**UNIVERSIDADE FEDERAL DE SANTA MARIA**
**CENTRO DE TECNOLOGIA**
**CURSO DE CIÊNCIA DA COMPUTAÇÃO**

# UMA FERRAMENTA PARA INTEGRAÇÃO DE APLICAÇÕES JAVA À COMPUTAÇÃO PARALELA DE PROPÓSITO GERAL EM PROCESSADORES GRÁFICOS

**TRABALHO DE GRADUAÇÃO**

**Bruno de Carvalho Christo**

**Santa Maria, RS, Brasil**

**2008**

# UMA FERRAMENTA PARA INTEGRAÇÃO DE APLICAÇÕES JAVA À COMPUTAÇÃO PARALELA DE PROPÓSITO GERAL EM PROCESSADORES GRÁFICOS

**por**

**Bruno de Carvalho Christo**

Trabalho de Graduação apresentado ao Curso de Ciência da Computação
da Universidade Federal de Santa Maria (UFSM, RS), como requisito
parcial para a obtenção do grau de
**Bacharel em Ciência da Computação**

**Orientador: Profª Andrea Schwertner Charão**

**Trabalho de Graduação N. 267**

**Santa Maria, RS, Brasil**

**2008**

**Universidade Federal de Santa Maria**
**Centro de Tecnologia**
**Curso de Ciência da Computação**

A Comissão Examinadora, abaixo assinada,
aprova o Trabalho de Graduação

**UMA FERRAMENTA PARA INTEGRAÇÃO DE APLICAÇÕES JAVA À COMPUTAÇÃO PARALELA DE PROPÓSITO GERAL EM PROCESSADORES GRÁFICOS**

elaborado por
**Bruno de Carvalho Christo**

como requisito parcial para obtenção do grau de
**Bacharel em Ciência da Computação**

**COMISSÃO EXAMINADORA:**

**Profª Andrea Schwertner Charão**
(Presidente/Orientador)

**Prof. Benhur de Oliveira Stein (UFSM)**

**Profª. Iara Augustin (UFSM)**

Santa Maria, 17 de Dezembro de 2008.

*“Se deres um peixe a um homem, ele alimentar-se-á uma vez; se o ensinares a pescar, alimentar-se-á durante toda a vida.”*
— KUAN-TSU

*“O descontentamento é o primeiro passo na evolução de um homem ou de uma nação.”*
— OSCAR WILDE

*“O saber é a razão de ser da existência do homem na terra, a primeira e última de suas tarefas. Faça com que o estímulo de consegui-lo vibre em você permanentemente, porque nele está a verdadeira finalidade de sua vida.”*
— DO LIVRO BASES PARA A SUA CONDUTA

# AGRADECIMENTOS

Gostaria de agradecer a todos aqueles que de alguma forma contribuíram para a elaboração deste trabalho.

Aos meus familiares e à minha namorada por terem me apoiado sempre e aguentado o meu mau-humor durante o final do curso,

Aos professores, por despertarem a minha curiosidade em busca de novos conhecimentos. Mais importante do que me ensinar, me ensinaram a aprender. Agradeço especialmente à professora Andrea Charão, não só pela excelente orientação ao longo do curso, mas também por ter estado sempre disposta a me ajudar até nos momentos mais difíceis, e nunca ter se contentado com o pouco que a minha preguiça insiste em fazer, sabendo que eu sempre poderia mais,

A todos os colegas que contribuíram para a minha formação, em especial o Bruno Dalmazo, o Francisco Avelar e o Gustavo Mora. Formamos uma verdadeira equipe durante o curso, e sem eles não sei se estaria me formando agora.

Enfim, o meu muito obrigado a todos.

# RESUMO

Trabalho de Graduação
Curso de Ciência da Computação
Universidade Federal de Santa Maria

**UMA FERRAMENTA PARA INTEGRAÇÃO DE APLICAÇÕES JAVA À COMPUTAÇÃO PARALELA DE PROPÓSITO GERAL EM PROCESSADORES GRÁFICOS**

Autor: Bruno de Carvalho Christo
Orientador: Profª Andrea Schwertner Charão
Local e data da defesa: Santa Maria, 17 de Dezembro de 2008.

O objetivo deste trabalho é desenvolver uma ferramenta para integração de aplicações Java com computação paralela de propósito geral em processadores gráficos (*General Purpose Computation on Graphics Processing Units*– GPGPU). Esse tipo de computação permite a execução de tarefas paralelizáveis com desempenho superior em relação às CPUs da atualidade. Nestas arquiteturas, as ferramentas de programação dominantes atualmente são as proprietárias: CUDA e Stream SDK, que são fornecidos pelas mesmas empresas que fornecem o *hardware* (GPUs), respectivamente NVIDIA e AMD/ATI. Estima-se que estas sejam as soluções de programação com melhor desempenho no seu respectivo *hardware*. No entanto, os programas gerados, por serem específicos ao *hardware* de cada fabricante, perdem em portabilidade. Neste contexto, a ferramenta proposta neste trabalho constitui-se de um pacote com classes Java que usam o código dos SDKs proprietários como base, de forma a oferecer um recurso flexível e que aumente a produtividade do programador sem sacrificar o desempenho. TAJMAHAL, a ferramenta desenvolvida, é apresentada com dois exemplos de programas paralelos: ordenação bitônica e multiplicação de matrizes. Cada um desses exemplos possui os seus respectivos projetos TAJMAHAL, contendo cada um deles uma implementação em CUDA e outra em Stream SDK. Para testar a eficácia da ferramenta foi feita uma análise quantitativa e uma análise qualitativa.

**Palavras-chave:** GPGPU; stream processing; CUDA; Stream SDK; brook; brook+; JNI; java.

# ABSTRACT

Trabalho de Graduação
Curso de Ciência da Computação
Universidade Federal de Santa Maria

**A SOFTWARE DEVELOPMENT TOOL TO INTEGRATE JAVA APPLICATIONS WITH GENERAL-PURPOSE COMPUTING ON GRAPHICS PROCESSING UNITS**

Author: Bruno de Carvalho Christo
Advisor: Profª Andrea Schwertner Charão

The purpose of this work is to develop a software development tool for general-purpose computing on graphics processing units (GPGPU). This kind of computation allows the execution of parallelizable tasks with superior performance relative to current CPUs. In these architectures, the programming tools most widely used are the proprietary ones: CUDA and Stream SDK, which are offered by the same companies that make the hardware (GPUs): NVIDIA and AMD/ATI, respectively. It is estimated that those are the programming solutions with the best performance on it's respective hardware. However, the generated programs, by being tied to the manufacturer's hardware, lose portability. In this context, the tool proposed in this work consists of a package with Java classes that use the proprietary SDK's code as part of the solution, to make available a flexible, productivity increasing solution that doesn't compromise performance. TAJMAHAL, the developed tool, contains two examples of parallel programs: bitonic sort and matrix multiplication. Each of these examples has their own TAJMAHAL project containing a CUDA implementation and a Stream SDK implementation. To test the efficiency of this tool, quantitative and qualitative analyses were made.

**Keywords:** GPGPU,stream processing,CUDA,Stream SDK,brook,brook+,JNI,java.

# LISTA DE FIGURAS

# LISTA DE TABELAS

# LISTA DE ABREVIATURAS E SIGLAS

| | |
|---|---|
| AMD IL | Advanced Micro Devices Intermediate Language |
| BSD | Berkeley Software Distribution |
| CAL | Compute Abstraction Layer |
| CPU | Central Processing Unit |
| CTM | Close to Metal |
| CUDA | Compute Unified Device Architecture |
| GNU | GNU is Not Unix |
| GPGPU | General-purpose computing on graphics processing units |
| GPL | General Public License |
| GPU | Graphics Processing Unit |
| JNI | Java Native Interface |
| RAM | Random Access Memory |
| SDK | Software Development Kit |
| SIMD | Single-instruction, multiple-data |
| SIMT | Single-instruction, multiple-thread |

# SUMÁRIO

# 1 INTRODUÇÃO

Avanços recentes no desenvolvimento de placas gráficas proporcionaram a realização de computação de alto desempenho nestes dispositivos de baixo custo. A programação genérica em GPUs (*General Purpose Computation on Graphics Processing Units*– GPGPU) permite a execução concorrente de milhares de *threads* numa mesma placa gráfica, provendo um melhor aproveitamento de recursos. Linguagens de alto nível surgiram para o *hardware* gráfico, tornando esse poder computacional acessível (LUEBKE; HARRIS, 2004).

Em relação à arquitetura das GPUs, elas são processadores de fluxo (*stream processors*) altamente paralelos, otimizados para operações vetoriais. Esses processadores são capazes de computação de propósito geral, além das aplicações gráficas para as quais eles foram projetados. Pesquisadores descobriram que o uso de uma GPU pode acelerar alguns problemas em algumas ordens de magnitude sobre a CPU (LUEBKE; HARRIS, 2005).

Atualmente, existem duas soluções proprietárias para se utilizar programação genérica em GPUs: CUDA (NVIDIA Corporation, 2007), da NVIDIA, e Stream SDK (Advanced Micro Devices Inc., 2007a), da AMD/ATI, que é a evolução do CTM (Advanced Micro Devices Inc., 2007b). Antes destas, existiram algumas tentativas de encapsular as tecnologias que são licenciadas no modelo software livre, como libSh (RapidMind Inc., 2003) e BrookGPU (Stanford University, 2004), e outras comerciais como RapidMind (RapidMind Inc., 2006), que evoluiu da libSh. Apesar do número de soluções disponíveis atualmente, nenhuma delas é simultaneamente independente de fabricante, disponível sob uma licença de software livre, atualizada com freqüência, flexível e com um desempenho semelhante às soluções proprietárias.

O Khronos Group, reconhecido por homologar vários padrões, entre eles o OpenGL,

nos dias finais da conclusão deste trabalho de graduação, homologou o OpenCL, que é uma API que procura padronizar a programação GPGPU através de uma API (Khronos Group, 2008). Este padrão tem o apoio de várias empresas, como a AMD/ATI, Apple, Broadcom, Intel, Motorola, Nokia, NVIDIA entre outras (The Tech Report, 2008).

Neste contexto, o presente trabalho trata do desenvolvimento de uma ferramenta que propõe-se a auxiliar os programadores de aplicações Java a usufruir do alto desempenho proporcionado pela programação no modelo GPGPU.

A ferramenta de desenvolvimento, denominada TAJMAHAL, é constituída de vários elementos: um pacote Java que contém classes auxiliares, *Makefiles* que automatizam diversas tarefas, um *shell script* auxiliar, dois projetos TAJMAHAL fornecidos como exemplos completos de uso da ferramenta, e um projeto-base *Template* que deverá ser copiado e utilizado como base para se criar novos projetos. Estes elementos são as fundações da ferramenta, e o programador que a utilizar deverá criar projetos TAJMAHAL usando essas fundações.

Os projetos TAJMAHAL contêm um *frontend*, que é a aplicação Java do programador, uma *Makefile* com variáveis específicas ao projeto e pelo menos uma implementação do(s) algoritmo(s) que deve(rão) ser executado(s) na GPU em algum dos *backends*. Essa implementação nada mais é do que um projeto específico CUDA ou Stream SDK, que deve ser copiado para dentro do projeto TAJMAHAL. Essa cópia deverá sofrer algumas alterações para ser integrada com a aplicação Java. Isto será descrito em detalhes no capítulo 4.

TAJMAHAL possui uma organização modular para poder suportar *backends* futuros. Apesar de permitir uma interação de alto nível com aplicações Java, a ferramenta não retira a liberdade do programador de realizar ajustes finos nos projetos específicos da sua aplicação para otimizar o seu desempenho. Assim, o programador tem de um lado a maioria das vantagens da linguagem Java, e, do outro, a possibilidade de realizar ajustes de baixo nível, o que lhe permite obter o alto desempenho esperado com o uso do modelo GPGPU.

# 2 SOLUÇÕES PARA GPGPU

O uso de *hardware* gráfico para computação de propósito geral tem sido uma área ativamente pesquisada por muitos anos, tendo-se iniciado em máquinas como o Ikonas, no final dos anos 70, que era usado em simuladores de vôo da força aérea dos Estados Unidos. Nos anos 80, um grupo universitário, Pixel-Planes, foi criado para estudar arquiteturas gráficas de alto desempenho. Mas essa área não se restringiu a aplicações gráficas. No começo dos anos 90, por exemplo, *hardware* gráfico foi utilizado para se efetuar o planejamento de movimentos de robôs.

No final da mesma década outros pesquisadores utilizaram técnicas de *z-buffer* para a computação de diagramas de Vonoroi. Outras aplicações de interesse surgiram no mesmo período, como por exemplo, o uso de *hardware* gráfico para quebra de senhas UNIX e para a realização de computação de redes neurais artificiais. Desde então, avanços recentes no desenvolvimento de placas gráficas têm proporcionado a computação de alto desempenho nestes dispositivos de baixo custo.

Um destes avanços foi a criação de linguagens de programação específicas para esse paradigma. Algumas destas linguagens requerem um forte conhecimento de computação gráfica, como é o caso de Cg, HLSL, ASHLI, entre outras (HOUSTON, 2005). Já outras linguagens, como Brook e Sh são de nível mais alto, e surgiram com a intenção de tornar a programação GPGPU mais simples.

Tradicionalmente, a programação GPGPU consistia em adaptar problemas à programação gráfica. Com o surgimento das novas arquiteturas das GPUs da atualidade, como a Tesla G80, da NVIDIA, e a R600, da AMD/ATI, surgiu também uma nova forma de programação GPGPU, já que a programação genérica pode agora ser desvinculada da programação gráfica graças a essas novas arquiteturas e às camadas de abstração criadas por estes fabricantes.

Ambos os fabricantes fornecem SDKs para se realizar esta nova forma de programação em suas respectivas placas. A NVIDIA fornece CUDA, e a AMD/ATI, o Stream SDK. Porém, cada SDK é específico para as placas do seu fabricante, sendo o código gerado incompatível entre os mesmos. Além destes SDKs, existem softwares de terceiros que permitem que aplicações tradicionais *single-threaded* utilizem multiprocessadores (sejam CPUs ou GPUs) através da linguagem C++ e APIs proprietárias, como é o caso da RapidMind (RapidMind Inc., 2006) e a PeakStream, indisponível após ter sido adquirida pelo Google em 2007 (IDG Now!, 2007).

## 2.1 Arquitetura de uma GPU

Esta seção descreve a arquitetura Tesla, um conjunto de multiprocessadores SIMT com memória compartilhada *on-chip*, da NVIDIA como exemplo de implementação de uma GPU. A arquitetura Tesla foi construída em torno de um conjunto escalável de multiprocessadores *multithreaded*. Quando um programa CUDA no CPU hospedeiro invoca um *kernel* sobre uma grade, os blocos da grade são enumerados e distribuídos pelos multiprocessadores que possuam capacidade de execução disponível. As *threads* de um mesmo bloco executam concorrentemente em um multiprocessador. À medida que os blocos forem finalizando o seu processamento, novos blocos são enviados aos multiprocessadores que ficaram ociosos.

Um multiprocessador é composto de oito processadores de núcleo escalar, duas unidades especiais para transcendentais, uma unidade de instrução *multithreaded*, e memória *on-chip* compartilhada. O multiprocessador cria, gerencia e executa *threads* concorrentes em *hardware* sem sobrecarga de escalonamento. Uma barreira de sincronização, utilizada para sincronizar as *threads*, é implementada com uma única instrução. Essa sincronização rápida das *threads*, juntamente com um baixo custo de criação de *threads* e ausência de sobrecarga de escalonamento, contribui para um suporte eficiente de paralelismo de granularidade extremamente fina. Isso permite, por exemplo, a decomposição fina de um problema através da atribuição de uma *thread* a cada elemento do dado (como um *pixel* em uma imagem, um *voxel* em um volume, uma célula em computação baseada em grade).

Para gerenciar centenas de *threads* executando vários programas diferentes, o multiprocessador utiliza uma nova arquitetura chamada *Single-Instruction, Multiple-Thread*

*(SIMT)*. O multiprocessador mapeia cada *thread* a um processador de núcleo escalar, e cada *thread* escalar executa independentemente com o seu próprio endereço de instrução e estado dos registradores. A unidade de multiprocessamento SIMT cria, gerencia, escalona, e executa *threads* em grupos de 32 *threads* paralelas chamadas de dobras. As *threads* individuais que compõem uma dobra SIMT iniciam todas no mesmo endereço, mas elas são livres para desviar e executar independentemente.

Quando são fornecidos um ou mais blocos de *threads* para um multiprocessador executar, este os divide em dobras que são escalonadas pela unidade SIMT. A arquitetura SIMT é semelhante à organização dos vetores da SIMD, onde uma única instrução controla múltiplos elementos de processamento. Uma diferença chave é que uma organização de vetor SIMD expõe a largura SIMD ao software, enquanto que as instruções SIMT especificam o comportamento de desvio e a execução de uma única *thread*. Em contraste com o vetor de máquinas SIMD, SIMT permite que os programadores escrevam código paralelo a nível de *thread* para *threads* independentes e escalares, assim como código paralelo para *threads* coordenadas.

Como ilustrado pela figura 2.1, cada multiprocessador possui memória *on-chip* dos seguintes tipos:

- Um conjunto de registradores locais de 32-bits por processador,
- Espaços de memória locais e globais para leitura e escrita, sem cache,
- Uma cache paralela ou memória compartilhada que é compartilhada por todos os processadores de núcleos escalares,
- Uma cache de constantes somente para leitura que é compartilhada por todos os processadores de núcleos escalares, e que acelera leituras do espaço de memória destinado a constantes, que é uma região da memória do dispositivo somente para leitura,
- Uma cache de texturas somente para leitura que é compartilhada por todos os processadores escalares, e que acelera leituras do espaço de memória destinado a texturas, que é uma região da memória do dispositivo somente para leitura; cada multiprocessador acessa a cache de textura através de uma unidade de textura que implementa diversos modos de endereçamento e filtragem de dados.

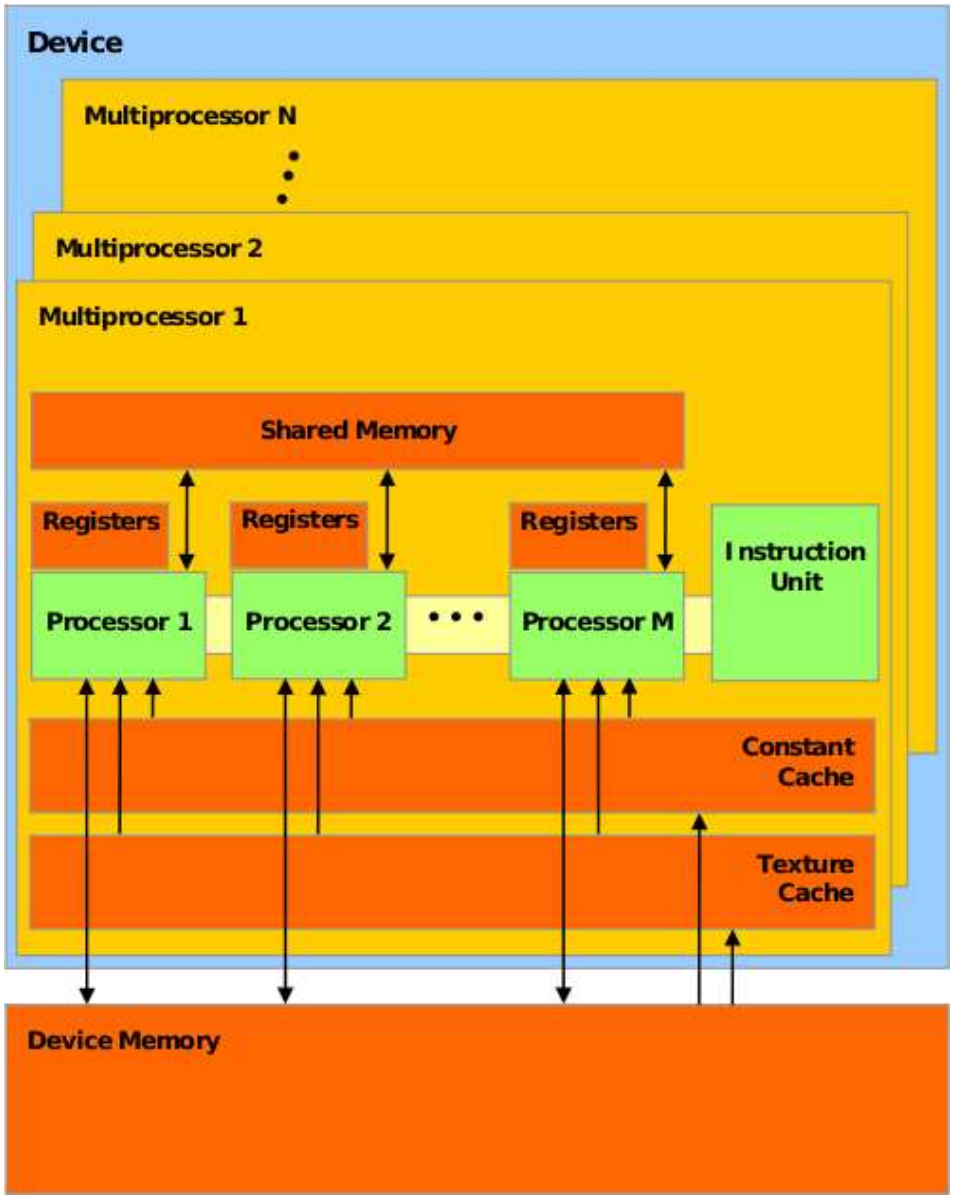


Figura 2.1: Arquitetura Tesla da NVIDIA (NVIDIA Corporation, 2008)

A quantidade de blocos que um multiprocessador pode processar de cada vez depende de quantos registradores por *thread* e de quanto de memória compartilhada por bloco são necessários para um *kernel*, já que os registradores e a memória compartilhada são divididos entre todas as *threads* do conjunto de blocos. Se não existirem registradores suficientes ou memória compartilhada disponível por multiprocessador para processar pelo menos um bloco, o *kernel* não poderá ser executado e gerará um erro. Cada multiprocessador pode executar no máximo oito blocos simultaneamente.

## 2.2 CUDA

CUDA (*Compute Unified Device Architecture*) é um compilador C e um conjunto de ferramentas de desenvolvimento que permitem que programadores utilizem a linguagem C para criar algoritmos que executem em GPUs (NVIDIA Corporation, 2007). CUDA foi desenvolvida pela NVIDIA e para utilizar esta arquitetura é necessário possuir uma GPU da empresa e os últimos *drivers*, que atualmente possuem todos os componentes CUDA. CUDA funciona com todas as GPUs da NVIDIA da série G8X em diante, incluindo GeForce, Quadro e Tesla, esta última projetada especificamente para o mercado de computação de alto desempenho.

A NVIDIA garante que os programas desenvolvidos para as placas gráficas GeForce 8 também funcionarão sem alterações em todas as placas gráficas futuras da NVIDIA, devido à compatibilidade binária da arquitetura. CUDA fornece aos desenvolvedores acesso ao conjunto de instruções nativas e à memória dos elementos computacionais massivamente paralelos presentes nas GPUs suportadas por CUDA.

## 2.3 AMD Stream

O AMD Stream SDK (Advanced Micro Devices Inc., 2007a) da AMD/ATI é o equivalente à CUDA da NVIDIA, mas para as GPUs da AMD, Radeon (a partir da R580) e FireGL (a partir da R600).

Fazem parte do SDK a linguagem Brook+, que é a versão modificada pela AMD do compilador open source Brook, e CAL, também da AMD, que é a interface multiplataforma para a GPU (Advanced Micro Devices Inc., 2008).

Brook+ é uma linguagem de alto nível que é mais fácil de usar do que a CAL, porém não fornece toda a sua funcionalidade.

CAL é uma interface multiplataforma, permite que os desenvolvedores de software interajam com os *stream processors* no nível mais baixo possível para otimizar o seu desempenho, com a segurança de que a compatibilidade futura será mantida.

# 3 PROGRAMAÇÃO EM GPGPU

O capítulo anterior utilizou CUDA como exemplo para explicar como é implementada uma GPU de última geração. Este capítulo utiliza o AMD Stream SDK como exemplo para demonstrar a programação GPGPU. Para facilitar a compreensão do componente Brook+, as extensões Brook também são apresentadas. Finalmente, este capítulo introduz uma API proprietária e revela alguns motivos que justificam porque a sua utilização pode nem sempre ser uma boa escolha.

## 3.1 Brook

Brook é um conjunto de extensões para a linguagem C para programação com streams originalmente desenvolvida pela Universidade de Stanford nos Estados Unidos (Stanford University, 2004).

O objetivo de Brook é encapsular toda a parte do gerenciamento da API 3D e expor a GPU como um co-processador para cálculos paralelos. Para este fim, Brook é composto por um compilador, que obtém um arquivo de extensão *.br* contendo código em C++ com extensões e gera código padrão C++ que será ligado a uma biblioteca de run-time. Esta biblioteca possui vários *backends*, como OpenGL ARB, DirectX, OpenGL NV3x, e x86 (figura 3.1).

Um aspecto importante da linguagem Brook foi ter popularizado o conceito de GPGPU. Talvez isto se deva ao fato de ter simplificado o acesso aos recursos da GPU, permitindo que mais pessoas tenham começado a aprender esse novo modelo de programação. Apesar disso, Brook ainda tem um longo caminho a percorrer antes de poder tornar as GPUs unidades de cálculo confiáveis (Tom's Hardware, 2008).

Um dos problemas encontrados nesta área surgiu a partir das diferentes camadas de abstração, em particular na carga de trabalho excessiva gerada pela API 3D. Porém, o

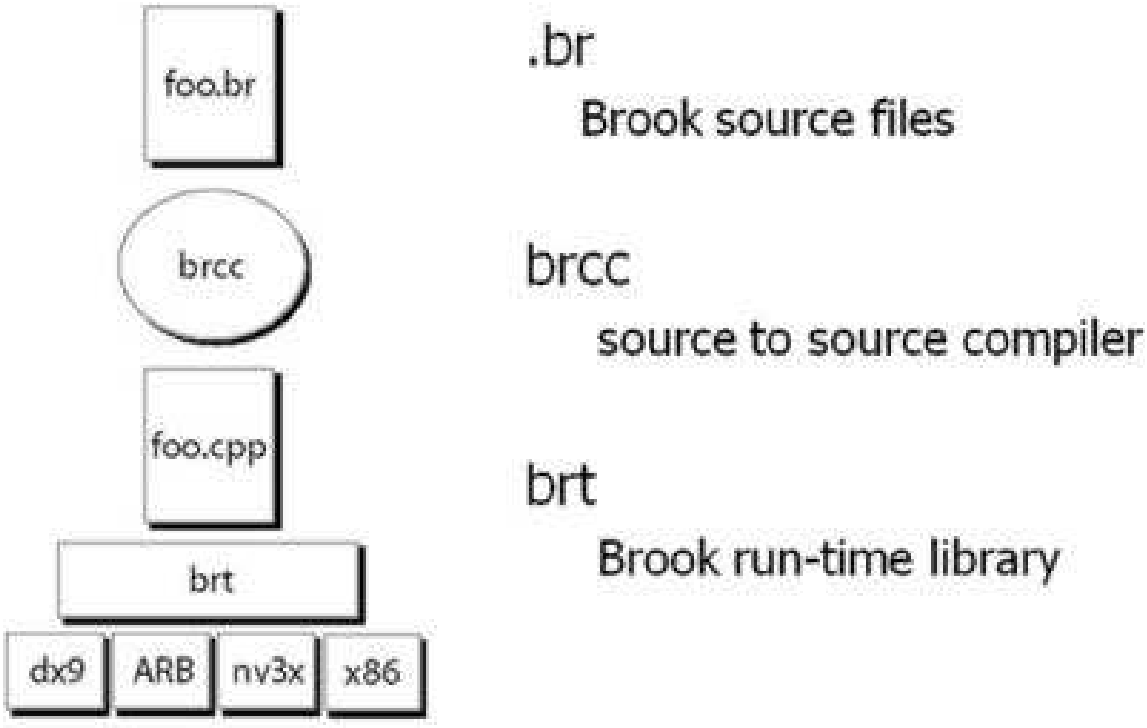


Figura 3.1: Hierarquia Brook (Stanford University, 2004)

maior dos problemas, que estava fora do controle dos programadores do Brook, seria a compatibilidade. Os fabricantes das GPUs otimizam e atualizam os seus *drivers* com frequência, em especial devido à forte competição entre os mesmos. Essas otimizações podem quebrar a compatibilidade em qualquer lançamento novo de *drivers*, e assim dificultam a utilização dessa API em código de alta qualidade. Devido a isso, Brook por muito tempo foi apenas de interesse de pesquisadores e programadores curiosos.

## 3.2 Componentes do AMD Stream SDK

### 3.2.1 Brook+

Brook+ é uma implementação da AMD para a especificação BrookGPU com algumas melhorias sobre a Compute Abstraction Layer (CAL). O seu código é aberto e com licença BSD (Advanced Micro Devices Inc., 2007c).

O compilador Brook+ converte arquivos fonte Brook+ em código C++. Os *kernels*, escritos em C, são compilados para o código AMD IL para a GPU ou código C para a CPU. No run-time Brook+, o código IL é executado na GPU. O *backend* é escrito em CAL, que é uma camada de abstração de nível mais baixo. Esta hierarquia é ilustrada na figura 3.2.

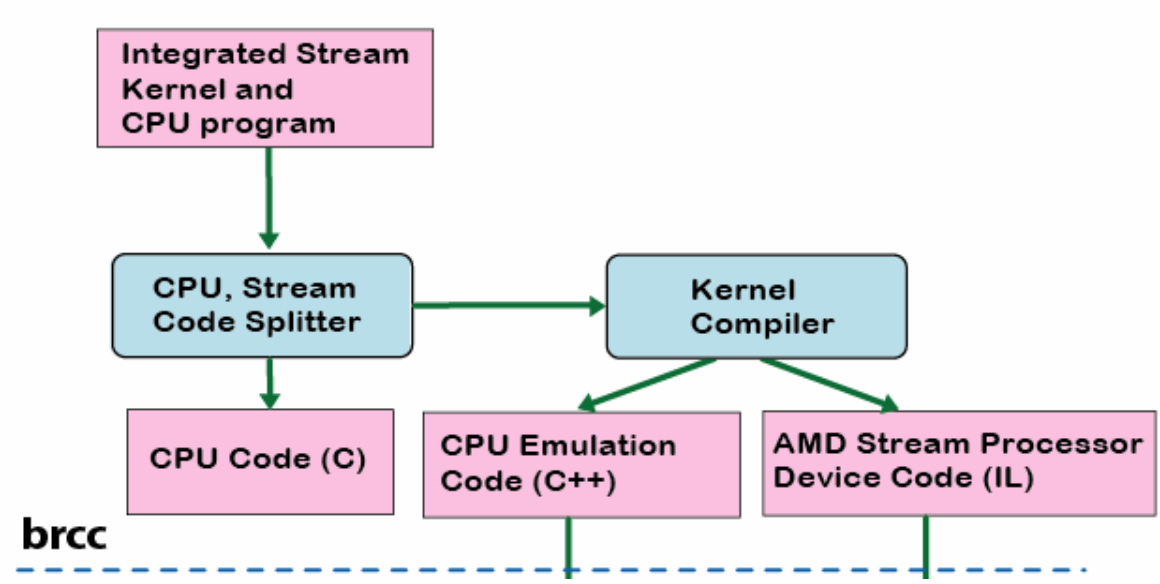


Figura 3.2: Hierarquia Brook+ (Advanced Micro Devices Inc., 2007c)

### 3.2.2 CAL

A Compute Abstraction Layer (CAL) da AMD fornece uma interface que, segundo a empresa, é fácil de utilizar e projetada com compatibilidade futura em mente (Advanced Micro Devices Inc., 2008).

Essa interface fornece acesso aos núcleos dos processadores das GPUs, gerenciamento de dispositivos e de recursos, geração de código, e o carregamento e execução dos *kernels*. CAL providencia uma biblioteca de drivers de dispositivos que permite às aplicações interagir com os núcleos dos processadores ao nível mais baixo possível para um desempenho ótimo. Também garante compatibilidade futura.

## 3.3 RapidMind

RapidMind é uma plataforma de desenvolvimento e *runtime* que permite que aplicativos gerenciáveis e *single threaded* tenham acesso completo a processadores multicore. Com RapidMind, desenvolvedores podem continuar escrevendo código em C++ padrão e utilizar as suas habilidades existentes, ferramentas e processos.

A plataforma RapidMind paraleliza a aplicação sobre múltiplos *cores* e gerencia a sua execução. Essa paralelização pode ser feita sobre diversos *backends*, como GPUs e processadores Cell.

Além de ter simplificado a programação GPGPU, é uma plataforma versátil, já que é baseada em OpenGL e suporta diversos *backends*. Porém, possui alguns pontos negativos:

- uma aplicação paralelizada desta forma jamais terá o desempenho esperado de uma aplicação nativa, já que possui a sobrecarga da API + OpenGL;
- o caráter genérico desta plataforma retira controle do programador sobre os detalhes de cada arquitetura;
- o programador deve aprender uma nova API e converter o seu código existente para esta API;
- o programador fica dependente do sucesso da empresa, já que a API é proprietária;
- ter toda a aplicação escrita na linguagem C++ pode não ser a melhor opção.

# 4 DESENVOLVIMENTO DA FERRAMENTA

Este capítulo irá apresentar o projeto da ferramenta, a sua implementação, e uma análise prática do processo de instalação, configuração e utilização.

## 4.1 Projeto

A ferramenta TAJMAHAL provê recursos para integração entre um *software* implementado para executar em GPU e o *software* em Java do programador. Estes recursos formam uma camada de *software* entre o código específico para cada plataforma de GPGPU e o código Java da aplicação, conforme ilustra a figura 4.1.


Figura 4.1: TAJMAHAL, funcionando como uma camada intermediária, permite que aplicações Java realizem computações em GPUs.

### 4.1.1 Integração de aplicações Java com GPGPU

Um dos objetivos da ferramenta TAJMAHAL é disponibilizar as vantagens da GPGPU a aplicações Java, através da integração de ambas. Esta seção explica alguns motivos que justificam a integração com esta linguagem, em oposição ao desenvolvimento de aplicações totalmente em C++ ou Brook+ (que gera código C++).

As diferenças entre as linguagens C++ e Java podem ser identificadas nos seus projetos, já que ambas têm objetivos diferentes. A linguagem C++ foi projetada principalmente para programação de sistemas, estendendo a linguagem C. Para esta linguagem de programação procedimental projetada para execução eficiente, C++ adicionou suporte à programação orientada a objetos, tratamento de exceções, e programação genérica. Também adicionou uma biblioteca padrão que inclui objetos contêineres e algoritmos (STROUSTRUP, 2006).

O objetivo inicial da linguagem Java foi o suporte de computação em rede. Ela depende de uma máquina virtual para ser segura e altamente portável. Acompanha uma biblioteca extensa projetada para fornecer uma abstração completa da plataforma abaixo. Java é uma linguagem orientada a objetos que usa uma sintaxe semelhante a C, mas não é compatível com esta. A linguagem Java foi projetada do zero, com o objetivo de ser fácil de utilizar e acessível a mais programadores (LEMAY; CADENHEAD, 2001).

Apesar das semelhanças com C++, os aspectos mais propensos a erro desta linguagem foram eliminados de Java. Por exemplo, não existe aritmética de ponteiros em Java, apesar desta linguagem utilizar ponteiros implicitamente. Esses aspectos são fáceis de serem incorretamente utilizados em um programa e muito difíceis de serem resolvidos. Outra vantagem é que o gerenciamento de memória é automático, tornando as fugas de memória um problema muito menos freqüente.

Java é portanto uma linguagem menos propensa a erros, e os erros podem ser consertados com maior rapidez. Além disso, a programação em Java proporciona uma produtividade superior a C++ (PHIPPS, 1999). Java tem sido frequentemente apelidada de linguagem lenta desde as suas origens, porém hoje em dia ela pode chegar a ser mais rápida que C++ em alguns casos (J.P.LEWIS; NEUMANN, 2003).

### 4.1.2 Java Native Interface (JNI)

A JNI permite a integração de código escrito em Java com código escrito em outras linguagens como C e C++. Com isto, programadores podem ter todas as vantagens de Java sem ter que abandonar o seu investimento em código legado (LIANG, 2002), já que JNI permite que, com algumas alterações no código, uma aplicação existente escrita em outra linguagem de programação se torne acessível a aplicações Java.

Neste trabalho, os projetos TAJMAHAL utilizam JNI com o propósito de criar a ligação entre bibliotecas nativas geradas a partir dos projetos específicos de cada *backend* e uma classe Java especial (*frontend*), que utiliza um ou mais métodos nativos presentes nessas bibliotecas. Cada projeto TAJMAHAL deve possuir pelo menos um *frontend* e um *backend*.

### 4.1.3 GNU *Makefiles* e os projetos específicos

GNU *Make* (Free Software Foundation, 2006) é uma ferramenta que controla a geração de executáveis e outros arquivos não-fonte a partir de arquivos do código fonte do programa. O programa *Make* descobre como construir o programa através de um arquivo chamado *Makefile*, o qual lista cada um dos arquivos não-fonte e como os gerar a partir de outros arquivos. Quando um projeto TAJMAHAL é criado, este deve conter um *Makefile*, de modo que seja possível usar o programa *Make* para construir e utilizar o programa.

Em relação ao projeto em questão, um *Makefile* personalizado deve ser utilizado juntamente com o projeto TAJMAHAL para definir alguns parâmetros necessários para gerar bibliotecas que contêm o código nativo das GPUs, que serão carregadas e utilizadas nesse projeto. Essas bibliotecas são geradas a partir de implementações nativas em C/C++ e Brook+ (indiretamente), através dos compiladores inclusos nos SDKs CUDA e Stream, modificadas para funcionar com JNI.

## 4.2 Implementação

A ferramenta TAJMAHAL foi implementada como um pacote Java com *Makefiles*, classes auxiliares e projetos TAJMAHAL, e pode ser definida como uma ferramenta de desenvolvimento de aplicações Java que necessitem de acessar os recursos disponíveis em uma ou mais GPUs para obter um ganho de desempenho.

### 4.2.1 Hierarquia da ferramenta TAJMAHAL

A ferramenta de desenvolvimento TAJMAHAL é composta de diversos elementos, o que causa a necessidade de se organizar destes elementos de forma hierárquica. A figura 4.2 demonstra esta hierarquia, que é descrita nos parágrafos seguintes.

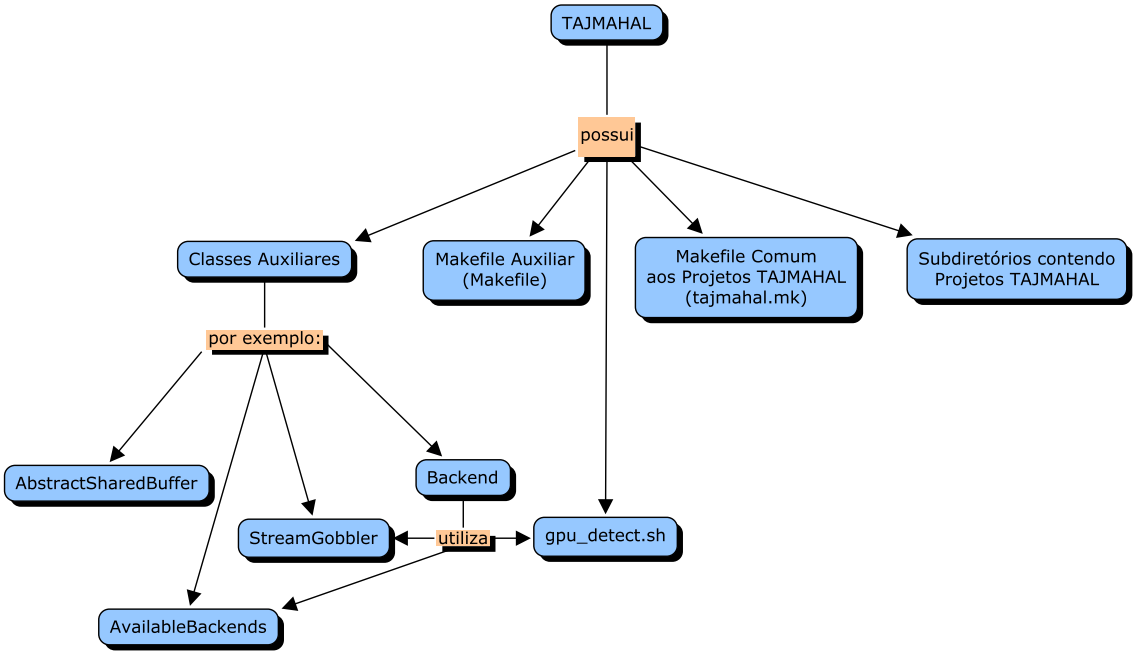


Figura 4.2: Hierarquia da ferramenta TAJMAHAL.

A principal função das classes auxiliares é permitir que o programador crie as suas próprias classes sem necessitar reimplementar funcionalidades comuns aos projetos TAJMAHAL. Por exemplo, a classe auxiliar abstrata AbstractSharedBuffer (no apêndice B.1) implementa a alocação dos *buffers* compartilhados entre a aplicação Java e o seu *backend* (por isso o seu nome - SharedBuffer, *buffer* compartilhado). Já a forma como os dados são carregados e descarregados da memória (a partir de arquivos) são métodos abstratos; um programador pode implementar a carga/descarga segundo as necessidades do seu projeto TAJMAHAL estendendo a classe SharedBuffer (exemplo de implementação no apêndice C.1). A classe Backend (nos apêndices B.3 e B.4), ao contrário do que possa parecer, não é a implementação de um *backend*. Ela apenas implementa a auto-detecção da placa gráfica em uso no sistema para a escolha correta do *backend* correspondente em tempo de execução.

O Makefile Auxiliar (exemplo no apêndice B.6), localizado na raíz da ferramenta TAJMAHAL, é utilizado para facilitar o processo de aplicação de modificações necessárias nos *backends*. Isto é feito através do uso de *patches*. O apêndice A.1 contém o *patch* para

CUDA e o apêndice A.2 contém o *patch* para o AMD Stream SDK. A seção 4.4 possui mais informações a respeito destes *patches*.

O arquivo *tajmahal.mk* (exemplo nos apêndices B.7 e B.8) contém variáveis comuns a todos os projetos TAJMAHAL; deve ser incluído em cada *Makefile* específico (exemplo no apêndice C.10.

Cada projeto TAJMAHAL deverá possuir o seu subdiretório, idêntico ao nome do projeto. Este subdiretório deve conter classes Java pertencentes ao pacote TAJMAHAL.nomeDoProjeto (onde nomeDoProjeto é o nome do projeto TAJMAHAL). Uma dessas classes será o *frontend* que implementa um ou mais métodos nativos (exemplo nos apêndices C.3 e C.4). O projeto também deverá possuir um *Makefile* (como a do apêndice C.10) e um ou mais projetos específicos. Esses projetos específicos nada mais são que projetos CUDA e Stream SDK, modificados para funcionar com JNI (exemplo de modificações necessárias nos apêndices C.7, C.8, C.9) e gerar bibliotecas nativas para uso posterior. Um exemplo de bibiloteca nativa gerada a partir do projeto *BitonicSort* é a 'libjava-BitonicSort-stream.so'. Os projetos específicos localizam-se dentro dos projetos TAJMAHAL no diretório *backends*.

### 4.2.2 Compilação de um Projeto TAJMAHAL

A compilação de um projeto TAJMAHAL é de uma certa complexidade, e seriam trabalhosas para um programador que tivesse que as executar manualmente. Devido a isto, várias etapas foram automatizadas com a ajuda de *Makefiles*. A figura 4.3 demonstra as etapas envolvidas no processo de compilação de um projeto TAJMAHAL e os comandos usados para as realizar.

O *Makefile* de um projeto TAJMAHAL tem como objetivo a geração de bibliotecas compartilhadas nativas a partir de projetos específicos criados para os *backends* atuais: Stream SDK e NVIDIA CUDA. Se um projeto TAJMAHAL possuir implementações em ambos os projetos específicos, estes deverão disponibilizar a mesma funcionalidade a uma aplicação Java, utilizando JNI como intermediária. A(s) biblioteca(s) gerada(s) será(ão) utilizada(s) por uma classe Java especial (*frontend*) que implementa os métodos nativos.

O *Makefile* localizado no diretório do seu respectivo projeto contém algumas variáveis específicas a este, como o nome do projeto, que deve ser igual ao subdiretório contido na raiz da ferramenta TAJMAHAL, a classe que possui os métodos nativos e faz uso da JNI,

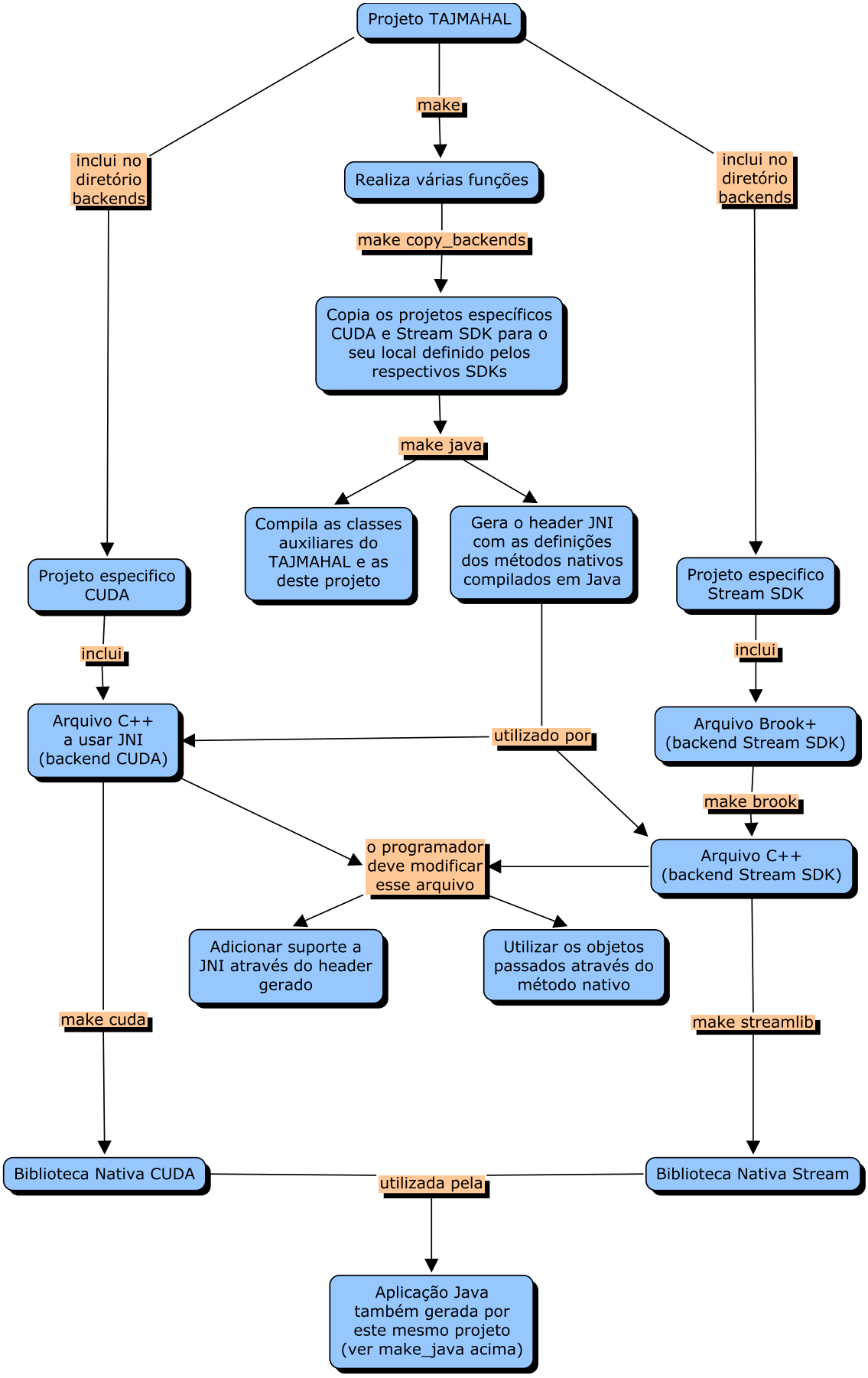


Figura 4.3: Passos necessários para a compilação de um projeto TAJMAHAL que faça uso dos 2 *backends* disponíveis atualmente.

os nomes dos projetos específicos dos *backends*, que novamente devem ser iguais aos seus subdiretórios, e um arquivo intermediário, que é um caso especial: é uma classe C++ gerada a partir de código Brook+. Esse caso é explicado mais adiante. Um exemplo de *Makefile* de projeto TAJMAHAL pode ser encontrado no apêndice C.10.

Após a compilação das classes Java do projeto, um arquivo *header* JNI é gerado a partir de uma classe especial (*frontend*) que declara métodos nativos. Um exemplo de *header* JNI encontra-se no anexo C.6. Este arquivo deverá ser incluído e utilizado pelos projetos específicos, no momento em que estes forem modificados para funcionar com JNI. O anexo C.7 demonstra um arquivo C++ antes do uso desse *header* e os anexos C.8 e C.9 demonstram como incluir e utilizar esse *header*. É neste ponto que a comunicação entre a aplicação Java e o *backend* é estabelecida, e dados podem transitar entre *frontend* e *backend*.

A compilação de um projeto específico CUDA é realizada em um único passo, enquanto que a compilação de um projeto específico Stream SDK se divide em dois. Isto se deve ao fato do processo de compilação de uma implementação normal, como nos exemplos disponibilizados no Stream SDK, ser realizado na sequência ilustrada na figura 4.4.

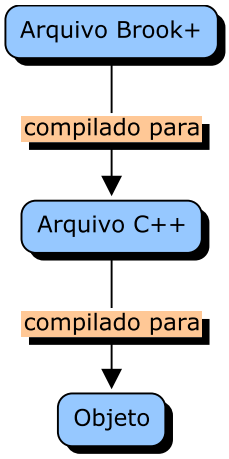


Figura 4.4: A compilação do código Brook+ é realizada em dois passos, e as modificações para realizar a integração do código C++ com Java devem ser realizadas no arquivo intermediário C++.

Um problema encontrado foi o fato da linguagem Brook+ não possuir suporte a JNI. Porém, o processo de compilação dos projetos inclusos no ATI Stream SDK, conforme apresentado na seção 3.1, consiste na transformação da linguagem Brook+ em arquivos intermediários C++, que devem ser modificados para incluir suporte a JNI.

O programador que utilizar a ferramenta desenvolvida com o ATI Stream SDK deverá primeiramente criar e testar a sua implementação em Brook+ normalmente. Quando a funcionalidade desejada for obtida, e faltar apenas a etapa de integração com JNI para disponibilizar a implementação nativa ao seu programa Java, o código em Brook+ deverá ser convertido em C++ no diretório do projeto TAJMAHAL correspondente através do comando: 'make brook'. Este comando gerará um arquivo C++ no mesmo diretório com o mesmo nome do arquivo Brook+ convertido.

As alterações necessárias deverão ser realizadas neste arquivo para que funcione com JNI. Quando estiver concluído, basta digitar 'make streamlib', e a biblioteca nativa será gerada e copiada para o diretório do projeto TAJMAHAL para ser utilizada pela aplicação Java. Para executar o programa, basta utilizar o *script* de conveniência *run.sh* localizado na pasta do projeto TAJMAHAL.

Nesta primeira versão da ferramenta, foram incluídos dois projetos TAJMAHAL como exemplos: Ordenação Bitônica, e Multiplicação de Matrizes. Cada um destes projetos TAJMAHAL possui dois projetos específicos: um no *backend* CUDA, e o outro no *backend* Stream SDK. Esses SDKs fornecem alguns projetos específicos como exemplos.

Ambos os SDKs possuem exemplos de ordenação bitônica e multiplicação de matrizes. Os exemplos *bitonic_sort* e *bitonic* foram adaptados para funcionar como projetos específicos do projeto TAJMAHAL *BitonicSort*. Analogamente, os exemplos *matrixMul* e *optimized_matmult* foram adaptados para serem fornecidos com a ferramenta como projetos específicos do projeto *MatrixMul*.

Os projetos específicos a cada SDK devem ser compilados em locais de projeto definidos pelos seus respectivos SDKs. Estes locais são definidos no *Makefile* de cada projeto TAJMAHAL. Por esse motivo, é necessária a cópia dos projetos específicos localizados na pasta *backends* para os locais de projeto definidos pelos SDKs, e o *Makefile* faz isso automaticamente antes de cada compilação. Isso também é vantajoso para o programador, já que este pode editar o seu projeto TAJMAHAL e os seus projetos específicos sem precisar se deslocar aos locais de projeto definidos por CUDA e Stream SDK.

## 4.3 Aplicação

Como foi dito anteriormente, a ferramenta TAJMAHAL inclui dois projetos TAJMAHAL como exemplos de uso: Ordenação Bitônica (*BitonicSort*) e Multiplicação de Ma-

trizes (*MatrixMul*).

### 4.3.1 Ordenação Bitônica

A ordenação bitônica tem como base o conceito de seqüência bitônica, que é uma lista com propriedades específicas que serão utilizadas no algoritmo de ordenação (CÁCERES; MONGELLI, 2002).

Nem todo algoritmo de ordenação possui algoritmos ou implementações que utilizem eficientemente *hardware* altamente paralelo como uma GPU. A ordenação bitônica foi escolhida justamente por atender a esses requisitos: ela possui não só a possibilidade de ser implementada utilizando-se um algoritmo paralelo, mas também implementações nos SDKs de ambos os fabricantes. Assim, os códigos existentes terão que sofrer poucas alterações para serem aproveitados como *backends* do projeto de exemplo *BitonicSort* da ferramenta TAJMAHAL.

### 4.3.2 Multiplicação de Matrizes

A multiplicação de matrizes consiste na obtenção de uma matriz C, que é o resultado de uma multiplicação de duas outras matrizes, A e B.

Por ser um problema clássico da programação paralela, e possuir implementações nativas tanto em CUDA como Stream SDK, fornecidas como exemplos, este problema foi escolhido por possuir as mesmas vantagens da ordenação bitônica. Os *backends* são aproveitados com poucas alterações para o projeto de exemplo *MatrixMul* fornecido juntamente com TAJMAHAL.

## 4.4 Utilização da Ferramenta

Esta etapa descreve um exemplo completo de instalação e utilização da ferramenta TAJMAHAL. Serão descritos os passos utilizados na instalação da ferramenta, bem como os passos utilizados para a criação de um projeto TAJMAHAL, o *BitonicSort*.

### 4.4.1 Instalação da ferramenta TAJMAHAL

Supõe-se que os SDKs dos *backends* a serem utilizados pela aplicação Java estão devidamente instalados. O usuário deverá definir os parâmetros de configuração referenciados na documentação dos SDKs. A variável JAVA_HOME também deverá estar configurada corretamente.

O arquivo compactado TAJMAHAL.tar.bz2 deverá ser copiado para um diretório e extraído. Será criado um diretório TAJMAHAL, chamado de raiz da ferramenta TAJMAHAL. Os projetos TAJMAHAL localizam-se em diretórios com o nome do projeto dentro dessa raiz.

A raiz contém vários diretórios e arquivos, entre eles o arquivo *tajmahal.mk* (exemplo no apêndice B.7). Esse arquivo possui variáveis comuns a todos os projetos TAJMAHAL, locais de instalação dos *backends*, geração de arquivos intermediários e as regras de compilação. Deverá ser o primeiro arquivo a ser personalizado após a extração da ferramenta. As variáveis CUDA_SDK_PATH e STREAM_SDK_PATH deverão conter os caminhos de instalação dos respectivos SDKs.

Um projeto TAJMAHAL faz uso de Java Native Interface. Esta interface requer bibliotecas compartilhadas nativas, porém nem CUDA nem Stream SDK possuem esta opção totalmente implementada, pelo menos não para os exemplos fornecidos. Isto requer alguns passos adicionais para que os *backends* passem a suportar a geração dessas bibliotecas.

Os SDKs devem ser informados sobre o caminho de instalação da JNI. Estas funcionalidades foram adicionadas ao SDK através de *patches*, que encontram-se em 'TAJMAHAL/sdk_patches'.

Caso os SDKs não se encontrem instalados nos diretórios padrão, as variáveis STREAM_ROOT e CUDA_ROOT presentes no arquivo *Makefile* da raiz deverão ser ajustadas. Depois destes ajustes, o *patch* para o Stream SDK, main.mk (no apêndice A.1), pode ser aplicado digitando-se 'make patch_stream' no diretório raiz da ferramenta. Analogamente, o *patch* para CUDA pode ser aplicado digitando-se 'make patch_cuda' no diretório raiz da ferramenta TAJMAHAL. O comando 'make patches' realiza as duas operações de uma só vez. Caso algum destes *patches* não seja aplicado, bibliotecas compartilhadas dos respectivos *backends* simplesmente não serão geradas. Após a correta realização deste passo, a geração automática de bibliotecas compartilhadas é possível.

### 4.4.2 Criação de um Projeto TAJMAHAL

Por conveniência, juntamente com a ferramenta é fornecido um projeto TAJMAHAL chamado *Template*. Esse projeto não realiza operação alguma, mas tem como objetivo ser uma referência para a criação de novos projetos TAJMAHAL. Esses projetos podem

ser criados através de cópias do diretório *Template*, na raíz da ferramenta TAJMAHAL. Dentro de uma cópia, os arquivos *Makefile* e *run.sh* deverão ser personalizados, bem como as classes incluídas *Main.java*, uma classe com alguns *imports* e o método inicial *main*, *GPUSample.java*, um exemplo de classe que realiza chamadas a métodos nativos, e *SharedBuffer.java*, que estende a classe abstrata auxiliar *AbstractSharedBuffer.java*. O projeto *Template* também possui uma pasta *backends* vazia onde o programador deverá colocar os seus projetos específicos. Antes da criação do projeto, algumas suposições são feitas:

- O programador deseja criar uma classe Java chamada *GPUSorter*, que ordena uma matriz de elementos do tipo ponto flutuante na GPU que estiver disponível no sistema do usuário, independentemente do fabricante,
- O programador criou projetos específicos CUDA e Stream SDK com implementações para esse algoritmo, que foram testadas e funcionam corretamente. Essas implementações estão nos diretórios ~/NVIDIA_CUDA_SDK/projects/bitonic e /usr/local/amdbrook/samples/apps/bitonic_sort/, respectivamente,
- O programador está utilizando uma versão atual de Linux,
- Tanto CUDA quanto Stream SDK foram instalados nos diretórios padrão,
- A raiz da ferramenta TAJMAHAL é ~/TAJMAHAL.

Os passos iniciais para a criação do BitonicSort são:

- cd ~/TAJMAHAL
- cp -a *Template BitonicSort*
- cd BitonicSort
- Editar a *Makefile* desse diretório. Deverá ficar igual à do apêndice C.10.
- Mover o arquivo *GPUSample.java* para *GPUSorter* e editar esse arquivo. Deverá ficar igual ao código dos apêndices C.3 e C.4.
- Adicionar métodos de entrada e saída que tratem algum formato específico de dados no arquivo *SharedBuffer.java*.

- Criar uma classe *Matrix*, que utilize a classe *SharedBuffer*.
- Editar o *shell script run.sh* e trocar o nome da classe *Template* para *BitonicSort*, como no apêndice C.11.
- Editar o arquivo *Main.java* e criar um teste para a classe *GPUSorter*. Aquele deverá ser semelhante ao código do apêndice C.5.

Os projetos específicos deverão ser copiados para dentro do projeto TAJMAHAL:

- cp -a ~/NVIDIA_CUDA_SDK/projects/bitonic *BitonicSort*/*backends*/cuda/TAJMAHAL_bitonic
- cp -a /usr/local/amdbrook/samples/apps/bitonic_sort *BitonicSort*/*backends*/stream/TAJMAHAL_bitonic_sort

O programador agora deve digitar 'make java' para compilar a sua aplicação Java e gerar o *header* JNI. Este deverá ficar semelhante ao exemplo do anexo C.6.

Devido à aplicação dos *patches* explicada na seção 4.4.1, é possível compilar bibliotecas compartilhadas nos projetos fornecidos como exemplos CUDA. Para isso, basta alterar a *Makefile* correspondente do projeto, trocando-se no *Makefile* localizado em 'TAJMAHAL/BitonicSort/backends/cuda/bitonic' a variável 'EXECUTABLE := bitonic' para 'SHARED_LIB := libjava-bitonic_sort-cuda.so'.

Assim como no caso de CUDA, o projeto específico Stream SDK também deve sofrer algumas mudanças: é necessário alterar o *Makefile* desse projeto, no diretório 'TAJMAHAL/BitonicSort/backends/stream/bitonic_sort'. Mais especificamente, o nome da variável 'GENERATE_EXECUTABLE' deve ser alterado para 'GENERATE_SHARED_LIBRARY'. A biblioteca será gerada em '/usr/local/amdbrook/samples/lib/lnx_x86_64', se o sistema operacional for de 64-bits. Essa biblioteca será posteriormente copiada automaticamente e utilizada pelo *frontend GPUSorter.java*. O nome da biblioteca será 'libbitonic_sort.so'. Para isso, a variável 'RELEASE := 1' deve ser incluída na *Makefile*. Caso não esteja, o nome da biblioteca gerada será 'libbitonic_sort_d.so', que é uma versão para depuração. Nos *Makefiles* dos projetos específicos Stream SDK é recomendável adicionar no final da atribuição da variável 'CFLAGS', a variável '$FPIC', para o *Makefile* da ferramenta TAJMAHAL

autodetectar se o sistema é 32 ou 64-bits. A definição deverá ser então: CFLAGS += $(C_INCLUDE_FLAG)"../../common".

Os projetos específicos deverão ser alterados para fazer uso do *header* gerado, neste caso os métodos contidos em TAJMAHAL_BitonicSort_GPUSorter.h. Um exemplo de alterações necessárias pode ser visualizado nos apêndices C.7, antes das alterações, e C.8, C.9 após as alterações.

Com os projetos específicos modificados, devem agora ser copiados para a pasta do SDK correspondente com o comando 'make copy_backends'.

A compilação para o *backend* CUDA no projeto TAJMAHAL pode agora ser iniciada com o comando 'make cuda'. Após a compilação, a biblioteca gerada estará em '~/NVIDIA_CUDA_SDK/lib/libjava-bitonic_sort-cuda.so'. Neste mesmo passo, a biblioteca será copiada para o diretório do projeto TAJMAHAL para ser utilizada pela aplicação Java.

Para testar o programa, basta utilizar o *script* de conveniência *run.sh* localizado no diretório do projeto TAJMAHAL. Se o programador possuir uma GPU NVIDIA na sua máquina, esta deverá ser autodetectada e o programa deverá realizar a ordenação na GPU. Caso não possua, basta alterar a classe *Main.java* e substituir a chamada ao método *detect()* por *set(AvailableBackends.CUDA)* para forçar a utilização deste *backend*. Mesmo o programador não possuindo a GPU, poderá executar em modo de emulação - consulte a documentação do SDK para mais detalhes.

O *frontend GPUSorter* agora funciona com o *backend* CUDA. O programador deseja que o seu programa funcione também com o Stream SDK, então deverá digitar 'make brook'. Receberá uma mensagem sugerindo que edite um arquivo antes de prosseguir com a segunda parte da compilação. O motivo desta divisão é explicado na seção 4.2.2. Após a edição desse arquivo, deve digitar 'make streamlib', e a biblioteca *stream* será gerada e copiada para a raiz do projeto TAJMAHAL. O *frontend* pode também agora realizar a ordenação em GPUs da AMD. Se o programador possuir uma GPU AMD/ATI na sua máquina, esta deverá ser autodetectada e o programa deverá realizar a ordenação nessa GPU. Caso não possua, basta alterar a classe *Main.java* e substituir a chamada ao método *detect()* por *set(AvailableBackends.STREAM)* para forçar a utilização deste *backend*. Mesmo o programador não possuindo a GPU, poderá executar em modo de emulação - consulte a documentação do SDK para mais detalhes.

# 5 AVALIAÇÃO

Todo projeto de *software* passa por uma fase na qual avalia-se se o que foi implementado está de acordo com os requisitos levantados. Este projeto foi analisado quantitativa e qualitativamente, e os resultados dessas análises são apresentados abaixo.

## 5.1 Avaliação Quantitativa

De acordo com os requisitos levantados e seguindo o projeto previamente definido, foi realizada a implementação da ferramenta TAJMAHAL.

Conforme descrito no capítulo 4, a ferramenta desenvolvida deve permitir que o programador consiga integrar o código nativo de uma ou mais GPUs com código Java, com perda mínima de desempenho.

Para se determinar se a ferramenta causa algum tipo de impacto negativo, foram realizadas medições dos tempos de um mesmo problema sobre as placas de vídeo de ambos os fabricantes, com as soluções tradicionais e as soluções utilizando a ferramenta de desenvolvimento TAJMAHAL.

O problema escolhido foi a multiplicação de matrizes em GPU, com matrizes quadradas de vários tamanhos.

### 5.1.1 Ambiente de Testes

Visando demonstrar o funcionamento prático da ferramenta, foi criado um exemplo de utilização da ferramenta TAJMAHAL. Para a execução deste exemplo, foi realizada a instalação da ferramenta em duas máquinas diferentes, cada uma equipada com uma placa de vídeo de cada fabricante:

Máquina A:

- **Processador**: AMD Phenom X4 2.2 Ghz;
- **Memória Principal**: 4 GB;
- **Placa gráfica**: 2 AMD Radeon HD4850 (512MB cada);
- **Sistema Operacional**: GNU/Linux versão 2.6.25-9 (x86-64);
- **Java**: Sun Java 1.6;
- **Stream SDK**: 1.2.1-beta.

Máquina B:

- **Processador**: AMD Athlon X2 2.8 Ghz;
- **Memória Principal**: 2 GB;
- **Placa gráfica**: NVIDIA GeForce 9600GT (512MB);
- **Sistema Operacional**: GNU/Linux versão 2.6.24-19 (x86);
- **Java**: Sun Java 1.6;
- **CUDA**: 2.0-beta.

O passo seguinte foi a execução das seguintes quatro implementações:

- Máquina A: Stream SDK
- Máquina A: Stream SDK + TAJMAHAL
- Máquina B: CUDA
- Máquina B: CUDA + TAJMAHAL

### 5.1.2 Resultados

Para cada uma das implementações, foi realizada uma medição de cada um dos tempos: entrada/saída, tempo de processamento, que inclui a alocação de memória do sistema e da GPU, e tempo total de execução. Cada tempo foi medido 10 vezes e o resultado presente nas tabelas representa a mediana desses 10 valores. O número de linhas de cada matriz quadrada usada em cada teste está representado nas colunas das tabelas abaixo. Os tempos de todas as tabelas estão em segundos.

O resultado dos testes realizados na máquina A está nas tabelas 5.3 e 5.4. A tabela 5.3 ilustra o resultado obtido na implementação que somente usa Stream SDK, e a tabela 5.4 a implementação que utiliza Stream SDK juntamente com a ferramenta TAJMAHAL.

Analogamente, o resultado dos testes realizados na máquina B está nas tabelas 5.1 e 5.2. A tabela 5.1 demonstra o resultado obtido na implementação que somente usa CUDA, e a tabela 5.2 a implementação que utiliza CUDA juntamente com a ferramenta TAJMAHAL.

Os resultados das tabelas foram similares: A diferença de tempos de entrada e saída deve-se a implementações diferentes em C++ e em Java. Este foi o motivo do aparente ganho de desempenho com o uso da ferramenta TAJMAHAL. Na verdade, este ganho não ocorreu devido ao uso da ferramenta, e sim a uma implementação pouco otimizada do código de entrada/saída C++. A real sobrecarga da ferramenta pode ser visualizada na diferença dos tempos de processamento de cada matriz nas tabelas 5.1 e 5.2, e também na diferença desses mesmos tempos nas tabelas 5.3 e 5.4. Esta sobrecarga é significativa nas matrizes menores, porém ela vai diminuindo consideravelmente à medida que o tamanho da matriz aumenta.

Tabela 5.1: Medição dos tempos da implementação do problema de multiplicação de matrizes em CUDA.

| CUDA | 64 | 128 | 256 | 512 | 1024 | 2048 |
|---|---|---|---|---|---|---|
| Entrada/Saída | 0,01 | 0,04 | 0,16 | 0,72 | 2,73 | 11,10 |
| Processamento | <10ms | <10ms | <10ms | 0,01 | 0,04 | 0,34 |
| Tempo de execução | 0,05 | 0,08 | 0,20 | 0,77 | 2,83 | 11,50 |

Tabela 5.2: Medição dos tempos da implementação do problema de multiplicação de matrizes em CUDA, utilizando a ferramenta TAJMAHAL.

| TAJMAHAL/CUDA | 64 | 128 | 256 | 512 | 1024 | 2048 |
|---|---|---|---|---|---|---|
| Entrada/Saída | 0,26 | 0,37 | 0,49 | 0,93 | 2,28 | 8,70 |
| Processamento | 0,04 | 0,04 | 0,04 | 0,05 | 0,09 | 0,39 |
| Tempo de execução | 0,31 | 0,42 | 0,54 | 0,97 | 2,39 | 9,36 |

Tabela 5.3: Medição dos tempos da implementação do problema de multiplicação de matrizes em Stream SDK.

| Stream SDK | 64 | 128 | 256 | 512 | 1024 | 2048 |
|---|---|---|---|---|---|---|
| Entrada/Saída | 0,01 | 0,04 | 0,15 | 0,60 | 2,47 | 10,02 |
| Processamento | 0,32 | 0,32 | 0,32 | 0,33 | 0,35 | 0,47 |
| Tempo de execução | 0,33 | 0,36 | 0,47 | 0,93 | 2,84 | 10,57 |

Tabela 5.4: Medição dos tempos da implementação do problema de multiplicação de matrizes em Stream SDK, utilizando a ferramenta TAJMAHAL.

| TAJMAHAL/Stream SDK | 64 | 128 | 256 | 512 | 1024 | 2048 |
|---|---|---|---|---|---|---|
| Entrada/Saída | 0,13 | 0,20 | 0,30 | 0,59 | 1,49 | 6,12 |
| Processamento | 0,33 | 0,33 | 0,33 | 0,34 | 0,37 | 0,51 |
| Tempo de execução | 0,46 | 0,54 | 0,63 | 0,94 | 1,87 | 6,66 |

## 5.2 Avaliação Qualitativa

Como mencionado no capítulo anterior, a ferramenta além de garantir que o desempenho seja mantido o mais próximo ao original possível, ela também deverá permitir que o programador tenha mais flexibilidade no momento de optar entre o *hardware* de um fabricante quanto o outro para realizar a computação a nível de GPU. Ainda outra característica desejável é que a programação GPGPU seja facilitada.

### 5.2.1 Resultados

A programação GPGPU foi facilitada, já que a ferramenta TAJMAHAL permite a programação de parte do problema em Java que, comparada a C++, geralmente é menos propensa a erros. Isto pode ser visualizado no apêndice C.5, que ilustra o código de mais alto nível necessário, após a configuração da ferramenta, para a computação da ordenação na GPU de 512 elementos. Também existe um ganho de produtividade ao disponibilizar aos programadores as classes TAJMAHAL, como a classe *Backend* (apêndice B.3) e a classe abstrata *AbstractSharedBuffer* (apêndice B.1), já que estas implementam funcionalidades necessárias e comuns aos projetos TAJMAHAL.

O uso desta ferramenta requer uma certa proximidade com o *hardware* gráfico, já que o programador terá que programar as implementações de baixo nível ou utilizar alguma existente, e mesmo que utilize, esta terá que ser ligeiramente alterada para ser utilizada com JNI. Algumas destas alterações podem ser visualizadas nos apêndices C.7, C.8 e C.8, realizadas no algoritmo de ordenação bitônica. O *kernel*, que é a parte do código que executa na GPU, não foi alterado. O *header* JNI do apêndice C.6 é gerado automaticamente pela Makefile a partir da classe Java que implementa os métodos nativos; neste caso a classe é a GPUSorter, cuja implementação está no apêndice C.3 .

Na Engenharia de Software, uma métrica comum para a produtividade de um programador é o número de linhas de código escritas por dia (GUSTAFSON, 2002). Em ambos os casos, houve uma melhora considerável na produtividade, já que bastam poucas linhas de código para se realizar uma computação na placa gráfica se for feito uso das *Makefiles* e classes auxiliares da ferramenta TAJMAHAL. A escrita de código economizada nesse ponto permite que o programador invista mais tempo em outras partes do projeto. O uso de Java em parte do projeto também pode ser visto como um benefício, já que, como foi citado na seção 4.1.1, Java é uma linguagem menos propensa a erros do que C++. Isto é

outra forma de melhorar a produtividade.

Além do programador agora poder realizar programação GPGPU com poucas linhas de código e ter a possibilidade de o integrar a uma aplicação Java existente, ele também terá uma facilidade adicional. Notou-se que se um programador possui duas implementações para um mesmo problema, uma para cada GPU, talvez deseje alternar automaticamente entre aquelas, dependendo de qual hardware estiver instalado no momento. Isto é implementado na classe auxiliar Backend (apêndice B.3). Esta classe não é uma implementação nativa, e depende do enum AvailableBackends (apêndice B.2). Esta, como o nome indica, contém enumerações dos *backends* disponíveis em uma dada versão da ferramenta de desenvolvimento TAJMAHAL e do *shell script* gpu_detect.sh (apêndice B.9). Estas classes *Backend* proporcionam uma maior flexibilidade ao programador, já que permitem que este descubra que *backend* está disponível e o selecione em tempo de execução.

# 6 CONCLUSÃO

Este trabalho apresentou a implementação de uma ferramenta de desenvolvimento que permite que um programador crie projetos em Java habilitados a realizar algum tipo de computação GPGPU, podendo optar entre dois *backends* em tempo de execução.

Primeiramente, procurou-se conhecer um pouco da história e da evolução da programação GPGPU, até às ferramentas disponíveis atualmente. Então, os SDKs dos maiores fabricantes das GPUs da atualidade foram obtidos, instalados e estudados, para entender como poderia ser realizada a sua integração com Java.

A ferramenta JNI possibilita esta integração, então a sua API foi estudada para se determinar uma forma de realizar a passagem de dados entre Java e o *backend* com perda mínima de desempenho. Notou-se que os passos para realizar esta integração eram numerosos, e que alguma automatização era necessária. As GNU *Makefiles* forneceram esta funcionalidade.

No estágio atual da implementação, os elementos da ferramenta de desenvolvimento TAJMAHAL automatizam parte do processo, porém o *backend* deve ser levemente alterado para funcionar com JNI. A versão atual da ferramenta TAJMAHAL possui uma limitação: Uma vez que o primeiro *backend* for escolhido em tempo de execução, não será possível carregar outro backend nessa mesma execução. Isso se deve ao fato das bibliotecas nativas implementarem métodos nativos com o mesmo nome em todos os *backends*. Estima-se que a grande maioria dos casos não necessite de realizar esta operação, então não foi uma prioridade oferecer essa funcionalidade em uma primeira versão da ferramenta. Poderá eventualmente ser implementada como trabalho futuro.

A avaliação se deu através de medições de tempo com e sem o uso da ferramenta. Estima-se que a ferramenta será utilizada em projetos que realizem computações sobre conjuntos de dados maiores do que os testados para justificar o uso da GPGPU. Como o

*overhead* tende a ocupar uma porcentagem cada vez menor do tempo de processamento à medida que o conjunto de dados aumenta, podemos concluir que o *overhead* da ferramenta é baixo, viabilizando o seu uso. O aumento de produtividade também foi observado através da necessidade de escrita de poucas linhas de código e do uso de classes auxiliares. Apesar do programador necessitar de possuir algum conhecimento da programação nos SDKs, o código do *kernel* que executa na GPU não necessita de nenhuma alteração.

No final deste trabalho de graduação, foi lançada a primeira versão da especificação OpenCL. As empresas que suportam este padrão deverão implementá-lo nos seus *drivers* em um futuro próximo. O futuro deste padrão é promissor, porém isto irá depender da qualidade de implementação das empresas que o suportam, e será necessário aguardar algum tempo até que as implementações OpenCL fiquem estáveis e sejam amplamente utilizadas pelo mercado. Se isto acontecer, será em detrimento de outras ferramentas, como CUDA e Stream SDK. Neste cenário, um trabalho futuro poderá ser a alteração da ferramenta para suportar esse novo *backend*, o OpenCL.

# REFERÊNCIAS

Advanced Micro Devices Inc. **Stream SDK**. Disponível em: http://ati.amd.com/technology/streamcomputing/sdkdwnld.html . Acesso em: 07 de novembro de 2008.

Advanced Micro Devices Inc. **Close To Metal**. Disponível em: http://sourceforge.net/projects/amdctm . Acesso em: 07 de novembro de 2008.

Advanced Micro Devices Inc. **Brook+**. Disponível em: http://ati.amd.com/technology/streamcomputing/Advanced Micro Devices Inc.-Brookplus.pdf . Acesso em: 07 de novembro de 2008.

Advanced Micro Devices Inc. **Stream Computing User Guide 1.1**. Disponível em: http://ati.amd.com/technology/streamcomputing/sdkdwnld.html. Acesso em: 07 de novembro de 2008.

CÁCERES, E. N.; MONGELLI, H. **Algoritmos Paralelos**: ordenação. Campo Grande, MS: Universidade Federal de Mato Grosso do Sul, 2002. Disponível em: www.dct.ufms.br/ mongelli/disciplinas/mestrado/ordena.ps . Acesso em: 5 de dezembro de 2008.

Free Software Foundation. **GNU Make**. Disponível em: http://www.gnu.org/software/make/ . Acesso em: 07 de novembro de 2008.

GUSTAFSON, D. A. **Theory and Problems of Software Engineering**. [S.l.]: Computing and Information Sciences Department, Kansas State University, 2002. (Schaum's Outline Series).

HOUSTON, M. **General Purpose Computation on Graphics Processors (GPGPU)**. Palo Alto, California, USA: Stanford University, 2005.

IDG Now! **Google adquire empresa de software para servidores PeakStream**. Disponível em: http://idgnow.uol.com.br/mercado/2007/06/06/idgnoticia.2007-06-06.9967627580/ . Acesso em: 07 de novembro de 2008.

J.P.LEWIS; NEUMANN, U. **Performance of Java versus C++**. Los Angeles, California, USA: Univeristy of Southern California, 2003.

Khronos Group. **Khronos OpenCL API Registry**. Disponível em: http://www.khronos.org/registry/cl/ . Acesso em: 9 de dezembro de 2008.

LEMAY, L.; CADENHEAD, R. **Sams Teach Yourself Java 2 in 21 Days**. 2.ed. [S.l.]: Sams, 2001. Disponível em: http://workbench.cadenhead.org/book/21java/excerpt.shtml . Acesso em: 07 de novembro de 2008.

LIANG, S. . [S.l.]: SUN CORPORATION, 2002. Disponível em: http://java.sun.com/docs/books/jni/html/titlepage.html . Acesso em: 07 de novembro de 2008.

LUEBKE, D.; HARRIS, M. **GPGPU**: general purpose computation on graphics hardware. Charlottesville, Virginia, USA: Univeristy of Virginia, 2004.

LUEBKE, D.; HARRIS, M. **SIGGRAPH 2005 GPGPU COURSE**. Charlottesville, Virginia, USA: Univeristy of Virginia, 2005.

NVIDIA Corporation. **Compute Unified Device Architecture**. Disponível em: http://www.nvidia.com/cuda . Acesso em: 07 de novembro de 2008.

NVIDIA Corporation. **NVIDIA Corporation CUDA User Guide 2.0**. Disponível em: http://www.nvidia.com/object/cuda_get.html . Acesso em: 07 de novembro de 2008.

PHIPPS, G. Comparing Observed Bug and Productivity Rates for Java and C++. **Software - Practice & Experience**, [S.l.], v.29, n.4, 1999.

RapidMind Inc. **libSh**. Disponível em: http://libsh.org/ . Acesso em: 07 de novembro de 2008.

RapidMind Inc. **RapidMind**. Disponível em: http://www.rapidmind.net/ . Acesso em: 07 de novembro de 2008.

Stanford University. **Brook for GPUs**. Disponível em: http://www-graphics.stanford.edu/projects/brookgpu/ . Acesso em: 07 de novembro de 2008.

STROUSTRUP, B. **The C++ Programming Language**. Disponível em: http://www.research.att.com/ bs/C++.html. Acesso em: 07 de novembro de 2008.

The Tech Report. **OpenCL 1.0 spec released, Advanced Micro Devices Inc. and Nvidia are on board**. Disponível em: http://techreport.com/discussions.x/16024 . Acesso em: 9 de dezembro de 2008.

Tom's Hardware. **Nvidia's CUDA**: the end of the cpu? Disponível em: http://www.tomshardware.com/reviews/nvidia-cuda-gpu,1954-5.html . Acesso em: 07 de novembro de 2008.

# APÊNDICE A *PATCHES* CRIADOS PARA CUDA E STREAM SDK

**Código-fonte A.1** Patch main.mk.patch a ser aplicado em CUDA

```
--- main.mk_original 2008-12-21 01:17:12.000000000 -0200
+++ main.mk 2008-12-21 11:23:31.000000000 -0200
@@ -220,6 +220,11 @@
 # $(MANIFEST_TOOL) -manifest $@.manifest \
 #  "-outputresource:$@;1"
 #endif
 endif
+ifdef GENERATE_SHARED_LIBRARY
+$(OUTPUTDIR)/$(OUTPUT): output_dir $(OBJS)
+ $(ECHO) Building $@
+ $(LD) $(LDFLAGS) $(LD_OUTPUT_FLAG)$@ $(OBJS) $(ADDLIBS)
+endif
 ifdef GENERATE_STATIC_LIBRARY
 $(OUTPUTDIR)/$(OUTPUT): output_dir $(OBJS)
  $(ECHO) Building $@
@@ -233,13 +238,13 @@

 $(OBJDIR)/%$(OBJSUFFIX): %.cpp $(DEPDIR)/%.depend
  $(MKDIR) $(OBJDIR)
-$(CC) $(CFLAGS) $(C_OUTPUT_FLAG)$@ $(C_COMPILE_FLAG) \
$(C_INCLUDE_FLAG)$(INCLUDEDIR) $<
+ $(CC) $(CFLAGS) $(C_OUTPUT_FLAG)$@ $(C_COMPILE_FLAG) \
$(C_INCLUDE_FLAG)$(INCLUDEDIR) $<

 # .c -> object

 $(OBJDIR)/%$(OBJSUFFIX): %.c $(DEPDIR)/%.depend
  $(MKDIR) $(OBJDIR)
-$(CC) $(CFLAGS) $(C_OUTPUT_FLAG)$@ $(C_COMPILE_FLAG) \
$(C_INCLUDE_FLAG)$(INCLUDEDIR) $<
+ $(CC) $(CFLAGS) $(C_OUTPUT_FLAG)$@ $(C_COMPILE_FLAG) \
$(C_INCLUDE_FLAG)$(INCLUDEDIR) $<

 # .br -> .cpp

@@ -253,7 +258,7 @@

 $(OBJDIR)/%$(OBJSUFFIX): $(OBJDIR)/%.cpp
  $(MKDIR) $(OBJDIR)
-$(CC) $(CFLAGS) $(C_OUTPUT_FLAG)$@ $(C_COMPILE_FLAG) \
       $(C_INCLUDE_FLAG)$(INCLUDEDIR) $(C_INCLUDE_FLAG). $<
+ $(CC) $(CFLAGS) $(C_OUTPUT_FLAG)$@ $(C_COMPILE_FLAG) \
       $(C_INCLUDE_FLAG)$(INCLUDEDIR) $(C_INCLUDE_FLAG). \
       $(C_INCLUDE_FLAG)$(JAVA_HOME)/include \
       $(C_INCLUDE_FLAG)$(JAVA_HOME)/include/linux $<

 # Regression testing (both running on this machine and \
       packaging tests for running on other machines)
```

**Código-fonte A.2** Patch common.mk.patch a ser aplicado em Stream SDK

```
--- common.mk_original 2008-12-20 22:44:34.497725360 -0200
+++ common.mk 2008-12-20 22:45:53.441723120 -0200
@@ -67,7 +67,8 @@
 LINK       := g++ -fPIC

 # Includes
-INCLUDES  += -I. -I$(CUDA_INSTALL_PATH)/include \
+INCLUDES  += -I. -I$(CUDA_INSTALL_PATH)/include \
      -I$(COMMONDIR)/inc
+INCLUDES  += -I. -I$(CUDA_INSTALL_PATH)/include \
      -I$(COMMONDIR)/inc -I$(JAVA_HOME)/include \
      -I$(JAVA_HOME)/include/linux
+

 # architecture flag for cubin build
 CUBIN_ARCH_FLAG := -m32
@@ -203,7 +204,21 @@
 ifneq ($(STATIC_LIB),)
  TARGETDIR := $(LIBDIR)
  TARGET   := \
    $(subst .a,$(LIBSUFFIX).a,$(LIBDIR)/$(STATIC_LIB))
-LINKLINE  = ar qv $(TARGET) $(OBJS)
+ LINKLINE  = ar qv $(TARGET) $(OBJS)
+else ifneq ($(SHARED_LIB),)
+ LIB += -lcutil$(LIBSUFFIX)
+ # Device emulation configuration
+ ifeq ($(emu), 1)
+ NVCCFLAGS   += -deviceemu
+ CUDACCFLAGS +=
+ BINSUBDIR   := emu$(BINSUBDIR)
+ # consistency, makes developing easier
+ CXXFLAGS += -D__DEVICE_EMULATION__
+ CFLAGS += -D__DEVICE_EMULATION__
+ endif
+ TARGETDIR := $(LIBDIR)
+ TARGET   := \
    $(subst .so,$(LIBSUFFIX).so,$(LIBDIR)/$(SHARED_LIB))
+ LINKLINE  = $(LINK) -o $(TARGET) $(OBJS) $(LIB) -shared
 else
  LIB += -lcutil$(LIBSUFFIX)
  # Device emulation configuration
```

# APÊNDICE B COMPONENTES-BASE DA FERRAMENTA TAJMAHAL

**Código-fonte B.1** Classe auxiliar abstrata AbstractSharedBuffer da ferramenta TAJMAHAL.

```
package TAJMAHAL;
import TAJMAHAL.*;
import java.io.*;
import java.nio.*;
import java.util.StringTokenizer;
import java.lang.reflect.*;

public abstract class AbstractSharedBuffer<B> {
    protected String bufferType;
    protected B buffer;

    public AbstractSharedBuffer(String type, int size) {
        bufferType = type;
        if(bufferType.equals("Float")) {
            this.buffer = (B) ByteBuffer.allocateDirect(size * SIZE_INT)
                .order(ByteOrder.nativeOrder()).asFloatBuffer();
        }
        else if(bufferType.equals("Int")) {
            this.buffer = (B) ByteBuffer.allocateDirect(size * SIZE_INT)
                .order(ByteOrder.nativeOrder()).asIntBuffer();
        }
        else if(bufferType.equals("Byte"))
        {
            this.buffer = (B) ByteBuffer.allocateDirect(size * SIZE_INT)
                .order(ByteOrder.nativeOrder());
        }
        else {
            System.err.println("TAJMAHAL: Invalid SharedBuffer type");
        }
    }
    public abstract B getInternalBuffer();
    public abstract void loadFromTextFile(
            String inputFileName, int size, int columns);
    public abstract void saveToTextFile(
            String outputFileName, int size, int columns);
    public abstract void loadFromBinaryFile(
            String inputFileName, int size, int columns);
    public abstract void saveToBinaryFile(
            String outputFileName, int size, int columns);
    public static final int SIZE_BYTE = 1;
    public static final int SIZE_CHAR = 2;
    public static final int SIZE_SHORT = 2;
    public static final int SIZE_INT = 4;
    public static final int SIZE_FLOAT = 4;
    public static final int SIZE_LONG = 8;
    public static final int SIZE_DOUBLE = 8;
}
```

**Código-fonte B.2** Classe auxiliar AvailableBackends da ferramenta TAJMAHAL.

```
package TAJMAHAL;

public enum AvailableBackends
{
    NONE, STREAM, CUDA;
}
```

**Código-fonte B.3** Parte I da classe auxiliar Backend da ferramenta TAJMAHAL.

```
package TAJMAHAL;
import TAJMAHAL.*;
import java.util.*;
import java.io.*;

public class Backend
{
    public Backend()
    {
        this.selectedBackend = AvailableBackends.NONE;
    }

    public AvailableBackends get()
    {
        return selectedBackend;
    }

    public void set(AvailableBackends backend)
    {
        switch(backend)
        {
            case STREAM:
                selectedBackend = AvailableBackends.STREAM;
                break;
            case CUDA:
                selectedBackend = AvailableBackends.CUDA;
                break;
            default:
                selectedBackend = AvailableBackends.NONE;
        }
    }

    public void set(int backend)
    {
        switch(backend)
        {
            case 1:
                selectedBackend = AvailableBackends.CUDA;
                break;
            case 2:
                selectedBackend = AvailableBackends.STREAM;
                break;
            default:
                selectedBackend = AvailableBackends.NONE;
        }
    }
```

**Código-fonte B.4** Parte II da classe auxiliar Backend da ferramenta TAJMAHAL.

```
    public void detect() {
        try
        {
            String cmd = "./" + gpuDetectScript;
            Runtime rt = Runtime.getRuntime();
            Process proc = rt.exec(cmd);

            // any error???
            StreamGobbler errorGobbler = new
                StreamGobbler(proc.getErrorStream(), "ERR");

            // any output?
            StreamGobbler outputGobbler = new
                StreamGobbler(proc.getInputStream(), "OUT");

            // kick them off
            errorGobbler.start();
            outputGobbler.start();

            // wait for bash script to return
            set(proc.waitFor());
        } catch (IOException ioe)
        {
            System.err.println(
                  "Could not detect backend: gpu_detect.sh not found");
        } catch (Throwable t)
        {
            t.printStackTrace();
        }
    }

    // Backend class usage example
    public static void main(String args[])
    {
        Backend b = new Backend();
        System.out.println("Selected: "+b.get());
        b.detect();
        System.out.println("Selected: "+b.get());
        b.set(AvailableBackends.STREAM);
        System.out.println("Selected: "+b.get());
    }

    private String gpuDetectScript = "../gpu_detect.sh";
    private AvailableBackends selectedBackend;
}
```

**Código-fonte B.5** Classe auxiliar StreamGobbler da ferramenta TAJMAHAL.

```
package TAJMAHAL;

import java.util.*;
import java.io.*;

// Java World -- Fueling Innovation
// When Runtime.exec() won't -- Michael C. Daconta
// http://www.javaworld.com/javaworld/jw-12-2000/jw-1229-traps.html
// December 2004
// IDG

class StreamGobbler extends Thread {
    InputStream is;
    String type;
    OutputStream os;

    StreamGobbler(InputStream is, String type) {
        this(is, type, null);
    }
    StreamGobbler(InputStream is, String type, OutputStream redirect) {
        this.is = is;
        this.type = type;
        this.os = redirect;
    }

    public void run() {
        try {
            PrintWriter pw = null;
            if (os != null)
                pw = new PrintWriter(os);

            InputStreamReader isr = new InputStreamReader(is);
            BufferedReader br = new BufferedReader(isr);
            String line=null;
            while ( (line = br.readLine()) != null) {
                if (pw != null)
                    pw.println(line);
                System.out.println(type + ">" + line);
            }
            if (pw != null)
                pw.flush();
        } catch (IOException ioe) {
            ioe.printStackTrace();
        }
    }
}
```

**Código-fonte B.6** *Makefile* localizado na raiz da ferramenta TAJMAHAL

```
# TAJMAHAL Makefile used for applying SDK patches
# Paths must not include trailing slash
SDK_PATCHES := ./sdk_patches
STREAM_ROOTDIR := /usr/local/amdbrook
STREAM_PATCH := $(SDK_PATCHES)/main.mk.patch
CUDA_ROOTDIR := $(HOME)/NVIDIA_CUDA_SDK
CUDA_PATCH := $(SDK_PATCHES)/common.mk.patch

all:
$(JAVA_HOME)/bin/javac -cp . *.java

patches: patch_cuda patch_stream

patch_cuda:
patch -p0 $(CUDA_ROOTDIR)/common/common.mk < $(CUDA_PATCH)

patch_stream:
patch -p0 $(STREAM_ROOTDIR)/samples/utils/build/main.mk \
   < $(STREAM_PATCH)

full: clean

clean:
 rm -f hs_err*.log
 rm -f *.class
 rm -f obj/release/*o
 rm -f native/*.so
```

**Código-fonte B.7** Parte I da *Makefile* tajmahal.mk com variáveis comuns a todos os projetos TAJMAHAL. Localizado na raiz da ferramenta TAJMAHAL.

```
# Makefile for CUDA / JNI and Stream SDK / JNI
# Generic Makefile for TAJMAHAL projects
# Autodetect bit size
define Bit_size
if [ '$(shell /bin/uname -m)' = 'x86_64' ]; then
   echo 64;
else
   echo 32;
fi
endef
# If 64 bits, use -fPIC on Stream Makefile
define fPIC
if [ '${BIT}' = '64' ]; then
   echo -fPIC;
fi
endef

BIT := $(shell $(Bit_size))
FPIC := $(shell $(fPIC))
# Paths must not include trailing slash
TAJMAHAL_PATH := ..
TAJMAHAL_PROJECT_PATH := $(TAJMAHAL_PATH)/$(TAJMAHAL_PROJECT)
STREAM_TARGET_LIB :=  \
 $(TAJMAHAL_PROJECT_PATH)/libjava-$(TAJMAHAL_PROJECT)-stream.so
CUDA_TARGET_LIB :=  \
 $(TAJMAHAL_PROJECT_PATH)/libjava-$(TAJMAHAL_PROJECT)-cuda.so
JNI_HEADER := $(subst .,_,$(JNI_CLASS)).h

# CUDA
CUDA_SDK_PATH := $(HOME)/NVIDIA_CUDA_SDK
CUDA_PROJECT_ROOT := $(CUDA_SDK_PATH)/projects/
CUDA_PROJECT_PATH := $(CUDA_PROJECT_ROOT)/$(CUDA_PROJECT_NAME)
CUDA_INCLUDE_PATH  :=  $(CUDA_SDK_PATH)/common/inc
CUDA_LIB_PATH  :=  $(CUDA_SDK_PATH)/lib
CUDA_GENERATED_LIB :=
   $(shell /bin/grep SHARED_LIB \
   backends/cuda/$(CUDA_PROJECT_NAME)/Makefile \
   | awk '{ print $$3 }')

# STREAM SDK
STREAM_BROOK_PATH  :=  /usr/local/amdbrook
STREAM_PROJECT_ROOT := $(STREAM_BROOK_PATH)/samples/apps/
STREAM_PROJECT_PATH := \
 $(STREAM_PROJECT_ROOT)/$(STREAM_PROJECT_NAME)
STREAM_INCLUDE_PATH  :=  $(STREAM_BROOK_PATH)/sdk/include
STREAM_LIB_PATH  :=  $(STREAM_BROOK_PATH)/sdk/lib
STREAM_GENERATED_LIB := $(STREAM_BROOK_PATH)/samples/lib \
   /lnx_x86_${BIT}/lib$(basename $(STREAM_JNI_FILENAME)).so
```

**Código-fonte B.8** Parte II da *Makefile* tajmahal.mk com variáveis comuns a todos os projetos TAJMAHAL. Localizado na raiz da ferramenta TAJMAHAL.

```
all: copy_backends cuda brook

cuda: copy_cuda_backend java
 cp $(JNI_HEADER) $(CUDA_PROJECT_PATH)
 make -C $(CUDA_PROJECT_PATH)
 cp $(CUDA_PROJECT_PATH)/../../lib/$(CUDA_GENERATED_LIB) \
    $(CUDA_TARGET_LIB)

brook: copy_stream_backend java
 cp $(JNI_HEADER) $(STREAM_PROJECT_PATH)
 FPIC=${FPIC} make -C $(STREAM_PROJECT_PATH)
 $(shell if [ -f $(STREAM_JNI_FILENAME) ]; then
       touch $(STREAM_JNI_FILENAME); fi)
 cp -ub $(STREAM_PROJECT_PATH)/built/$(STREAM_JNI_FILENAME) .
 @echo "Please make sure ./$(STREAM_JNI_FILENAME) has been \
    altered to work with JNI. If it hasn't, add JNI support \
    before executing 'make streamlib'."

streamlib:
@echo $(shell if [ ! -f $(STREAM_JNI_FILENAME) ]; then
      echo "Please type make brook first."; exit; fi)
cp $(STREAM_JNI_FILENAME) $(STREAM_PROJECT_PATH)/built
rm -rf $(STREAM_PROJECT_PATH)/built/*.o
FPIC=${FPIC} make -C $(STREAM_PROJECT_PATH)
cp $(STREAM_GENERATED_LIB) $(STREAM_TARGET_LIB)

copy_cuda_backend:
cp -af ./backends/cuda/$(CUDA_PROJECT_NAME) \
$(CUDA_PROJECT_ROOT)

copy_stream_backend:
cp -af ./backends/stream/$(STREAM_PROJECT_NAME) \
$(STREAM_PROJECT_ROOT)

copy_backends: copy_cuda_backend copy_stream_backend

java:
 $(JAVA_HOME)/bin/javac -cp .:$(TAJMAHAL_PATH)/../ *.java
 $(JAVA_HOME)/bin/javah -jni \
 -classpath .:$(TAJMAHAL_PATH)/../ $(JNI_CLASS); \

clean:
 rm -f *.class $(CUDA_TARGET_LIB) $(STREAM_TARGET_LIB) \
 $(JNI_HEADER)
javaclean:
/bin/rm -f *.class
nativeclean:
rm -f $(CUDA_TARGET_LIB) $(STREAM_TARGET_LIB)
```

**Código-fonte B.9** Shell script para autodetectar o *hardware* acelerador gráfico disponível

```
#!/bin/bash
if ( $(lsmod | grep -q nvidia) ); then
   exit 1
elif ( $(lsmod | grep -q fglrx) ); then
   exit 2
else
   exit 0
fi
```

# APÊNDICE C EXEMPLO DE PROJETO TAJMAHAL: *BITONICSORT*

**Código-fonte C.1** Classe SharedBuffer usada nos projetos *BitonicSort* e *MatrixMul*. Exemplo de implementação da classe abstrata AbstractSharedBuffer da ferramenta TAJMAHAL.

```
package TAJMAHAL.BitonicSort;

import TAJMAHAL.*;
import java.io.*;
import java.nio.*;
import java.util.StringTokenizer;

public class SharedBuffer<B> extends AbstractSharedBuffer<FloatBuffer> {

    public SharedBuffer(String type, int size) {
        super(type,size);
    }

    public FloatBuffer getInternalBuffer()
    {
        return buffer;
    }

    public void loadFromTextFile(String inputFileName, int size, int columns) {
        // ...
    }

    public void saveToTextFile(String outputFileName, int size, int columns) {
        // ...
    }

    public void loadFromBinaryFile(String inputFileName, int size, int columns) {
        // Implement me and remove the line below
        System.err.println(
              "SharedBuffer: loadFromBinaryFile not implemented"
        );
    }

    public void saveToBinaryFile(String outputFileName, int size, int columns) {
        // Implement me and remove the line below
        System.err.println(
              "SharedBuffer: saveToBinaryFile not implemented"
        );
    }
}
```

**Código-fonte C.2** Classe Matrix usada nos projetos *BitonicSort* e *MatrixMul*. Exemplo de uso da classe SharedBuffer.

```
package TAJMAHAL.BitonicSort;
import TAJMAHAL.*;
import java.nio.*;

public class Matrix {
    private int width, height, length;
    private SharedBuffer<FloatBuffer> buffer;

    public Matrix(String elementType, int width, int height)
    {
        this.width = width;
        this.height = height;
        length = width * height;
        if(height < 1 || width < 1 || length < 1)
        {
            System.err.println(
                  "Matrix: Invalid Dimensions: "
                  +height+"x"+width+"x"+"="+length);
        }
        buffer = new SharedBuffer(elementType, length);
    }

    public void loadMatrixData(String inputFileName)
    {
        buffer.loadFromTextFile(inputFileName, length, width);
    }
    public void saveMatrixData(String outputFileName)
    {
        buffer.saveToTextFile(outputFileName, length, width);
    }
    public int getWidth()
    {
        return width;
    }
    public int getHeight()
    {
        return height;
    }
    public int getLength()
    {
        return length;
    }
    public SharedBuffer getSharedBuffer()
    {
        return buffer;
    }
}
```

**Código-fonte C.3** Parte I da classe GPUSorter com o método nativo JNI. Classe utilizada na Main do projeto *BitonicSort* para ordenar os elementos.

```
package TAJMAHAL.BitonicSort;

import TAJMAHAL.*;
import java.nio.*;

public class GPUSorter {

    public GPUSorter(Backend backend) {
        this.projectName = getCurrentDir();
        this.backend = backend;
        loadNativeLibs();
    }

    private void loadNativeLibs()
    {
        try {
            ab = backend.get();
            switch(ab)
            {
                case STREAM:
                    System.loadLibrary("java-"+projectName+"-stream");
                    break;
                case CUDA:
                    System.loadLibrary("java-"+projectName+"-cuda");
                    break;
                default:
                    System.err.println(projectName+"No compatible GPU found.");
                    System.exit(-1);
            }
        } catch (Exception e)
        {
            e.printStackTrace();
        }
    }
```

**Código-fonte C.4** Parte II da classe GPUSorter com o método nativo JNI

```
    public int compute(Matrix M) {
        try {
            int height = M.getHeight();
            int width = M.getWidth();
            int length = M.getLength();
            //... error handling code omitted

            String[] args = new String[6];

            // Pass the arguments to the native method
            args[0] = "-x";
            args[1] = String.valueOf(width);
            args[2] = "-y";
            args[3] = String.valueOf(height);
            args[4] = "-q";
            args[5] = "-e";

            FloatBuffer fb = M.getSharedBuffer().getInternalBuffer();

            // Native invocation
            computeFloats(args, fb);
            //... catch and return omitted
    }

    // JNI Method
    public native int computeFloats(
            String[] args, FloatBuffer fb);

    private Backend backend;
    private String projectName;
    private AvailableBackends ab;
}
```

**Código-fonte C.5** Classe Main, onde o usuário utiliza o projeto TAJMAHAL *BitonicSort* para realizar ordenação bitônica de um conjunto de elementos em uma GPU a partir de Java

```
package TAJMAHAL.BitonicSort;

import TAJMAHAL.*;

public class Main {

    public static void main(String args[])
    {
        Matrix A = new Matrix("Float", 512, 1);
        A.loadMatrixData("data/random_floats.txt");

        Backend backend = new Backend();
        backend.detect();
        GPUSorter sorter = new GPUSorter(backend);

        //JNI method invocation
        sorter.compute(A);

        A.saveMatrixData("data/sorted_floats.txt");
    }
}
```

**Código-fonte C.6** Cabeçalho JNI gerado automaticamente pela *Makefile* do projeto *BitonicSort* a partir da classe GPUSorter.java. Usado pelos projetos específicos dos *backends*

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class TAJMAHAL_BitonicSort_GPUSorter */

#ifndef _Included_TAJMAHAL_BitonicSort_GPUSorter
#define _Included_TAJMAHAL_BitonicSort_GPUSorter
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     TAJMAHAL_BitonicSort_GPUSorter
 * Method:    computeFloats
 * Signature: ([Ljava/lang/String;Ljava/nio/FloatBuffer;)I
 */
JNIEXPORT jint JNICALL Java_TAJMAHAL_BitonicSort_GPUSorter_computeFloats
  (JNIEnv *, jobject, jobjectArray, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

**Código-fonte C.7** Código do projeto específico CUDA de *bitonic_sort* antes das modificações para funcionar com JNI.

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

#include "bitonic_kernel.cu"

int main(int argc, char** argv)
{
    CUT_DEVICE_INIT(argc, argv);

    float values[NUM];

    for(int i = 0; i < NUM; i++)
    {
        values[i] = rand()/(rand()*rand());
    }

    float * dvalues;
    CUDA_SAFE_CALL(cudaMalloc((void**)&dvalues, sizeof(float) * NUM));
    CUDA_SAFE_CALL(cudaMemcpy(
            dvalues, values, sizeof(float) * NUM, cudaMemcpyHostToDevice));

    bitonicSort<<<1, NUM, sizeof(float) * NUM>>>(dvalues);

    // check for any errors
    CUT_CHECK_ERROR("Kernel execution failed");

    CUDA_SAFE_CALL(cudaMemcpy(
            values, dvalues, sizeof(float) * NUM, cudaMemcpyDeviceToHost));

    CUDA_SAFE_CALL(cudaFree(dvalues));

    bool passed = true;
    for(int i = 1; i < NUM; i++)
    {
        if (values[i-1] > values[i])
        {
            passed = false;
        }
    }

    printf( "Test %s\n", passed ? "PASSED" : "FAILED");

    CUT_EXIT(argc, argv);
}
```

**Código-fonte C.8** Parte I do código do projeto específico CUDA *bitonic_sort* após modificações para funcionar com JNI.

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

#include "bitonic_kernel.cu"

#include "TAJMAHAL_BitonicSort_GPUSorter.h"

//int main(int argc, char** argv)
JNIEXPORT jint JNICALL Java_TAJMAHAL_BitonicSort_GPUSorter_computeFloats
  (JNIEnv *jenv, jobject self, jobjectArray jargv, jobject jByteBufferA)
{
    // Start of JNI Code (1)
    int argc=-1;
    const char* argv[128];
    argv[0] = "Bitonic Sort";

    argc = jenv->GetArrayLength(jargv);

    jobject jargvObject[128];
    for (int i=1; i<argc+1; i++)
    {
        jargvObject[i-1] = jenv->GetObjectArrayElement(jargv, i-1);
        argv[i] = jenv->GetStringUTFChars((jstring) jargvObject[i-1], 0);
    }
    argc++;

    unsigned int width=512;
    unsigned int height=1;
    int x;

    for (x = 1; x < argc; ++x) {
        switch (argv[x][1]) {
            case 'x':
                if (++x < argc) {
                    sscanf(argv[x], "%u", &width);
                }
                else {
                    fprintf(stderr, "Error: Invalid argument, %s", argv[x-1]);
                    exit(-1);
                }
                break;
            case 'y':
                if (++x < argc) {
                    sscanf(argv[x], "%u", &height);
                }
```

**Código-fonte C.9** Parte II do código do projeto específico CUDA *bitonic_sort* após modificações para funcionar com JNI.

```
                else {
                    fprintf(stderr, "Error: Invalid argument, %s", argv[x-1]);
                    exit(-1);
                }
                break;
    }
    jfloat *values = (jfloat *) jenv->GetDirectBufferAddress(jByteBufferA);
    if(values == NULL)
    {
        fprintf(stderr, "BitonicSort: Null pointer (values)!\n");
        return 0;
    }
    int length = width*height;
    // End of JNI code (1)
    CUT_DEVICE_INIT(1, argv);

    float * dvalues;
    CUDA_SAFE_CALL(cudaMalloc((void**)&dvalues, sizeof(float) * length));
    CUDA_SAFE_CALL(cudaMemcpy(
                dvalues, values, sizeof(float) * length, cudaMemcpyHostToDevice));
    bitonicSort<<<1, length, sizeof(float) * length>>>(dvalues, length);

    // check for any errors
    CUT_CHECK_ERROR("Kernel execution failed");

    CUDA_SAFE_CALL(cudaMemcpy(
                values, dvalues, sizeof(float) * length, cudaMemcpyDeviceToHost));

    CUDA_SAFE_CALL(cudaFree(dvalues));
    bool passed = true;
    for(int i = 1; i < length; i++)
    {
        if (values[i-1] > values[i])
        {
            passed = false;
        }
    }
    printf( "Test %s\n", passed ? "PASSED" : "FAILED");
    // Start of JNI code (2)
    for (int i = 1; i < argc; i++)
    {
        jenv->ReleaseStringUTFChars((jstring)jargvObject[i-1], argv[i]);
    }
    return 0; // End of JNI code (2)
}
```

**Código-fonte C.10** *Makefile* do projeto TAJMAHAL *BitonicSort*

```
# Makefile for CUDA / JNI and Stream SDK / JNI
# Project Specific Variables

TAJMAHAL_PROJECT := BitonicSort
JNI_CLASS := TAJMAHAL.$(TAJMAHAL_PROJECT).GPUSorter
CUDA_PROJECT_NAME := TAJMAHAL_bitonic
STREAM_PROJECT_NAME := TAJMAHAL_bitonic_sort
STREAM_JNI_FILENAME := bitonic_sort.cpp

include ../tajmahal.mk
```

**Código-fonte C.11** Shell script para facilitar a execução do aplicativo Java gerado pelo projeto TAJMAHAL BitonicSort

```
#!/bin/bash
java -classpath ".:../../" \
-Djava.library.path=.:/usr/lib/jni:/usr/lib \
TAJMAHAL/BitonicSort/Main "$@"
```