senha que se deseja descobrir, e iterar todas as possibilidades de senhas existentes, comparando o seu valor *hash* com

[1] Apesar de ser uma atividade ilegal, este cenário fornece boas condições para aplicar os conceitos de paralelismo.

o valor recebido como entrada. Essa iteração ocorre até encontrar uma senha cujo valor coincida com o valor de entrada.

Como as possibilidades de senha são inúmeras, esta é uma tarefa que demanda bastante tempo, se realizada de forma seqüencial. Uma forma de aumentar o desempenho é utilizar os recursos computacionais das GPUs para executar as comparações em paralelo. Esta é uma situação adequada para o uso de GPU, porque as comparações podem ser realizadas sem nenhuma dependência das demais. Se a situação é adequada para o uso de GPU, há a possibilidade de usar a ferramenta desenvolvida neste trabalho para facilitar a construção do algoritmo em CUDA.

6.2 Versão desenvolvida através da Ferramenta

Assim, podemos utilizar as funcionalidades da ferramenta desenvolvida para modelar o algoritmo e gerar a maior parte do código necessário para executá-lo em uma GPU.

O modelo gerado para o problema é apresentado na Figura 6.1:

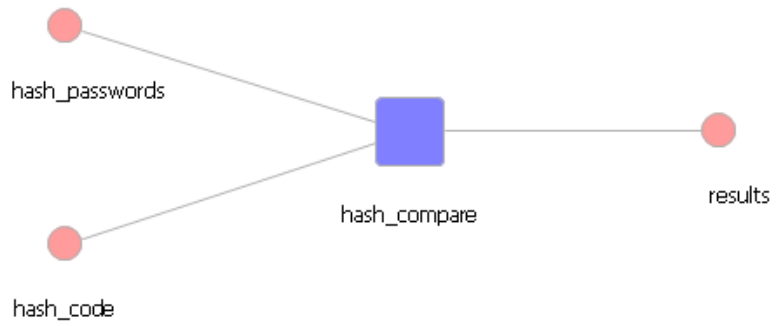


Figura 6.1 - Modelo do Algoritmo elaborado na Ferramenta.

O modelo especifica uma tarefa a ser executada em paralelo representando a comparação do valor do algoritmo *hash* das senhas. A tarefa recebe como entrada os valores *hash* das possibilidades de senhas no qual se deseja encontrar a senha correta, e produz como saída, valores booleanos indicando se alguma destas possibilidades é a senha desejada. Para simplificar, estamos abstraindo a geração das possibilidades de senhas. Assume-se que elas são produzidas pelo usuário e os valores *hash* repassados ao código a ser gerado pela ferramenta. O código gerado pela ferramenta para este modelo é apresentado na listagem a seguir.

```
1
2  #include <cutil.h>
3
4  __global__ void krnl_hash_compare(int* hash_code, int* hash_passwords, int* results) {
5       // Element
6       // Indexing Value
7       unsigned int index = threadIdx.x;
8
9       // Insert Processing
10      // Code Here
11
12 }
13
14 void function_hash_compare(int* hash_code, int* hash_passwords, int* results)  {
15
16     // INPUT CODE (hash_code):
17
18     // Size explicited in
19     // the Visual Model
20     const int size_hash_code = 1 * sizeof(int);
21
22     // Allocating Device
23     // Memory to the Input
24     int* device_hash_code = NULL;
25     cudaMalloc((void **) &device_hash_code, size_hash_code);
26
27     // Transferring
28     // Input Data to Device
29     cudaMemcpy(device_hash_code, hash_code, size_hash_code, cudaMemcpyHostToDevice);
30
31     // INPUT CODE (hash_passwords):
32
33     // Size explicited in
34     // the Visual Model
35     const int size_hash_passwords = 32 * sizeof(int);
36
37     // Allocating Device
38     // Memory to the Input
39     int* device_hash_passwords = NULL;
40     cudaMalloc((void **) &device_hash_passwords, size_hash_passwords);
41
42     // Transferring
43     // Input Data to Device
44     cudaMemcpy(device_hash_passwords, hash_passwords, size_hash_passwords, cudaMemcpyHostToDevice);
45
46     // KERNEL CALL (krnl_hash_compare):
47
48     // Size explicited in
49     // the Visual Model
50     const int size_results = 32 * sizeof(int);
51
52     // Allocating Device
53     // Memory to Kernel Output
54     int* device_results = NULL;
55     cudaMalloc((void **) &device_results, size_results);
56
57     // Calling
58     // the Kernel
59     krnl_hash_compare<<<1, 32>>>(device_hash_code, device_hash_passwords, device_results);
60
61     // Waiting
62     // Kernel Execution
63     cudaThreadSynchronize();
64
65     // OUTPUT CODE (results):
66
67     // Transferring Output Data
68     // from Device to Host
69     cudaMemcpy(results, device_results, size_results, cudaMemcpyDeviceToHost);
70
71 }
72
```

O código contém um *kernel* (linhas 4 a 12) para realização da tarefa em paralelo, e o código da função de acesso (linhas 14 a 71) que realiza a operação de forma paralela. Ainda assim, é necessário que o usuário forneça o código para completar o *kernel* realizando a comparação dos *hashs*, e armazenando o valor no dado de saída. Este código é similar ao apresentado nas linhas 9 a 14 da listagem a seguir.

```
4  __global__ void krnl_hash_compare(int* hash_code, int* hash_passwords, int* results) {
5      // Element
6      // Indexing Value
7      unsigned int index = threadIdx.x;
8
9      // Comparating
10     // Hashcodes
11     if (hash_passwords[index] == hash_code[0])
12         results[index] = 1;
13     else
14         results[index] = 0;
15
16 }
```

6.3 Discussão

É notável a diferença entre o código que um programador deveria escrever caso não utilizasse a ferramenta, e o código que ele escreveria utilizando-a. Além da questão de quantidade de código, há o aspecto de aprendizagem de uma nova tecnologia, no caso, CUDA. Sem o uso da ferramenta, seria necessário entender novos conceitos e recursos, o que demanda tempo, para poder escrever o código. Já no caso da utilização da ferramenta, não há necessidade explicita dessa aprendizagem, já que os únicos trechos de código que ele precisa escrever são os relacionados às tarefas que foram especificadas por ele, e nesses trechos não há a necessidade de utilizar os recursos de CUDA, a sintaxe padrão da linguagem C deve ser utilizada.

Apesar da utilidade desta ferramenta, a mesma ainda contém algumas restrições. Além de conter alguns erros de interface, a ferramenta está restrita apenas a dados primitivos, não permitindo, por exemplo, o uso de estruturas complexas como parâmetros para as tarefas.

Outro aspecto é que a ferramenta foi desenvolvida com foco na corretude do código gerado. Aspectos de performance, como otimização do código gerado, são bastante importantes no desenvolvimento em GPGPU, mas não foram considerados.

7. CONCLUSÃO E TRABALHOS FUTUROS

O presente trabalho demonstrou o uso do desenvolvimento baseado em modelos aplicado à área de GPGPU, através da implementação do CudaMDA, uma ferramenta que automatiza parte do processo de construção de um software nesta plataforma através de modelos. Os passos do desenvolvimento da ferramenta foram abordados no trabalho: requisitos especificados; arquitetura elaborada; e detalhes de implementação. Os conceitos fundamentais relacionados aos assuntos foram introduzidos nos capítulos iniciais deste trabalho, enquanto o resultado foi apresentado no capítulo de mesmo nome.

A principal contribuição da área de Engenharia de Software foi atingida neste trabalho, ou seja, o aumento de produtividade na elaboração de aplicações, conseqüência da redução no tempo para desenvolvimento de uma aplicação usando a ferramenta CudaMDA. A abordagem orientada a modelos adotada neste trabalho fornece uma visão de alto nível da aplicação, diminuindo a probabilidade de geração de erros quando se utiliza programação em baixo nível, tipicamente códigos não triviais.

Uma possível extensão ao trabalho seria abordar a automatização na construção de softwares em outras plataformas de programação, como Java ou C++. O refinamento do modelo do domínio utilizado na ferramenta, com o objetivo de aperfeiçoar a representação de uma aplicação na plataforma GPGPU pode ser considerada uma extensão à pesquisa desenvolvida neste trabalho. Abordar os aspectos de performance do código gerado é um trabalho futuro. Outra opção é a extensão da ferramenta CudaMDA, incluindo os requisitos não implementados como a geração de código CUDA completo e a execução do modelo. A interface gráfica do *plugin* precisa ser avaliada do ponto de vista de usabilidade, e se for o caso, adaptada após esta avaliação.

8. REFERÊNCIAS

[1] Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A. E., and Purcell, T. 2005. **A survey of general-purpose computation on graphics hardware**. In Proceedings of Eurographics 2005, State of the Art Reports. 21--51.

[2] General-Purpose GPU. http://www.gpgpu.org. Acesso em: Agosto de 2008.

[3] Stephen J. Mellor, Anthony N. Clark, Takao Futagami, **Guest Editors' Introduction: Model-Driven Development**, IEEE Software, vol. 20, no. 5, pp. 14-18, Sep/Oct, 2003.

[4] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. **GPU computing**. Proceedings of the IEEE, 96(5), May 2008.

[5] Luebke, D., Humphreys, G. **How GPUs Work**. *Computer* , vol.40, no.2, pp.96-100, Feb. 2007.

[6] NVIDIA. **Compute Unified Device Architecture: Programming Guide**. http://developer.download.nvidia.com/compute/cuda/1_0/NVIDIA_CUDA_Programming_Guide_1.0.pdf, acesso em: Novembro de 2008.

[7] Rumbaugh, J., Jacobson, I., and Booch, G. **The UML Reference Manual**. Addison-Wesley, Reading, England, 1999.

[8] Selic, B., **The pragmatics of model-driven development**, Software, IEEE , vol.20, no.5, pp. 19-25, Sept.-Oct. 2003.

[9] J. Miller and J. Mukerji, **MDA Guide Version 1.0.1**, doc. no. omg/2003-06-01, June 2003. http://www.omg.org/cgi-bin/apps/doc?omg/03-06-01.pdf.

[10] Shirley, P. 2002. **Fundamentals of Computer Graphics**. A. K. Peters, Ltd.

[11] Pharr, M. and Fernando, R. 2005 **GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)**. Addison-Wesley Professional.

[12] Object Management Group. **Meta Object Facility Specification**, Version 1.3, September, 1999. http://www.dstc.edu.au/Research/Projects/MOF/rtf/.

[13] Object Management Group, **XML Metadata Interchange Specification**, Version 1.1, http://www.omg.org/.

[14] Rebel Science. 2006. **COSA Project**. http://www.rebelscience.org/Cosas/COSA.htm.

[15] Boulet, P.: **Array-OL revisited, multidimensional intensive signal processing specification**. Research Report RR-6113, INRIA (2007).

[16] Ben Atitallah, R., Boulet, P., Cuccuru, A., Dekeyser, J.L., Honor´e, A., Labbani, O., Le Beux, S., Marquet, P., Piel, E., Taillard, J., Yu, H.: **Gaspard2 uml profile documentation**. Technical Report 0342, INRIA (2007)

[17] Open SystemC Initiative, **SystemC**. http://systemc.org/.

[18] Fernando, R. and Kilgard, M. J. 2003. **The Cg Tutorial: the Definitive Guide to Programmable Real-Time Graphics**. Addison-Wesley Longman Publishing Co., Inc.

[19] Tarditi, D., Puri, S., and Oglesby, J. 2006. **Accelerator: using data parallelism to program GPUs for general-purpose uses**. SIGOPS Oper. Syst. Rev. 40, 5 (Oct. 2006), 325-335.

[20] Rost, R. J. 2004. **Opengl(R) Shading Language**. Addison Wesley Longman Publishing Co., Inc.

[21] Microsoft MSDN. 2006. **HLSL**. http://msdn.microsoft.com/en-us/library/bb509561(VS.85).aspx.

[22] Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., and Hanrahan, P. 2004. **Brook for GPUs: Stream Computing on Graphics Hardware**. In Proceedings of SIGGRAPH.

[23] S. M. Larson, C. D. Snow, M. Shirts, and V. S. Pande. **Folding@Home and Genome@Home: Using distributed computing to tackle previously intractable problems in computational biology**. Computational Genomics (to appear).

[24] AndroMDA.org. 2008. **AndroMDA**. http://www.andromda.org/.

[25] Deepak Alur, John Crupi and Dan Malks. **Core J2EE Patterns**. Sun Microsystems Press, ISBN 0-13-064884-1, 2001.

[26] Compuware. 2008. **OptimalJ**. http://www.compuware.com/.

[27] Quatrani, T. 1998. **Visual Modeling with Rational Rose and UML**. Addison-Wesley Longman Publishing Co., Inc.

[28] Tigris. 2006. **ArgoUML**. http://argouml.tigris.org/.

[29] Davis, A. M. 1990. **Software Requirements: Analysis and Specification**. Prentice Hall Press.

[30] Eclipse Foundation. 2004. **Eclipse**. http://www.eclipse.org/.

[31] Arnold, K. and Gosling, J. 1998. **The Java Programming Language (2nd Ed.)**. ACM Press/Addison-Wesley Publishing Co.

[32] Eclipse Foundation. **Eclipse Modeling Framework (EMF)**.

http://www.eclipse.org/emf.

[33] Eclipse Consortium. 2005. **Eclipse Graphical Modeling Framework (GMF)**. http://www.eclipse.org/gmf.

[34] Eclipse Foundation (2007a), **'Eclipse Modeling: Java Emitter Templates'**. http://www.eclipse.org/emft/projects/jet/.

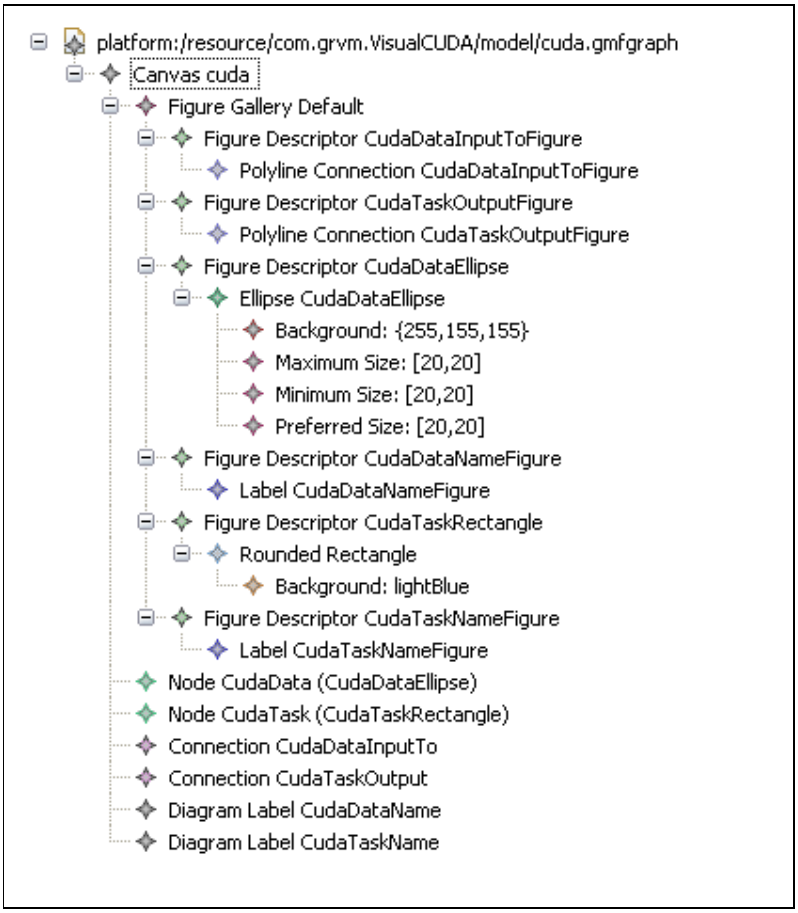
[35] Menezes, A. J., Vanstone, S. A., and Oorschot, P. C. 1996 **Handbook of Applied Cryptography**. 1st. CRC Press, Inc.

[36] Microsoft. 2005. **Microsoft Visual Studio**.

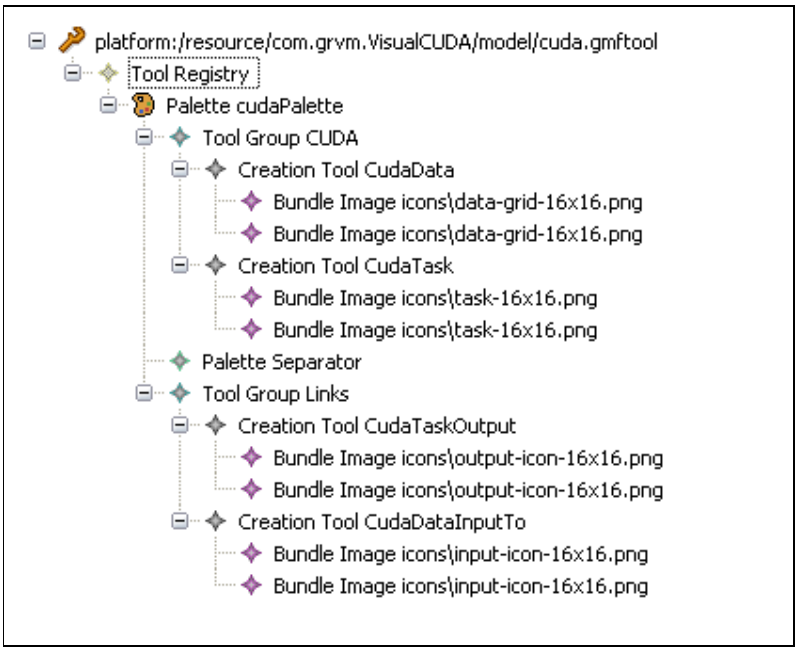
APÊNDICE A – MODELOS GMF

Neste apêndice estão contidos os modelos utilizados no processo de geração do componente de manipulação de modelos, através do GMF. A seguir estão os modelos visualizados no Eclipse:

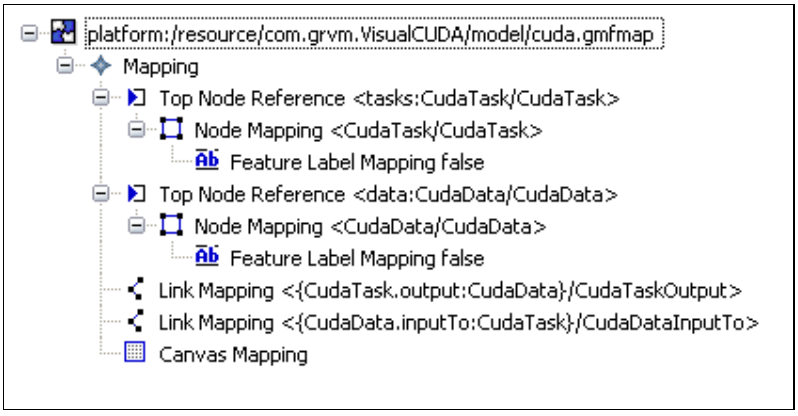
Modelo de Definição Gráfica (cuda.gmfgraph):



Modelo de Ferramenta (cuda.gmftool):



Definição de Mapeamento (cuda.gmfmap):



APÊNDICE B – TEMPLATES JET

Neste apêndice estão contidos os *templates* JET utilizados na geração de código CUDA. A seguir estão os cinco *templates*:

kernel_body.jet:

```
<%@ jet package="cuda.transform.template"
        imports="java.util.* cuda.model.code.*"
        class="KernelBodyTemplate" %>

<% Kernel kernel = (Kernel)argument; %>

__global__ void <%= kernel.getName() %>(<%

for (Iterator<String> iterator = kernel.getInOutNames(); iterator.hasNext(); ) {
    String argumentName = iterator.next();
    String argumentType = kernel.getInOutType(argumentName);

%><%= argumentType %>* <%= argumentName %><%

 if (iterator.hasNext()) { %>, <% }  } %>) {
     // Element
     // Indexing Value
     unsigned int index = threadIdx.x;

     // Insert Processing
     // Code Here

     // Pseudo Example:
     // out[index] = in[index] % 255;
}
```

main.jet:

```
<%@ jet package="cuda.transform.template"
        imports="java.util.* cuda.model.code.*"
        class="MainTemplate" %>

<% Program program = (Program)argument; %>

void <%= program.getName() %> (<%

for (Iterator<String> iterator = program.getInOutNames(); iterator.hasNext(); ) {
    String argumentName = iterator.next();
    String argumentType = program.getInOutType(argumentName);

%><%= argumentType %>* <%= argumentName %><%

 if (iterator.hasNext()) { %>, <% }  } %>)  {
```

main_init.jet:

```
1  <%@ jet package="cuda.transform.template"
2          imports="java.util.* cuda.model.code.*"
3          class="MainInitTemplate" %>
4
5  <% Program program = (Program)argument; %>
6  <%
7
8  for (Iterator<String> iterator = program.getInputNames(); iterator.hasNext(); ) {
9      String argumentName = iterator.next();
10     String argumentType = program.getInOutType(argumentName);
11     int argumentSize = program.getSize(argumentName);
12
13
14 %>
15     // INPUT CODE (<%= argumentName %>):
16
17     // Size explicited in
18     // the Visual Model
19     const int size_<%= argumentName %> = <%= argumentSize %> * sizeof(<%= argumentType %>);
20
21     // Allocating Device
22     // Memory to the Input
23     <%= argumentType %>* device_<%= argumentName %> = NULL;
24     cudaMalloc((void **) &device_<%= argumentName %>, size_<%= argumentName %>);
25
26     // Transferring
27     // Input Data to Device
28     cudaMemcpy(device_<%= argumentName %>, <%= argumentName %>, size_<%= argumentName %>, cudaMemcpyHostToDevice);
29 <%
30
31  if (iterator.hasNext()) {
32
33  %>
34
35 <% }  } %>
```

kernel_call.jet:

```
<%@ jet package="cuda.transform.template"
        imports="java.util.* cuda.model.code.*"
        class="KernelCallTemplate" %>

<% Kernel kernel = (Kernel)argument; %>

    // KERNEL CALL (<%= kernel.getName() %>):

<%
for (Iterator<String> iterator = kernel.getOutputNames(); iterator.hasNext(); ) {
    String argumentName = iterator.next();
    String argumentType = kernel.getInOutType(argumentName);
    int argumentSize = kernel.getSize(argumentName);
%>

    // Size explicited in
    // the Visual Model
    const int size_<%= argumentName %> = <%= argumentSize %> * sizeof(<%= argumentType %>);

    // Allocating Device
    // Memory to Kernel Output
    <%= argumentType %>* device_<%= argumentName %> = NULL;
    cudaMalloc((void **) &device_<%= argumentName %>, size_<%= argumentName %>);

<%
 }
 %>

    // Calling
    // the Kernel
    <%= kernel.getName() %><<<<%= kernel.getMaxSize() %>, 1>>>(<%

for (Iterator<String> iterator = kernel.getInOutNames(); iterator.hasNext(); ) {
    String argumentName = iterator.next();
%>device_<%= argumentName %><%
 if (iterator.hasNext()) { %>, <% }  }
 %>);

    // Waiting
    // Kernel Execution
    cudaThreadSynchronize();
```

main_end.jet:

```
<%@ jet package="cuda.transform.template"
        imports="java.util.* cuda.model.code.*"
        class="MainEndTemplate" %>

<% Program program = (Program)argument; %>
<%

for (Iterator<String> iterator = program.getOutputNames(); iterator.hasNext(); ) {
    String argumentName = iterator.next();

%>
    // OUTPUT CODE (<%= argumentName %>):

    // Transferring Output Data
    // from Device to Host
    cudaMemcpy(<%= argumentName %>, device_<%= argumentName %>, size_<%= argumentName %>, cudaMemcpyDeviceToHost);
<%

 if (iterator.hasNext()) {

 %>

<% }  } %>

    // Remeber to Free
    // the Device Memory
```