UNIVERSIDADE DO VALE DO RIO DOS SINOS

LUCAS CORREIA VILLA REAL

Uso de Hardware Dedicado para a Solução de Problemas Genéricos de Computação

Monografia apresentada como requisito parcial
para a obtenção do grau de
Bacharel em Ciência da Computação

Prof. Ms. Mauro Steigleder
Orientador

São Leopoldo, novembro de 2004

“Mas não basta pra ser livre
Ser forte, aguerrido e bravo
Povo que não tem virtude
Acaba por ser escravo”

FRANCISCO PINTO DA FONTOURA – “HINO RIOGRANDENSE”

AGRADECIMENTOS

A vontade de realizar este trabalho data de 2002, quando o hardware gráfico começou a apresentar unidades de processamento programáveis. Foi nesta época que conheci o Átila Bohlke Vasconcelos, que estava começando a pesquisar sobre este assunto por conta própria. Não demorou muito para que eu me interessasse também, até porque com a persuasão do Átila eu não teria outra escolha.

Foram muitos os emails trocados desde então, com relatórios empolgados a cada descoberta nova que era feita. Claro que muita informação não tinha muito fundamento, mas elas serviram para alimentar a imaginação das coisas que poderíamos fazer com uma placa de vídeo poderosa em nossas mãos. Valeu por todas as trocas de idéias, estou certo de que crescemos bastante com elas.

Nesta mesma época foi imprescindível contar com a ajuda de pessoas como o Marcelo Walter e o Mauro Steigleder (por email), que permitiram que eu colocasse os pés no chão e compreendesse muitos aspectos importantes de computação gráfica e da arquitetura do hardware gráfico. Os emails do Mauro eram uma avalanche de informações novas, que pareciam que não poderiam ser digeridas em um tempo finito. Talvez eu tenha sido intimidado de alguma forma com isto tudo, pois acabei o escolhendo como meu orientador para este trabalho de conclusão. Gostaria de agradecer muito a vocês pela disponibilidade e empenho dedicados. Agradeço também ao professor Christian Hoffsetz pela ajuda dada durante a estadia do Mauro no Canadá. As matrizes de textura nunca mais serão as mesmas depois das suas explicações.

Gostaria de agradecer também aos meus colegas Alex Garzão, pelo seu auxílio e sugestões em momentos importantíssimos, e ao Hisham Muhammad e ao Felipe Damasio, por todo o incentivo dado. Nestas horas também é ótimo contar com colegas como o Ricardo Sanchez, que levam medições de tempo ao extremo. Cada ciclo de CPU economizado e medido neste trabalho é em homenagem a ti!

Reservo um parágrafo especial ainda para meu amigo Moisés Pires de Souza, que desde que tomei interesse por computação esteve disposto a me ensinar e a me ajudar com minhas idéias malucas de software e hardware. Digo com convicção que eu não teria tomado este rumo incrível sem tua força. Muito obrigado mesmo por tudo.

Em principal, agradeço aos meus pais e à minha irmã, por todo o apoio dado nos estudos este tempo todo. Tenho certeza de que isto tudo não seria possível sem vocês. Obrigado ainda à Patrícia pela compreensão nos momentos em que precisei estudar e trabalhar sem parar, consumindo com nossos finais de semana valiosos.

Finalmente, gostaria de agradecer aos desenvolvedores da OSS e da ALSA pelos magníficos drivers de som por eles escritos que proporcionaram música contínua durante todo este trabalho.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
ARB	Architecture Review Board
CPU	Central Processing Unit
CPUID	CPU Identification
DSP	Digital Signal Processor
FFT	Fast Fourier Transform
FPU	Floating Point Unit
FXSR	Fast FPU Save and Restore
GCC	GNU C Compiler
GDI	Graphical Display Interface
GPU	Graphics Processing Unit
IA-32	Intel Architecture, 32 bits
MMX	MultiMedia eXtensions
OpenGL	Open Graphics Library
RGB	Red, Green and Blue
RGBA	Red, Green, Blue and Alpha
SIMD	Single Instruction, Multiple Data
SISD	Single Instruction, Single Data
SSE	Streaming SIMD Extension
TSC	Time Stamp Counter

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Este trabalho apresenta uma abordagem para a utilização de hardwares dedicados na computação de problemas que operam sobre dados matriciais e vetoriais de ponto flutuante. O objetivo deste trabalho consiste em definir um modelo de programação para estes tipos de dados, delegando a execução para os hardwares dedicados presentes na arquitetura local de forma automática para o usuário. Para tanto, são apresentados conceitos sobre o funcionamento de determinados hardwares dedicados. Em seguida, técnicas para realizar o mapeamento de problemas genéricos sobre estes dispositivos são apresentadas, seguido de avaliações de desempenho e conclusões finais.

Palavras-chave: Hardware dedicado, Processamento vetorial, Linguagens de programação.

The Use of Dedicated Hardware for Solving Generic Computational Problems

ABSTRACT

This work presents the use of special purpose hardware on the computation of problems from matrix and vector domain. The aim of this work is to define a programming model for this kind of data type, delegating the execution for the dedicated hardware presented on the local architecture. This should be done in an automatic way for the user. For in such a way, it is presented concepts about how such dedicated hardware work. After that, it is shown techniques to map generic problems on the top of these devices, followed by performance evaluations and final conclusions.

Keywords: Dedicated hardware, Vectorial processing, Programming languages.

1 INTRODUÇÃO

Com a crescente demanda por recursos computacionais nos decorrer dos últimos anos, diversas tecnologias de hardware vem sendo aprimoradas. A necessidade de diminuir a carga computacional delegada aos micro-processadores motivou melhorias importantes na sua arquitetura, como o aumento do pipeline de execução, o aumento de memória cache em CPU, e o suporte a trocas de contexto em hardware com o uso de tecnologias recentes como o *Hyper-Threading*(Marr 2002).

A popularização de aplicações que demandam cada vez mais o processamento massivo de matrizes e vetores contribuiu para estes avanços. Simulações científicas e jogos interativos foram grandes responsáveis pela necessidade da criação de extensões especializadas nos micro-processadores convencionais. Alguns destes recursos encontram-se atualmente em extensões como o 3DNow!(Devices 2000) e SSE (*Streaming SIMD Extension*)(Corporation 2003), que expõem ao programador um conjunto de instruções para a manipulação otimizada de vetores e pequenas matrizes.

Os requerimentos exigidos por jogos interativos foram um dos responsáveis por grande parte dos avanços nas tecnologias de hardware. A visualização de geometrias com alta complexidade e com qualidade elevada de sombreamento e efeitos de iluminação passou a ser freqüente nestes aplicativos, levando a uma constante evolução do hardware gráfico. Assim, o pipeline tradicional destes dispositivos foi estendido de forma a diminuir o processamento destas primitivas gráficas pela CPU. A demanda computacional em jogos trouxe a necessidade da adição de processadores programáveis para as unidades de vértice e de fragmento, o que permitiu que muitas operações complexas de ponto flutuante pudessem ser executadas paralelamente com a CPU(Ati 2003).

O mercado de jogos foi responsável também por avanços no hardware de som, embora não de forma tão acentuada como ocorrido com a CPU e com a arquitetura do hardware gráfico. Pela constante aplicação de filtros sobre *streams* de áudio em jogos eletrônicos, processadores de efeitos programáveis(Hoge 1998) foram incorporados às placas de som. Com isso é possível que pequenos programas sejam enviados ao processador de efeitos para atuar sobre as streams de entrada, modificando a saída de áudio, sem ocupar a CPU com esta tarefa(Inc. 1999).

Este trabalho visa definir um modelo de programação para operar matrizes e vetores, onde a execução das operações indicadas pelo usuário são mapeadas para o hardware dedicado disponível. O objetivo desta abordagem é aliviar o processamento feito com instruções comuns da CPU, que fica livre para realizar outras tarefas (Purcell et al. 2002). Esta concepção vem sendo adotada por alguns trabalhos que serviram de base para este, como o uso do hardware gráfico para realizar o cálculo em simulações de fluídos e na solução de álgebra linear(Bajaj et al. 2004). Processadores dedicados tem sido considerados também em aplicações de processamento de sinais, onde são empregados no cálculo

de transformadas de Fourier(Moreland e Angel 2003).

A primeira parte deste trabalho, descrita no Capítulo 2, apresenta conceitos e características de hardwares dedicados. São detalhadas as extensões SSE presentes em microprocessadores, bem como características inerentes ao hardware gráfico e as interfaces disponíveis para operá-los.

O Capítulo 3 apresenta as técnicas adotadas para mapear problemas do domínio proposto (matriciais e vetoriais) para estes dispositivos. São tratadas ainda questões como as abstrações para a manipulação do hardware gráfico e o uso de interfaces de acesso para os recursos SIMD do hardware dedicado.

Em seguida, o Capítulo 4 discute alguns trabalhos existentes e em andamento neste contexto. Este estudo permitirá situar o trabalho proposto no âmbito das soluções de uso de hardware dedicado para finalidades genéricas, como a computação científica.

O Capítulo 5 trata do modelo adotado para acessar e mapear os recursos do hardware dedicado. São apresentados os dispositivos empregados e as técnicas usadas para detectar os recursos dedicados presentes na máquina, enfatizando a interface de programação disponibilizada para o usuário e sua relação com os dispositivos mapeados.

Com a finalidade de mostrar os ganhos de desempenho obtidos com a implementação do produto deste trabalho, o Capítulo 6 mostra algumas operações mapeadas sobre o hardware dedicado e seus ganhos em relação às implementações sem o uso de recursos de hardware especiais.

Finalmente, o Capítulo 7 apresenta conclusões e considerações finais sobre este trabalho.

2 HARDWARES DEDICADOS

Dispositivos de propósito especial são também conhecidos como hardware dedicado. Estes dispositivos são projetados com a finalidade de ajudar a resolver problemas de domínio específico, como é o caso do hardware gráfico. Isto é geralmente concebido através da implementação em hardware de algoritmos até então executados via software, ou ainda através da execução de operações eficientes sobre tipos de dados usados nos problemas que ele deseja minimizar. Recursos semelhantes podem ser encontrados em placas programáveis FPGA e em extensões SIMD encontradas em micro-processadores atuais, especializadas na execução de instruções para acelerar aplicações multimídia.

Recentemente o uso de hardware dedicado tem se mostrado atrativo para o uso em computação genérica. Através da exploração destes dispositivos, é possível expandir sua aplicabilidade para realizar o processamento de problemas genéricos de computação, tipicamente representados em estruturas matriciais e vetoriais. Para isto, é necessário conhecer o funcionamento e técnicas de programação que permitam tirar proveito dos recursos oferecidos pelos hardwares dedicados.

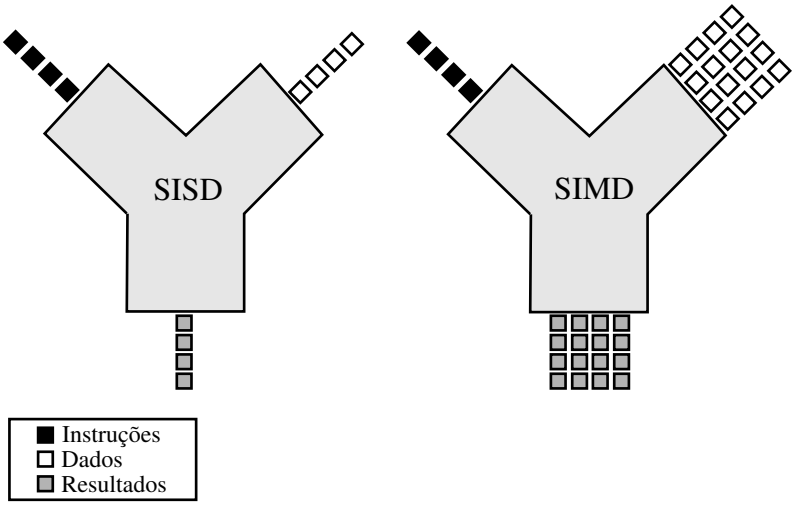
2.1 Os modelos SISD e SIMD em micro-processadores

Os primeiros micro-processadores não tinham a capacidade de operar sobre ponto flutuantes, ficando restritos a operações sobre inteiros. O cálculo sobre ponto flutuantes era realizado em separado, em um hardware dedicado, normalmente sob a forma de um co-processador aritmético. No mundo do x86, este hardware dedicado apareceu com o co-processador x87(Corporation 1994). Desde então, a tecnologia diminuiu o tamanho de transistores, permitindo que eles fossem acoplados na própria CPU, gerando os primeiros micro-processadores com suporte a inteiros e a ponto flutuante. Juntamente com o suporte a registradores de ponto flutuante em hardware, novas instruções foram adicionadas para usufruir destes recursos.

A adição de instruções e hardware SIMD a um micro-processador passa por um processo semelhante, apesar de ser um pouco mais complicado. Em um modo geral, a execução de uma instrução em um processador se dá sobre um único dado escalar por vez. Este modelo de execução é conhecido por SISD (*Single Instruction, Single Data*). Quando um processador é capaz de executar uma instrução sobre mais de um elemento de dado, é dito que este processador suporta o modelo SIMD (*Single Instruction, Multiple Data*). A Figura 2.1 ilustra o funcionamento destes dois modelos de processamento.

O problema de projetar um micro-processador orientado a executar instruções no modelo SIMD é que este modelo não é flexível o bastante para acomodar código de propósito geral, visto que o modelo de programação empregado no SIMD é diferente do SISD. A forma encontrada pelos fabricantes para incluir esta tecnologia em seus micro-

Figura 2.1: Modelos de execução SISD e SIMD



processadores foi através da introdução de extensões, permitindo o uso em conjunto dos dois modelos de execução (Thakkar e Huff 1999). Desta forma, é dada ao compilador a tarefa de determinar as instruções mais adequadas para serem adotadas em um programa (SISD ou SIMD). Na medida do possível o código gerado é otimizado através de instruções SIMD intercaladas às instruções SISD normais do programa, permitindo potenciais ganhos de desempenho na aplicação compilada.

A unidade básica de operação na qual as instruções SIMD trabalham é o vetor, que pode ser composto por registradores para inteiros ou para ponto flutuante. Estes vetores são representados em um formato chamado *packed data* (dado empacotado). Este formato descreve o agrupamento dos dados na forma de bytes (8 bits) ou words (16 bits), por exemplo, que são empacotados em um vetor para ser usado no processamento. Algumas arquiteturas SIMD disponibilizam registradores de ponto flutuante de 128 bits. Desta forma, é possível empacotar 4 componentes de 32 bits (trabalhando com precisão float simples) ou 2 componentes de 64 bits (trabalhando com precisão float dupla). Esta decisão costuma ser tomada de acordo com o objetivo da unidade SIMD projetada: algum hardware dedicado para trabalhar com componentes de cores (como as componentes RGBA) poderia se beneficiar mais do suporte a pacotes de 4x32. Com isto seria evitado que os dados de entrada precisassem ser reformatados para se ajustar em pacotes de 2x64 componentes.

A execução de instruções sobre estes registradores vetoriais se dá através de operações paralelas efetuadas sobre os operandos de entrada, como é ilustrado pela Figura 2.2. Cada componente do vetor X é operado com o componente respectivo no vetor Y (X0 e Y0, X1 e Y1, X2 e Y2, X3 e Y3), salvando o resultado no componente correspondente do vetor Z.

Geralmente as arquiteturas SIMD suportam também operações sobre valores escalares. Neste caso, apenas o primeiro pacote dos vetores de origem são usados para a operação, mantendo os pacotes restantes do primeiro vetor no resultado final. A Figura 2.3 mostra esta operação para um vetor de 128 bits.

Figura 2.2: Operação sobre 4 dados empacotados em um vetor de 128 bits

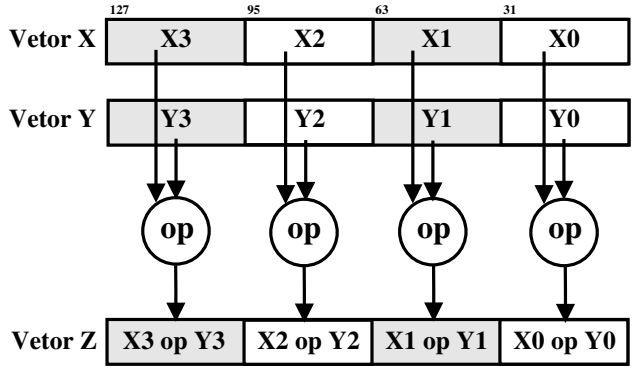
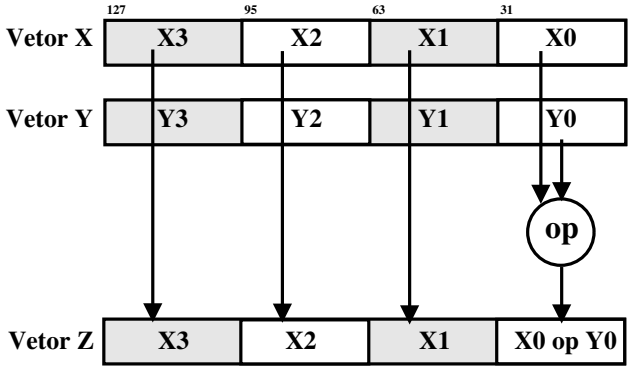


Figura 2.3: Operação escalar em um vetor de 128 bits



2.1.1 MMX

A primeira extensão SIMD para micro-processadores da família x86 veio através do MMX (*MultiMedia eXtensions*). Esta extensão incorporou aos micro-processadores Pentium uma unidade SIMD para inteiros, sem suporte a ponto flutuante. Esta unidade é composta por vetores de 64 bits, armazenados em 8 registradores MMX nomeados de `MM0` a `MM7`. As combinações de pacotes suportadas pelo MMX permitem aproveitar os registradores com diferentes precisões para inteiros, conforme a listagem abaixo:

- *packed byte*, com 8 bytes empacotados em 64 bits;
- *packed word*, com 4 words empacotados em 64 bits;
- *packet doubleword*, com 2 double-words empacotadas em 64 bits;
- *packed quadword*, com um único componente de 64 bits.

O suporte ao MMX na arquitetura x86 trouxe ao conjunto de instruções destes processadores uma nova variedade de instruções para suportar esta extensão. No total foram adicionadas 57 novas instruções, que levam um formato convencional de programação, composto de uma instrução e dois operandos, como no exemplo hipotético:

```
InstruçãoMMX mmreg1, mmreg2
```

Nesta instrução, os registradores `mmreg1` e `mmreg2` são ambos operandos de origem, enquanto o resultado é salvo no próprio `mmreg1`. Isto traz alguns inconvenientes, pois caso deseja-se utilizar o valor de `mmreg1` antes da instrução ter sido executada é necessário copiá-lo para outro registrador.

Alguns detalhes de implementação do MMX ainda impedem a operação simultânea com ponto flutuante nas mesmas rotinas em que estes registradores são usados. Isto ocorre pelo fato de que o espaço de endereçamento do MMX e do x87 (responsável pelas operações em ponto flutuante) é compartilhado, permitindo o uso apenas de um destes em um dado momento (Peleg e Weiser 1996). Esta decisão foi tomada para evitar que o suporte no sistema operacional precisasse ser modificado para salvar o estado dos registradores MMX durante trocas de contexto.

Apesar de não apresentar características interessantes para o uso na computação genérica, como a falta de suporte a unidades de ponto flutuante, o MMX serviu como modelo para a criação de extensões como o SSE, que expandiu este conjunto de instruções com suporte a ponto flutuante em registradores de precisão superior.

2.1.2 SSE

As extensões SSE foram introduzidas na família de micro-processadores Pentium III. As instruções SSE operam em valores empacotados de ponto flutuante de precisão simples, conforme o padrão IEEE (P754 1985), e encontram suporte na arquitetura através da adição de novos registradores especiais. O layout deste tipo de registrador é o mesmo visto nas Figuras 2.2 e 2.3, onde é permitido o armazenamento de 4 valores de ponto flutuante, representando um dado escalar ou vetorial, em um único registrador.

Com a extensão SSE foram adicionados 8 novos registradores de 128 bits de dados na arquitetura do Pentium III, denominados `XMM`, acessíveis pelos nomes `XMM0-XMM7`. Entretanto, diferente do espaço de endereçamento do MMX, os registradores `XMM` não compartilham o mesmo espaço com o x87, permitindo o uso conjunto de instruções SSE com MMX ou da FPU do x87. Para o suporte à instruções SIMD sobre inteiros, o SSE utiliza-se dos registradores MMX, que foram mantidos nas novas arquiteturas de micro-processadores.

Para suportar esta nova extensão SIMD, foram adicionadas duas unidades de hardware dedicado à CPU. O Pentium III apresenta duas unidades independentes de precisão simples para ponto flutuante; uma opera sobre instruções de multiplicação SIMD, através da expansão do hardware de multiplicação de ponto flutuante já existente, e outra para realizar somas sobre pacotes de dados (Corporation 2004).

Estas novas unidades de processamento independentes adicionaram ao IA-32 um novo estado de execução. Isto implica que o sistema operacional tenha conhecimento dos recursos oferecidos pelo processador, de forma que o estado de execução do SSE seja salvo em trocas de contexto de processos no kernel, da mesma forma que é realizado para salvar o estado da FPU do x87 e do MMX.

As instruções suportadas no SSE são sub-divididas nos seguintes grupos:

- Instruções de movimento de dados
- Instruções aritméticas
- Instruções lógicas
- Instruções de comparação

- Instruções de *shuffle* (combinação de 2 registradores em 1 único)
- Instruções de conversão de dados

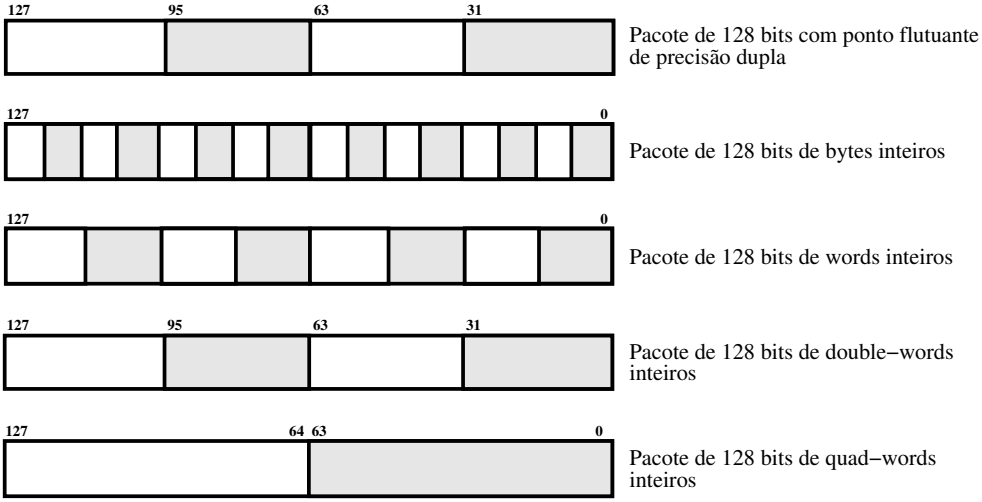
As instruções aritméticas permitem realizar operações como soma, subtração, multiplicação, divisão, raiz quadrada, máximo e mínimo, entre outras, sobre dados empacotados ou escalares. Nota-se que o uso delas apresenta a mesma característica vista no MMX: como o registrador de destino também armazena um dos valores de entrada para a operação, seu valor deve ser copiado para outro registrador caso ele precise ser usado no futuro. A interface disponibilizada para acessá-las é semelhante à do MMX:

```
InstruçãoSSE xmmreg1, xmmreg2
```

2.1.3 SSE2

O SSE2 (*Streaming SIMD Extensions 2*) foi introduzido na arquitetura IA-32 pelos processadores Pentium 4 e pelo Intel Xeon. Ele estende as tecnologias MMX e SSE através do suporte a pacotes de precisão dupla de ponto flutuante e de pacotes de valores inteiros sobre registradores de 128 bits. Os tipos de dados suportados pelas extensões SSE2 são apresentados na Figura 2.4.

Figura 2.4: Tipos de dados suportados pelo SSE2



Estas novas características do SSE2 deram à arquitetura IA-32 a possibilidade de realizar operações SIMD em pares de pacotes de ponto flutuante de precisão dupla. Isto permite que precisões maiores na computação sejam alcançadas com o uso dos registradores `XMM`, que aumenta o desempenho do processador em aplicações científicas e em aplicações que usam técnicas avançadas de geometria 3D, como o *ray tracing* (Amanatides 1987). Outras aplicações tomam vantagem do modelo de processamento SIMD, porém sobre os valores inteiros empacotados de 128 bits fornecidos pelos registradores `XMM`. Tais aplicações incluem a autenticação RSA e a criptografia RC5, que ganham bastante flexibilidade e vasão ao processarem operações sobre pacotes de valores inteiros.

2.1.4 Acesso aos recursos SIMD em micro-processadores

Nota-se que enquanto o acesso à instruções específicas do MMX, SSE e do SSE2 é realizado a partir de uma linguagem *assembly*, muitos compiladores são capazes de gerar

código otimizado para este modelo de execução. No entanto nem sempre o resultado é satisfatório, devido à complexidade de análise dos algoritmos em alguns casos. Isto faz com que muitas vezes estas extensões deixem de ser usadas, ou sejam usadas ineficientemente.

Para superar este problema, é possível usar os recursos SIMD do processador através da inclusão de código assembly diretamente em rotinas específicas. Este tipo de construção costuma ser suportado em linguagens como C, C++ e Pascal, que na própria construção da linguagem já prevêem o uso destas extensões. Este tipo de extensão possibilita o encapsulamento de todo o código *baixo nível* necessário em funções individuais, disponibilizando para o usuário apenas a API de acesso a elas.

Geralmente a inclusão de código assembly na linguagem requer que o programador especifique os registradores que deseja utilizar e as posições de memória em que os dados envolvidos nas instruções se encontram. Este requisito é aliviado com o uso de extensões estendidas suportadas por alguns compiladores, como o GCC. Nele, a interface de declaração de código assembly permite especificar os operandos da instrução usando expressões em C, sem que o usuário se preocupe com a alocação de registradores e com os endereços de memória em que se encontram os dados (Foundation 2002).

Um exemplo deste tipo de construção seguindo as extensões do GCC pode ser visto abaixo. Ele calcula a função seno com uma rotina em assembly para um valor fornecido pelo usuário, que tem acesso somente ao protótipo da função `calcula_seno()`:

```
1 float
2 calcula_seno(float valor)
3 {
4     float resultado;
5
6     asm("fsin" : "=t" (resultado) : "0" (valor));
7     return resultado;
8 }
```

Uma rotina para encapsular operações SIMD pode ser criada com base nas extensões do GCC. A configuração dos valores de entrada dos registradores `XMM` é feita através de uma operação de movimentação de dados, que toma como origem um array de valores com a precisão desejada. Para usufruir dos valores de ponto flutuante de precisão simples, basta configurar dois arrays de entrada com 4 posições float, bem como outro para armazenar o resultado. Uma interface de acesso pode ser definida em C, abstraindo os detalhes de implementação:

```
1  void
2  soma_simd(float operando1[4], float operando2[4], float resultado[4])
3  {
4          asm(
5                  "movups (%%edx), %%xmm0\n"
6                  "addps  (%%eax), %%xmm0\n"
7                  "movups %%xmm0, (%%ecx)\n"
8                   :
9                   : "a" (operando1), "d" (operando2), "c" (resultado)
10     );
11 }
```

2.2 Hardware gráfico

O poder computacional presente no hardware gráfico cresceu de forma significativa nos últimos anos. Junto a este crescimento, funcionalidades internas do hardware gráfico foram gradativamente sendo expostas ao programador, como a configuração e mais tarde a programação de algumas unidades de processamento. Além disso, com a introdução de dados de ponto flutuante, a precisão das GPUs se aproximou da precisão suportada nas CPUs. Isto fez com que algumas aplicações fora do domínio de computação gráfica pudessem aos poucos serem mapeadas e executadas no hardware gráfico, beneficiando-se da capacidade de processamento paralelo em relação à CPU.

Devido ao fato destes dispositivos não terem apresentado características programáveis até recentemente, poucos trabalhos de abstração ao acesso a eles estão disponíveis, salvo as APIs de programação de primitivas gráficas. Tendo em vista o mapeamento de algoritmos para o hardware gráfico, o acesso facilitado aos seus recursos de processamento é bastante importante, pois evita abstrações complicadas e mapeamentos tediosos de algoritmos para este hardware.

Esta seção apresenta as características trazidas ao hardware gráfico nos últimos anos. São apresentados o funcionamento do pipeline gráfico tradicional e do pipeline gráfico programável atual, bem como métodos de acesso aos processadores de vértices e fragmentos e o modelo de execução SIMD por eles adotado.

2.2.1 O pipeline gráfico tradicional

Fundamentalmente, o hardware gráfico moderno é quebrado em dois estágios: o de geometria e o de rasterização. Atualmente a geometria é representada como uma coleção de triângulos e definida por um conjunto de vértices, fornecido por uma aplicação do usuário. Estes vértices são transformados por algum processo, sendo em seguida rasterizados em fragmentos, ou seja, processados para uma representação em 2D. Os fragmentos, por sua vez, passam por outra etapa de transformação, e finalmente são gerados os valores de saída da computação. Os valores de saída geralmente são combinados com os valores já existentes no *frame buffer*, que é o buffer que armazena o conteúdo de vídeo que será exibido pelo hardware gráfico.

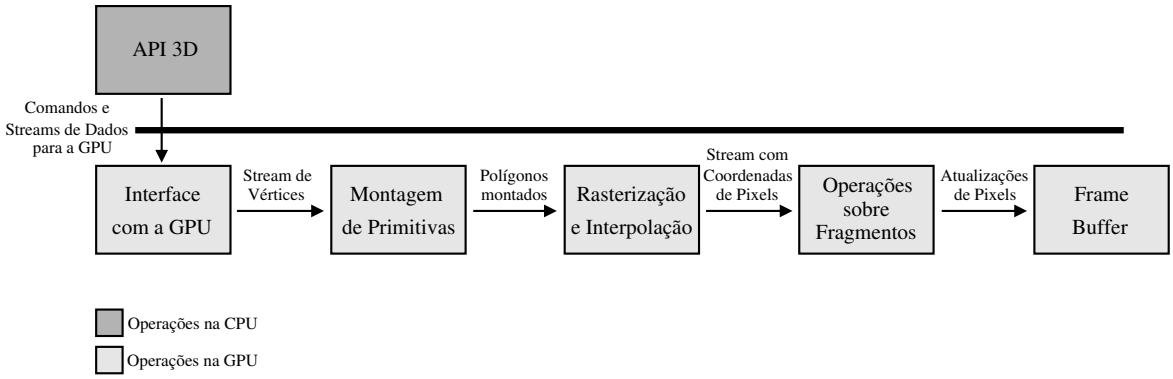
Estes dois estágios fundamentais fazem parte de um processo maior, denominado *pipeline gráfico*, ilustrado na Figura 2.5. Este pipeline inicia a partir do momento em que o usuário fornece a geometria para o hardware gráfico. No ponto em que a geometria é recebida pelo processador de vértices, ele encarregada-se de realizar uma série de transformações no sistema de coordenadas.

A geometria é inicialmente transformada do seu sistema de coordenadas próprio do objeto para o sistema de coordenadas de um *mundo virtual* comum, onde todos os outros objetos, fontes de luz e posições de câmera se encontram. Esta transformação é realizada para que todos os objetos sejam mantidos em uma mesma cena.

Este processo é seguido por outra transformação no sistema de coordenadas. A posição de visualização da cena é movido para a origem (X=0 e Y=0), e o plano de visualização é posicionado nas coordenadas de câmera fornecidas pelo usuário. Esta transformação muda as coordenadas do *mundo virtual* para as coordenadas da câmera, também chamada de *eye space*.

Após as transformações nos sistemas de coordenadas terem sido feitas, os pontos recebidos pelo processador de vértices são repassados para o rasterizador, em um sistema de coordenadas do dispositivo. Este sistema mantém os pontos dentro de um cubo, onde

Figura 2.5: O pipeline gráfico



as coordenadas $(x, y) \in [-1, 1]$ e $z \in [0, 1]$. Este cubo é usado para efetuar o recorte da primitiva, eliminando a necessidade de processar informações que não serão visíveis na imagem final.

Alguns outros recortes especiais também são feitos com este cubo, como o *stencil*, onde o recorte é feito baseado em uma máscara de bits armazenada em um buffer, o *alpha*, onde elementos com a componente `A` do canal `RGBA` com o valor 0 são removidas, e o *scissor*, que remove segmentos da imagem que estejam fora das coordenadas do dispositivo (Foley et al. 1996).

Os fragmentos gerados que passaram por estes testes são enviados para o processador de fragmentos. Também um conjunto de atributos pode ter sido calculado anteriormente no processador de vértices e passado para o rasterizador. Estes atributos são linearmente interpolados pelo rasterizador e passados para o processador de fragmentos, caso não tenham sido recortados. O processador de fragmentos então recebe o conjunto de atributos e efetua demais cálculos, retornando uma tupla de 4 componentes.

Com isto, finalmente as primitivas são convertidas para pixels, que são combinados no frame buffer e mostrados no display.

2.2.2 O pipeline gráfico atual

Arquiteturas de hardware gráfico recentes passaram a migrar do modelo tradicional de pipelines de funções fixas para enfatizar a versatilidade, expondo ao programador módulos do pipeline que antes eram somente configuráveis. Esta flexibilidade é provida através de maneiras de reprogramar os processadores envolvidos neste pipeline com rotinas escritas pelo usuário, denominadas *vertex* e *fragment shaders*, dependendo do processador para o qual são escritas.

A possibilidade de reprogramar o pipeline gráfico tornou possível a interpretação do hardware gráfico como um *processador de streams*, tal como as extensões SIMD presentes em micro-processadores. Este design consiste em uma stream de elementos de dados e de um *kernel* (programa) que opera nos elementos desta stream. Enquanto uma CPU tem acesso livre à memória e liberdade para ler e escrever instruções e dados, um processador de streams é apenas capaz de visualizar o elemento atual de processamento. Após carregado no processador de streams, o *kernel* processa todos os dados recebidos pelo processador, até que um novo *kernel* seja carregado (Kapasi et al. 2003).

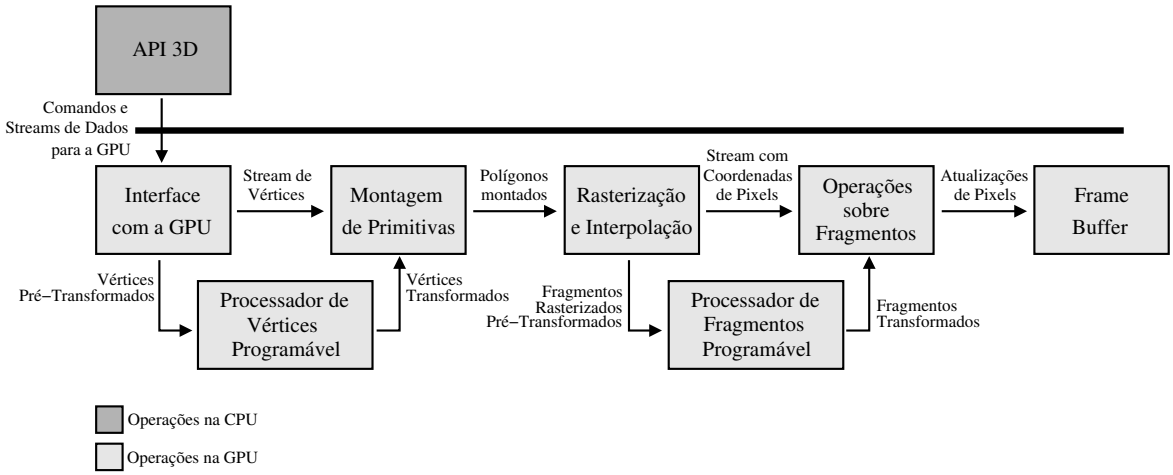
Este modelo de execução foi introduzido nas GPUs através da exposição dos proces-

sadores de vértice e de fragmento programáveis. Isto foi realizado através da intervenção no pipeline gráfico tradicional, com a especialização dos estágios de processamento de geometria e de rasterização. O processamento de geometria, que até então consistia em uma etapa de função fixa, permite atualmente que programas escritos pelo usuário sejam fornecidos e executados sobre cada geometria enviada a este processador. Com isso o usuário tem liberdade para indicar como será realizado o sombreamento através de algoritmos próprios.

Da mesma forma, o processo de rasterização passou a permitir a execução de programas para substituir a função fixa tradicional. Com programas escritos especialmente para o processador de fragmentos, o usuário é capaz de modificar o processo de geração de texturas, cor e *fog*, afetando diretamente a rasterização de pontos, linhas, polígonos e bitmaps.

Estas modificações significativas permitiram que diferentes algoritmos passassem a ser implementados diretamente no hardware gráfico, liberando ciclos importantes da CPU. Isto alterou a visão tradicional do layout do pipeline gráfico, que passou a ser representado como na Figura 2.6[1].

Figura 2.6: O pipeline gráfico programável atual



2.2.3 Processadores de vértice e de fragmento

Cabe ao processador de vértices realizar as modificações sobre o conjunto de primitivas gráficas fornecidas pelo usuário (tipicamente triângulos ou quadriláteros), especificados como um conjunto de vértices na aplicação. O processador de fragmentos realiza operações matemáticas especiais sobre as texturas envolvidas na computação, usando dados gerados pelo processo de rasterização.

Estas modificações e operações, descritas em uma linguagem assembly específica dos processadores de vértice e de fragmento, são armazenadas em um array de bytes e enviadas à GPU através de alguma API de acesso ao hardware gráfico. Em seguida, uma função específica da API usada pode ser invocada para informar a GPU de que o processo de geração de geometria será realizado pelas instruções contidas neste programa, e não pela função fixa tradicional. O mesmo pode ser feito para informá-la de que a geração

[1]Figuras 2.6 e 2.5 derivadas do artigo (Comba et al. 2003)

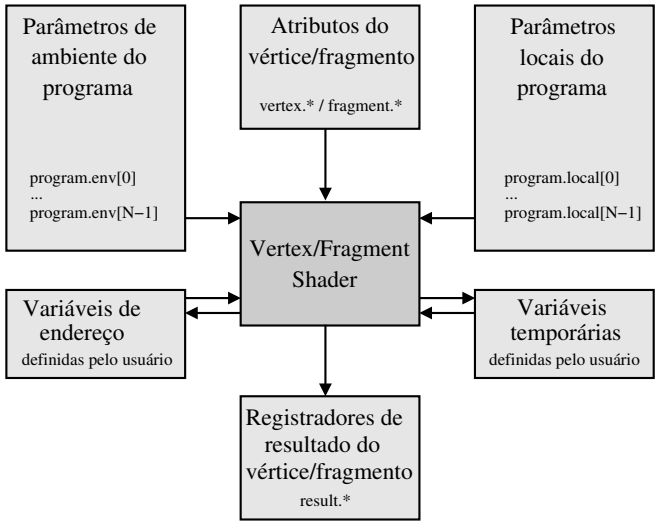
de cores dos pixels finais será calculado pelo programa de fragmentos especificado pelo usuário, através de outra função exportada pela API.

A arquitetura do hardware gráfico de hoje permite que estes programas acessem apenas o elemento *atual* de processamento. Não existem informações a respeito dos vértices ou fragmentos já processados e dos próximos a serem processados, caracterizando esta arquitetura como uma composição de processadores de streams. Esta característica traz uma grande vantagem à GPU, pois ela pode processar elementos independentemente uns dos outros, alcançando um grau de paralelismo no pipeline capaz de aumentar o desempenho da operação como um todo (1).

Além do dado de entrada, um programa descrito pelo usuário pode contar com parâmetros externos fixos, como valores constantes de ambiente locais ou globais, compartilhados por todos os outros programas. É permitido também que o programa faça uso de variáveis temporárias, de endereço e defina "apelidos" (*aliases*) para outras variáveis.

Após a execução do programa específico, a saída é gerada e usada pela próxima etapa do pipeline gráfico. O layout da arquitetura dos processadores de vértice e de fragmento é descrito na Figura 2.7. As setas indicam a direção na qual os dados são fornecidos ao programa (leitura ou escrita), representado no centro da imagem.

Figura 2.7: A arquitetura dos processadores de vértice e de fragmento



Cada registrador presente nos processadores de vértice e de fragmento é composto por um vetor de 128 bits, capaz de empacotar 4 componentes de ponto flutuante com precisão de 32 bits. Estes componentes representam as coordenadas `X`, `Y`, `Z` e `W` dos vértices, ou os canais `R`, `G`, `B` e `A` de cores, que podem ser acessados individualmente pelos programas. Isto permite que operações sejam feitas usando componentes específicas de cada registrador. A atribuição da soma dos canais `R` de um registrador `R0` e `G` de um registrador `R1` ao canal `B` de um registrador `R2` pode ser representada pela operação $R2.b = R0.r + R1.g$.

A representação dos dados na forma de pacotes permite que as GPUs modernas realizem uma única instrução sobre todos os dados empacotados. Isto significa que, no caso da representação de operandos com pacotes de 4 componentes, a execução de uma única instrução sobre duas tuplas é aplicada sobre estes 4 componentes simultaneamente, da mesma forma que as extensões SIMD presentes em micro-processadores. Por exemplo, a

operação:

$$\text{R2} \quad = \text{R0.rgab} + \text{R1.rrgg} \tag{2.1}$$

pode ser decomposta nas sub-operações:

$$\begin{aligned} \text{R2.r} &= \text{R0.r} + \text{R1.r} \\ \text{R2.g} &= \text{R0.g} + \text{R1.r} \\ \text{R2.b} &= \text{R0.a} + \text{R1.g} \\ \text{R2.a} &= \text{R0.b} + \text{R1.g} \end{aligned} \tag{2.2}$$

Com isso, a Operação 2.1 leva $1/4$ do tempo necessário para a computação das operações individuais 2.2. Nota-se que na Operação 2.1 os componentes do registrador `R0` estão fora de ordem – `rgab`, quando os valores são armazenados no registrador como componentes na seqüência `rgba`; esta operação é chamada de *swizzling*. O registrador `R1`, por sua vez, informa de forma replicada as componentes a serem usadas na operação; esta operação é chamada de *smearing*. Estas operações são realizadas na GPU sem perda de performance, em contraste com a extensão SSE, onde a movimentação de dados entre diferentes canais requer instruções adicionais.

Além de operações como a soma, estes processadores são capazes de realizar operações computacionalmente pesadas, como:

- Produto vetorial;
- Multiplicação;
- Multiplicação e soma;
- Raiz quadrada;
- Exponencial;
- Logaritmo.

Isto torna o uso da GPU atraente para a computação de problemas que não sejam do domínio de computação gráfica, visto o suporte a operações comuns em problemas genéricos de computação.

2.2.4 Interfaces de programação

Extensões SIMD como o MMX e a família SSE podem ser programadas com instruções assembly diretamente na linguagem. Isto não ocorre com as GPUs, devido ao fato dos fabricantes não não divulgarem informações importantes como o enderaçamento de registradores e o layout da memória de seus hardwares gráficos. Desta forma, o acesso aos recursos das GPUs é feito através de bibliotecas gráficas como a OpenGL (*Open Graphics Library*) e o DirectX, da Microsoft. Com isto é possível que os fabricantes distribuam módulos pré-compilados destas bibliotecas para o acesso ao seu hardware, que são podem ser acessados através de uma API de conhecimento público.

A interface de acesso através do OpenGL é feita através de várias extensões especificadas por um consórcio chamado ARB (*Architecture Review Board*), formado em 1992. No contexto de programação dos processadores de vértice e de fragmento, estas extensões definem métodos que devem ser suportados na implementação do OpenGL fornecida pelos fabricantes. A medida em que estas extensões vão sendo adotadas por diferentes fabricantes e usuários, elas são incorporadas à especificação do OpenGL. Isto garante a

portabilidade na programação de GPUs, uma vez que tanto as interfaces de acesso quanto os mnemônicos e as instruções usadas nos *shaders* são regidas por uma especificação.

Extensões como a `ARB_vertex_program` e a `ARB_fragment_program` provêem um mecanismo explícito para definir seqüências de instruções de vértices e fragmentos, respectivamente. Para descrever os programas (*shaders*), elas definem um modelo de programação que inclui suporte a um conjunto de instruções orientado a vetores de 4 componentes de ponto flutuante e a um número relativamente grande de registradores deste tipo. Fabricantes de hardware gráfico freqüentemente estendem estas extensões para suportar recursos específicos de suas GPUs, como as extensões `NV_vertex_program` e a `NV_fragment_program` fornecidas pela NVIDIA (Corporation 2003).

As extensões ARB para os *vertex* e *fragment shaders* adicionam à API do OpenGL procedimentos e funções para compilar, enviar e remover estes programas dos processadores específicos da GPU. Funções extras para habilitar a execução do *shader* no pipeline gráfico também fazem parte deste conjunto. De posse de uma biblioteca OpenGL com suporte a estas extensões, o usuário é capaz de descrever seus programas através de strings na própria linguagem de programação. O seguinte exemplo descreve o conteúdo de uma string para realizar a soma dos 4 componentes (RGBA ou XYZW) de duas texturas:

```
1 !!ARBfp1.0
2 TEMP textura0, textura1 ;
3 TXB textura0, fragment.texcoord [0], texture [0], RECT;
4 TXB textura1, fragment.texcoord [0], texture [1], RECT;
5 ADD result.color , textura0 , textura1 ;
6 END
```

A primeira linha indica que este programa está descrito conforme a extensão ARB 1.0 para o processador de fragmentos. Variáveis temporárias, descritas com a palavra reservada TEMP, são usadas para armazenar o conteúdo das texturas 0 e 1, obtido a partir das coordenadas atuais da textura 0. Após obtidos os valores dos componentes nestas texturas, a cor final do pixel é calculado com a operação de adição sobre estas variáveis temporárias, que por padrão age sobre todos os 4 componentes do registrador vetorial. O programa é finalizado com a palavra reservada END, e será invocado novamente para cada fragmento restante no pipeline.

O resultado da operação pode ser buscado através da leitura do conteúdo do frame buffer, que irá conter os pixels com o resultado das operações realizadas no decorrer de todo o processamento do pipeline. Estas operações são realizadas através de outros procedimentos e funções fornecidos pela API usada.

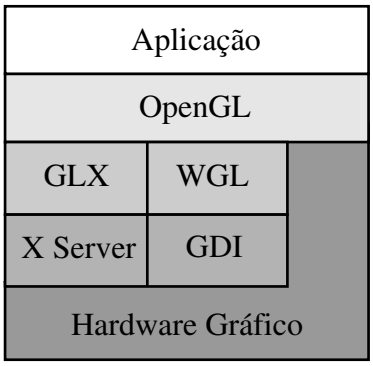
2.2.5 Relacionamento com o sistema de janelas

O OpenGL é definido para ser um *sistema de janelas independente*. Isto significa que as rotinas básicas do OpenGL não descrevem como criar um *contexto OpenGL* para uma aplicação ou como criar uma janela para usar na renderização do OpenGL. É função do sistema de janelas cuidar destes detalhes, como o *X Window System* em sistemas UNIX ou o GDI (*Graphical Display Interface*) no Windows.

Assim, é necessário uma camada para *unir* o sistema de janelas e o OpenGL. No X Window System, este papel é realizado pelo GLX, que une o *X Server* ao OpenGL. Em plataformas Windows este protocolo de comunicação é implementado através do WGL. A Figura 2.8 mostra esta relação desde a camada da aplicação até o hardware gráfico.

Esta característica traz algumas conseqüências para a programação do hardware grá-

Figura 2.8: Camadas de união entre o OpenGL e o hardware gráfico



fico: a medida em que extensões OpenGL são aprovadas pelo ARB, elas precisam ser adaptadas para a implementação em cada uma destas *camadas de união*. Essa adaptação as vezes pode ser muito custosa operacionalmente, atrasando a disponibilidade de recursos de desenvolvimento em plataformas específicas.

3 MAPEAMENTO DE PROBLEMAS SOBRE O HARDWARE DEDICADO

Após conhecidas as características das extensões SIMD de micro-processadores e de GPUs, faz-se necessário abstrair suas peculiaridades para que seja possível usá-las para a computação de problemas genéricos. Apesar disto não ser um problema de difícil abordagem para a execução de operações sobre o MMX/SSE/SSE2, que podem ser programados com rotinas em assembly no próprio código, isto não é verdade para o hardware gráfico, que precisa ter as entradas do usuário mapeadas de alguma forma para os tipos de dados suportados pelo pipeline gráfico e pelas APIs de programação.

3.1 Abstração do hardware gráfico

Para realizar algum mapeamento sobre o hardware gráfico, é necessário que os dados fornecidos pelo usuário sejam mapeados para alguma primitiva suportada pela GPU, que trabalha com dois tipos básicos de dados: mapas de geometria e de textura. No primeiro caso, os dados do usuário devem ser mapeados para malhas triangulares ou poligonais, que são os tipos processados pelo processador de vértices. Este tipo de dados exige um espaço de armazenamento grande para as informações, pois são necessárias propriedades extras para descrever cada vértice, como o vetor normal e coordenadas extras de texturas (Brown 2002).

A representação dos dados através de mapas de textura permite um mapeamento direto de cada elemento de entrada como elementos de uma textura. Isto traz uma vantagem, pois é possível realizar operações no processador de fragmentos envolvendo mais de uma textura e renderizar o resultado em uma nova textura, que pode ser usada como entrada para outras operações (Poddar e Womack 2001).

Apesar da extensão OpenGL que possibilita este tipo de renderização estar disponível apenas para a plataforma Windows atualmente, o fato de permitir um mapeamento direto em uma representação mais compacta faz com que o uso do processador de fragmentos seja ideal para mapear os problemas sobre o hardware gráfico. Por esta extensão não estar disponível no GLX, a técnica adotada para a renderização é a de enviar os dados para o frame buffer, e recuperá-los quando o resultado for necessário.

Desta forma, os dados de entrada são mapeados sobre texturas, onde um código específico é aplicado e elas. O resultado desta renderização é enviado para o frame buffer, que pode ter seus valores obtidos através de funções disponíveis na API do OpenGL para a leitura de pixels, que os armazenam em um array de dados.

Entretanto, por se tratar de um hardware especializado, o frame buffer não apresenta suporte para a representação de valores em ponto flutuante nas placas gráficas atuais. Isto

não é um problema para aplicações gráficas, que o utilizam com o fim de armazenar a cor do pixels finais. Para elas, basta um valor entre 0 e 1 para ser capaz de representar o valor de cada componente de cor. Por outro lado, isto traz um problema para a computação de problemas genéricos, pois resultados que ultrapassem estes limites são truncados. Por exemplo:

$$\begin{aligned} 0.3 + 0.4 &= 0.7 \\ 0.3 + 0.5 &= 0.8 \\ 0.3 + 0.6 &= 0.9 \\ 0.3 + 0.7 &= 1.0 \\ 0.3 + 0.8 &= 1.0 \\ 0.3 + 0.9 &= 1.0 \\ 0.3 + 1.0 &= 1.0 \\ 0.3 + 1.1 &= 1.0 \\ &... \end{aligned} \tag{3.1}$$

A solução para este problema é a utilização de frame buffers *off-screen*, onde seja possível armazenar os valores em uma representação de ponto flutuante, em uma área de memória não visível como o frame buffer. Isto é obtido com um recurso chamado *Pixel Buffer*, ou *pbuffer*, suportado tanto no GLX quanto no WGL (Oat 2002).

3.2 Mapeamento de operações

Os problemas de mapeamento dividem-se em três classes: as operações escalares, vetoriais e matriciais. Operações escalares têm como objeto de trabalho unidades singulares de dados, como números inteiros ou de ponto flutuante. Operações vetoriais envolvem algum destes dois tipos de dados, porém de forma agrupada para representar vetores. Finalmente, operações matriciais envolvem o cálculo destes tipos de dados representados sobre a forma de matrizes.

3.2.1 Operações vetoriais

Pela natureza dos tipos de dados suportados nas extensões SIMD e na GPU, operações sobre vetores são mapeáveis para estes dispositivos de forma direta. De posse de um *kernel* (função exclusiva para execução de instruções SIMD) que realize operações sobre vetores de entrada e salve o resultado em um vetor de saída, o uso de uma estrutura de dados como um array de vetores torna-se ideal. Desta forma, os valores de entrada do *kernel* podem ser enviados através de um loop, encarregado de percorrer todos os vetores deste array. Esta situação é bastante semelhante com a demonstrada pela função `soma_simd()`, apresentada na Seção 2.1.4, e pode ser usada para mapear vetores sobre o SSE/SSE2.

Figura 3.1: Dados de entrada usados em operações vetoriais

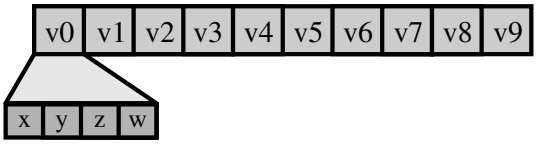


Figura 3.2: Vetores de dados mapeados em uma matriz de textura

	0	1	2	3	4	5	6	7	8	9
x	$v0_x$	$v1_x$	$v2_x$	$v3_x$	$v4_x$	$v5_x$	$v6_x$	$v7_x$	$v8_x$	$v9_x$
y	$v0_y$	$v1_y$	$v2_y$	$v3_y$	$v4_y$	$v5_y$	$v6_y$	$v7_y$	$v8_y$	$v9_y$
z	$v0_z$	$v1_z$	$v2_z$	$v3_z$	$v4_z$	$v5_z$	$v6_z$	$v7_z$	$v8_z$	$v9_z$
w	$v0_w$	$v1_w$	$v2_w$	$v3_w$	$v4_w$	$v5_w$	$v6_w$	$v7_w$	$v8_w$	$v9_w$

A maneira mais simples de mapear vetores sobre o hardware gráfico é através do uso de texturas uni-dimensionais, conforme a representação de um vetor, que é uma matriz de uma dimensão. Esta abordagem, apesar de simples, não é interessante por dois motivos. O primeiro deles trata da questão da memória disponível para texturas na GPU, que é limitada em seu número de linhas e colunas pela implementação do hardware. Desta forma, o tamanho máximo que uma matriz de textura uni-dimensional pode alcançar é o próprio limite no número de colunas imposto pelo hardware gráfico. O segundo motivo vem do fato de que texturas 2D são renderizadas com maior eficiência pelo hardware gráfico (Comba et al. 2003).

Considerando estas observações, esta operação pode ser executada sobre o hardware gráfico com o uso de uma técnica derivada da representação de vetores em texturas uni-dimensionais. Assim, os N arrays de dados vetoriais podem ser armazenados cada um em uma linha diferente desta matriz, aproveitando a capacidade de armazenamento 2D das matrizes de textura.

A Figura 3.1 mostra uma destas linhas onde um array de vetores é armazenado, representado por elementos de 4 componentes `X`, `Y`, `Z` e `W`. Cada um destes elementos é mapeado diretamente para uma posição na matriz de textura, composta de 4 componentes de 32 bits de ponto flutuante, como indicado na Figura 3.2.

Os primeiros operandos são passados para a GPU com o envio da matriz de textura onde estes dados foram mapeados. A operação repete-se para enviar os operandos que serão computados com estes primeiros, que são armazenados em uma segunda matriz de texturas, e também enviados ao hardware gráfico.

Após o envio das matrizes para a GPU, as rotinas SIMD são carregadas no processador de fragmentos, que se responsabilizará pelo processamento dos fragmentos. No entanto, por ser um hardware de domínio específico, o processamento não é efetuado caso não haja ainda uma geometria definida pela aplicação. Do contrário, a GPU seria desperdiçada realizando cálculos sobre texturas que não seriam aplicadas a nenhum objeto.

Como as operações são realizadas sobre texturas de duas dimensões, a geometria mais simples para esta situação é o quadrilátero, que deve ter as mesmas dimensões da matriz de textura usada para mapear os vetores de entrada. Após a definição da textura o processamento é realizado, e o resultado da operação pode ser obtido com a leitura da imagem salva no frame buffer (neste caso, o *pbuffer*). A imagem é recuperada em uma estrutura de dados do mesmo tipo usado para encapsular os vetores de entrada, e desta forma o resultado pode ser retornado imediatamente para ser usado na aplicação.

3.2.2 Operações escalares

O mapeamento de operações escalares é feito de forma semelhante ao de operações vetoriais. Neste caso,basta agrupar os dados de entrada em seqüência, como indicado na Figura 3.3, e realizar a operação sobre estas streams de dados. Como apresentado na Figura 2.3, existem instruções específicas para operarem sobre valores escalares: elas realizam a operação sobre o primeiro componente do vetor e replicam os dados das componentes restantes para as componentes respectivas no registrador de resultado.

Com isso, faz-se necessário realizar n instruções, onde n é o número de operações matemáticas envolvidas. Devido a isso, operações sobre valores escalares levam 4 vezes mais tempo para serem calculadas em relação à operações sobre vetores, pois nesta última a representação natural dos dados já utiliza os 4 componentes dos registradores vetoriais.

Figura 3.3: Array de dados escalares

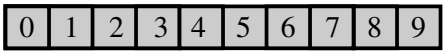
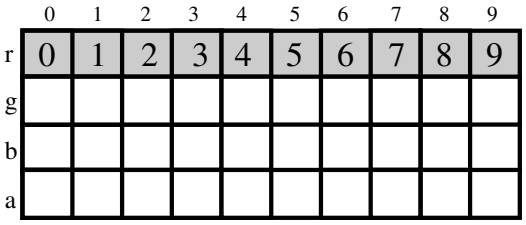


Figura 3.4: Dados escalares mapeados em uma matriz de textura



Apesar de ser possível mapear valores escalares para os 4 componentes dos registradores, isto acarreta em um tempo de processamento extra em CPU para fazer a conversão. Assim, o mapeamento é feito sem conversões, conforme indicado na Figura 3.4. Nota-se que, diferente da Figura 3.2, a matriz de textura apresenta seus componentes através da nomenclatura `RGBA`. Ambas nomenclaturas são aceitas nas linguagens dos processadores de pixel e de fragmento, desde que não sejam misturadas componentes de uma nomenclatura com as da outra, como `RGBW` ou `XYZA`.

O processamento em si é realizado da mesma forma que o processamento de operações vetoriais, e o resultado, que virá na forma de 4 componentes, terá valores significativos apenas na primeira componente `RGBA`.

Nas extensões SSE, o mapeamento desta operação é feito da mesma forma que nas operações sobre vetores, porém usando instruções específicas para escalares. Isto evita que o processador perca ciclos computando valores para as componentes não utilizadas, e então as componentes não usadas são simplesmente replicadas no registrador de destino, conforme demonstrado na Figura 2.3.

3.2.3 Operações matriciais

3.2.3.1 Adição e subtração

Operações como adição e subtração de matrizes são feitas sobre elementos de mesma posição nas matrizes de origem, que pode ser representada pela operação

$$C_{ij} \quad = A_{ij} \; op \; B_{ij} \tag{3.2}$$

A exemplo da adição de duas matrizes A e B de mesma dimensão, definida por $C = A + B$, denota-se a operação de suas somas por:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{a22} \end{bmatrix} + \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11} + B_{11} & A_{12} + B_{12} \\ A_{21} + B_{21} & A_{22} + B_{22} \end{bmatrix} \tag{3.3}$$

Esta operação assemelha-se à técnica usada para manipular vetores, onde a operação é aplicada sobre elementos correspondentes nos dados de origem. A diferença estará basicamente na forma de armazenamento da estrutura para representar as matrizes em software, conforme explicado a seguir.

Tanto as extensões SSE/SSE2 quanto a GPU trabalham sobre registradores de 4 componentes. Para tirar proveito desta característica, pode-se representar uma matriz de forma que seus elementos sejam armazenados de 4 em 4 dentro destes registradores. Isto permitirá que até 4 operações sejam feitas com uma única instrução, acelerando o processo de computação.

Por fins de simplicidade de representação, os elementos escolhidos para serem agrupados foram os das colunas. Uma matriz de 30×8 (linhas $\times$ colunas), neste caso, será alocada internamente em 8 linhas e 8 colunas apenas. Ou seja, a cada 4 linhas de uma matriz M, os elementos $M_{i,j}, M_{i+1,j}, M_{i+2,j}$ e $M_{i+3,j}$ de uma coluna j serão agrupados em um único registrador vetorial.

Este método de armazenamento é aproveitado diretamente para operações com a GPU. Pelo fato dos dados estarem no mesmo formato usado por matrizes de textura, não é necessário fazer conversão de dados para fornecê-los ao processador de fragmentos.

O mesmo pode ser feito para os registradores `XMM` do SSE, que podem armazenar valores de ponto flutuante em cada um de seus 4 componentes. Após ter os dados de entrada do usuário armazenados em uma matriz compacta, com elementos também de 4 componentes, o cálculo pode ser efetuado sobre cada par de dados das matrizes de origem.

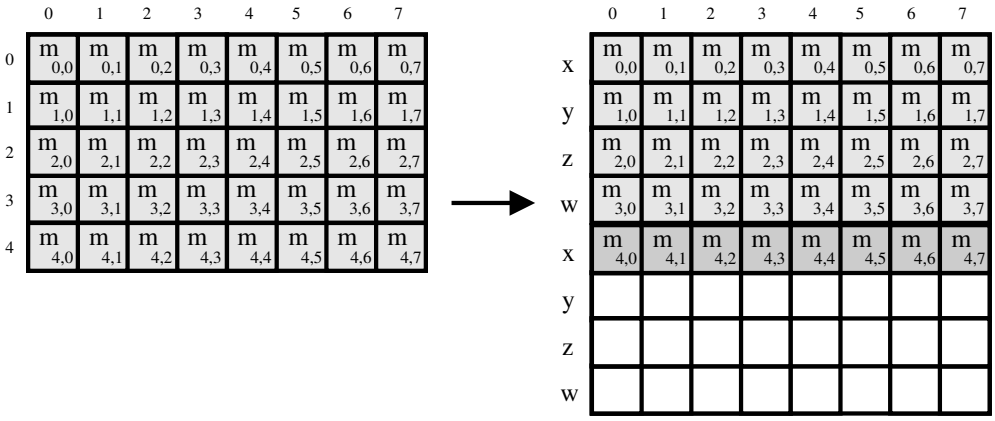
A exemplo de uma operação de adição no SSE, o seguinte algoritmo pode ser utilizado:

```
1 soma_matrizes(matriz A, matriz B, matriz Resultado)
2 {
3     for ( i =0; i < numero_de_linhas(A); i++)
4         for ( j =0; j < numero_de_colunas(B); j++)
5             Resultado_ij = soma_SSE(A_ij, B_ij)
6 }
```

Neste exemplo, as funções `numero_de_linhas()` e `numero_de_colunas()` são responsáveis por retornar o número real de linhas e colunas alocados para a representação da matriz de dados nesta estrutura compacta[1]. Cabe assim à função `soma_sse()` realizar a execução de uma instrução de adição sobre 4 componentes nas matrizes A e B, armazenando o resultado nas 4 componentes correspondentes na matriz $Resultado$.

[1] Apenas para apresentar a nomenclatura usada, pois as dimensões das matrizes devem ser as mesmas nesta operação

Figura 3.5: Matriz fornecida pelo usuário mapeada em uma matriz de textura



O formato interno desta matriz compacta é o mesmo usado por matrizes de textura nas placas gráficas, conforme mostrado pela Figura 3.5. Ela apresenta uma matriz de dimensões 4×7 mapeada em uma matriz de texturas. Como os elementos são armazenados em registradores de 4 componentes, as dimensões reais alocadas podem ultrapassar as dimensões da matriz fornecida pelo usuário. Neste caso as componentes extras são preenchidas com valores nulos, de forma a não influenciar no resultado final.

Com esta representação, a execução de operações como soma e subtração na GPU são realizadas através do armazenamento de uma textura para a primeira matriz e outra textura para a segunda matriz. Após o envio do programa para o processador de fragmentos e da criação de uma geometria de mesma dimensão destas matrizes, o resultado pode ser lido do frame buffer e usado pelo usuário.

3.2.3.2 Multiplicação

Enquanto o mapeamento de operações de adição e subtração é feito com uma certa naturalidade, o produto de duas matrizes requer o uso de técnicas mais complexas quando mapeado para o hardware dedicado. Para apresentá-las, uma breve introdução sobre propriedades de multiplicação de matrizes faz-se necessária.

As dimensões das matrizes A, B e C envolvidas na multiplicação primeiramente devem satisfazer a equação

$$(n \times m)(m \times p) = (n \times p), \tag{3.4}$$

onde $(a \times b)$ denota uma matriz com a linhas e b colunas. A primeira matriz na equação representa as dimensões da matriz A, seguido das dimensões necessárias nas matrizes B e C.

Satisfazendo esta equação, o produto entre as matrizes A e B pode ser denotado da seguinte forma:

$$\begin{bmatrix} C_{11} & C_{12} & \dots & C_{1p} \\ C_{21} & C_{22} & \dots & C_{2p} \\ \vdots & \vdots & \ddots & C_{1p} \\ C_{n1} & C_{n2} & \dots & C_{np} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} & \dots & A_{1m} \\ A_{21} & A_{22} & \dots & A_{2m} \\ \vdots & \vdots & \ddots & A_{1m} \\ A_{n1} & A_{n2} & \dots & A_{nm} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} & \dots & B_{1p} \\ B_{21} & B_{22} & \dots & B_{2p} \\ \vdots & \vdots & \ddots & B_{1p} \\ B_{m1} & B_{m2} & \dots & B_{mp} \end{bmatrix}, \tag{3.5}$$

onde

$$
\begin{array}{rclcl}
C_{11} & = & A_{11}B_{11} & + & A_{12}B_{21} + \ldots + A_{1m}B_{m1} \\
C_{12} & = & A_{11}B_{12} & + & A_{12}B_{22} + \ldots + A_{1m}B_{m2} \\
C_{1p} & = & A_{11}B_{1p} & + & A_{12}B_{2p} + \ldots + A_{1m}B_{mp} \\
C_{21} & = & A_{21}B_{11} & + & A_{22}B_{21} + \ldots + A_{2m}B_{m1} \\
C_{22} & = & A_{21}B_{12} & + & A_{22}B_{22} + \ldots + A_{2m}B_{m2} \\
C_{2p} & = & A_{21}B_{1p} & + & A_{22}B_{2p} + \ldots + A_{2m}B_{mp} \\
C_{n1} & = & A_{n1}B_{11} & + & A_{n2}B_{21} + \ldots + A_{nm}B_{m1} \\
C_{n2} & = & A_{n1}B_{12} & + & A_{n2}B_{22} + \ldots + A_{nm}B_{m2} \\
C_{np} & = & A_{n1}B_{1p} & + & A_{n2}B_{2p} + \ldots + A_{nm}B_{mp}
\end{array}
\tag{3.6}
$$

Assim, a operação do produto entre duas matrizes pode ser representada como uma série de iterações sobre as linhas da primeira matriz e sobre as colunas da segunda. Neste processo, cada elemento de uma coluna na segunda matriz é multiplicado pelas linhas da primeira, resolvendo a solução de uma coluna do resultado final.

Este é um problema no qual diferentes técnicas de otimização já foram abordadas. Em geral estas técnicas envolvem questões como a reorganização dos dados nas matrizes ou tentativas na redução das operações de multiplicação necessárias para o cálculo do produto na CPU.

Otimizações muitas vezes envolvem também a arquitetura na qual o processamento será efetuado. Arquiteturas como o Cray-2 tiveram muitos trabalhos de otimização realizados, visto seu grande uso em aplicações envolvendo o processamento intensivo de matrizes. Como exemplo de otimizações realizadas nesta arquitetura multi-processada cita-se (Bailey 1988), onde o uso da memória local ao processador trouxe uma redução nas contenções de dados nos bancos da memória principal, devido ao compartilhamento do barramento com os outros processadores. Devido à busca freqüente pelos dados na memória, a multiplicação pôde atingir ganhos consideráveis nesta arquitetura.

Desconsiderando a otimização deste processo, esta operação pode ser implementada com o uso de um algoritmo genérico, como expressado a seguir:

```
1 multiplica_matrizes (matriz A, matriz B, matriz Resultado)
2 {
3     for ( i=0; i < numero_de_linhas(A); i++)
4         for ( j=0; j < numero_de_colunas(B); j++) {
5             Resultado_ij = 0
6             for (k=0; k < numero_de_linhas(A); k++)
7                 Resultado_ij = Resultado_ij + A_ik * B_kj
8         }
9 }
```

Este algoritmo, enquanto bastante intuitivo, apresenta uma complexidade assintótica na ordem de $O(n^3)$. Algoritmos de multiplicação de matrizes capazes de reduzir esta quantidade de operações têm sido projetados desde 1969, como o proposto por Strassen (Strassen 1969), capaz de realizar esta mesma operação em uma ordem de $O(n^{2,807})$.

Outros algoritmos foram desenvolvidos após o de Strassen, como o Pan (Pan 1980) e o Coppersmith-Winograd (Coppersmith e Winograd 1982), sendo este último capaz de reduzir o expoente de n para $2,376$. Entretanto estes algoritmos são consideravelmente mais complicados que o proposto por Strassen, além de ganharem desempenho significativo apenas para matrizes de grandes proporções (acima de 1000×1000) (Bailey 1988).

A implementação proposta por Strassen baseia-se em uma propriedade das matrizes que permite que elas sejam representadas através de matrizes menores, chamadas de blocos. Cada um destes blocos na matriz de origem é operado com um bloco correspondente na matriz de destino. A operação de produto entre duas matrizes de bloco quadradas é similar à multiplicação de matrizes convencionais, conforme o exemplo abaixo, onde A e B representam sub-matrizes.

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix} \tag{3.7}$$

Enquanto esta operação exige a computação de 8 operações de multiplicação, o algoritmo de Strassen reduz este número para 7. Como este tipo de operação é computacionalmente caro, este algoritmo torna-se uma alternativa interessante para ser executado na CPU. Este motivo levou à adoção deste algoritmo pela implementação em software da biblioteca desenvolvida.

A solução da multiplicação de matrizes segundo o algoritmo de Strassen faz uso de matrizes auxiliares, que posteriormente são utilizadas para compor o resultado na matriz C, conforme as seguintes equações:

$$\begin{aligned} P_1 &= (A_{11} + A_{22})(B_{11} - B_{22}) \\ P_2 &= (A_{21} + A_{22})B_{11} \\ P_3 &= A_{11}(B_{12} - B_{22}) \\ P_4 &= A_{22}(B_{21} - B_{11}) \\ P_5 &= (A_{11} + A_{12})B_{22} \\ P_6 &= (A_{21} - A_{11})(B_{11} + B_{12}) \\ P_7 &= (A_{12} - A_{22})(B_{21} + B_{22}) \\ C_{11} &= P_1 + P_4 - P_5 + P_7 \\ C_{12} &= P_3 + P_5 \\ C_{21} &= P_2 + P_4 \\ C_{22} &= P_1 + P_3 - P_2 + P_6 \end{aligned} \tag{3.8}$$

Pesquisas sobre a multiplicação de matrizes foram feitas também para permitir seu cálculo com o uso do hardware gráfico. Recursos como os de renderização, suporte a texturas e de processadores de fragmento e vértice programáveis foram e continuam sendo explorados para realizar esta tarefa de maneira eficiente.

Uma das primeiras soluções propostas para a multiplicação eficiente de matrizes na GPU veio com o suporte a múltiplas texturas pelas placas gráficas, com o algoritmo proposto por Larsen e McAllister (Larsen e McAllister 2001). Baseado na na idéia de "visualizar" o processo de renderização de uma imagem a partir de duas texturas, a operação não chegava a envolver a programação de processadores especiais, devido à limitação dos recursos disponíveis no hardware gráfico na época. Como base deste algoritmo estão alguns conceitos a respeito da renderização de imagens, como segue.

Uma matriz de dimensões $m \times n$ pode ser desenhada na tela com a definição de um retângulo de mesmo tamanho, representado por 4 vértices com coordenadas de textura $(0, 0), (0, n-1), (m-1, n-1)$ e $(m-1, 0)$, seguindo o sentido horário a partir da coordenada superior esquerda. Da mesma forma que a matriz de textura tem seu conteúdo reproduzido na tela com esta renderização, sua matriz transposta pode ser obtida apenas modificando o mapeamento da textura neste retângulo. Isto pode ser feito utilizando um retângulo de dimensões $n \times m$, com coordenadas $(0, 0), (m-1, 0), (m-1, n-1)$

e $(0, n-1)$. Com isso, não é necessário armazenar uma segunda textura apenas para representar a matriz transposta.

Nota-se que a notação usada para indexar texturas é tradicionalmente feita através de tuplas (s,t), onde s indexa o eixo horizontal e t indexa o eixo vertical, ao contrário da notação tradicional para matrizes, onde o primeiro índice representa a linha e o segundo a coluna.

O algoritmo de Larsen-McAllister para a multiplicação de matrizes então inicia-se com os seguintes passos:

- Limpa-se a tela, preenchendo-a com a cor branca (valores nulos);
- Configura-se o modo de desenho para *overlay*, permitindo a sobreposição de imagens;
- Carrega-se a textura TexA com a matriz A;
- Carrega-se a textura TexB com a matriz B;
- Configura-se o modo de multi-textura da biblioteca gráfica para fazer modulação. Com isto os valores computados são multiplicados pelos já existentes;
- Configura-se o frame buffer para acumular os valores nele escritos.

Após a configuração do modo de renderização, as iterações sobre as texturas são realizadas conforme a listagem a seguir. As variáveis n e m indicam o número de linhas e colunas da matriz A, respectivamente, enquanto as variáveis m e p as indicam para a matriz B.

1 **for** (i =0; i<numero_de_linhas(A); i++)
2 desenha um retangulo de dimensoes $m \times n$ com coordenadas de:
3 TexA = $(0,i),(0,i),(n-1,i),(n-1,i)$
4 TexB = $(i,0),(i,m-1),(i,m-1),(i,0)$

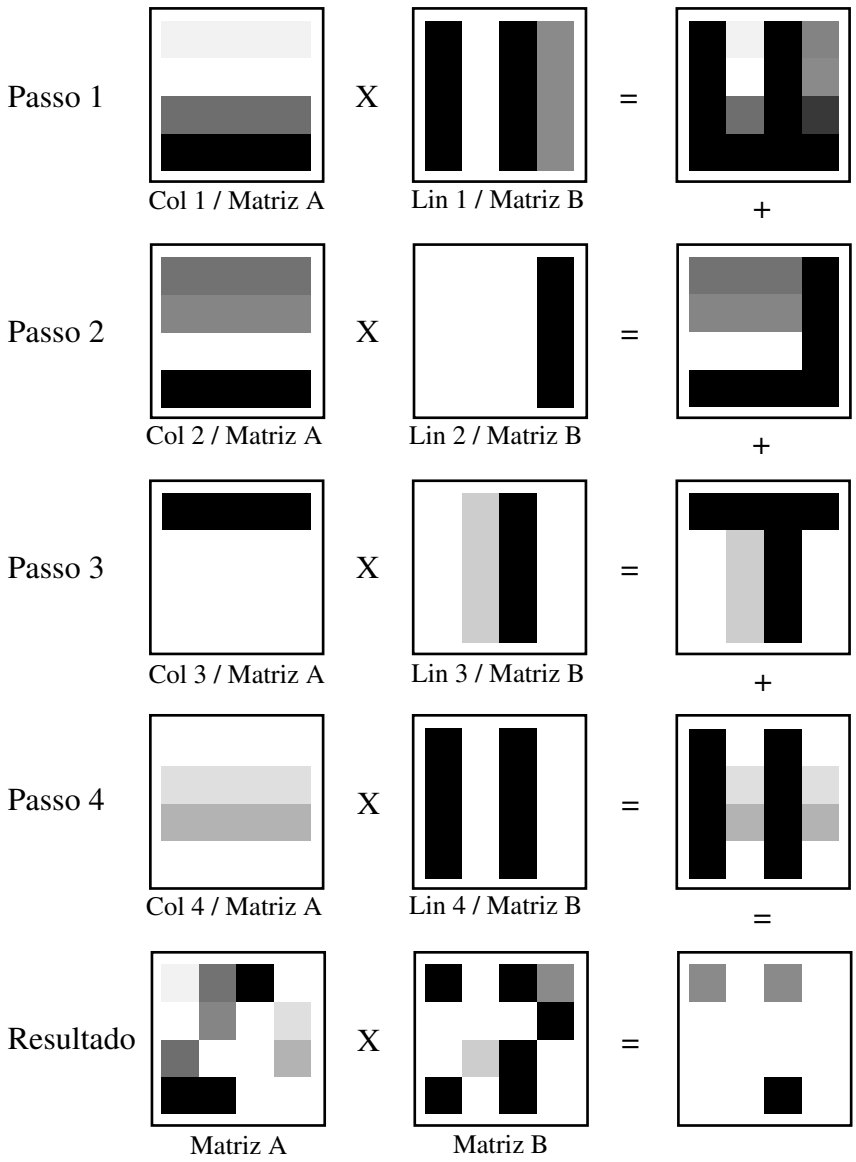
Ao término deste loop, o conteúdo renderizado no frame buffer contém o resultado do produto entre A e B. As iterações realizadas por este algoritmo para uma matriz 4x4 são apresentadas na Figura 3.6[2], onde os dados de entrada são representados por tons de cinza. Devido a esta característica, expressões onde o resultado excede o valor 1 ou é inferior ao valor 0 são truncados a estes valores.

Com a introdução dos processadores de fragmento programáveis nas GPUs, algoritmos mais eficientes puderam passar a ser implementados, visto que as operações sobre matrizes podem ser feitas através da programação deste processador. Abordagens baseadas também nos recursos de multi-texturas para executar o mapeamento dos componentes das matrizes de origem têm estado presente em várias propostas recentes (Bolz et al. 2003)(Krüger e Wester

O uso de código específico ao processador de fragmentos pode ser incorporado ao processo de multiplicação de matrizes com alguma semelhança às iterações realizadas no método proposto por Larsen e McAllister. A idéia é que as iterações sobre as texturas sejam realizadas em software, com o uso de janelas posicionadas sobre áreas específicas destas. Ao serem geradas as geometrias sobre estas texturas, o código do processador de fragmentos para computar a multiplicação das matrizes posicionadas sob estas janelas é executado, salvando o resultado no frame buffer.

[2]Figura obtida de (1)

Figura 3.6: Algoritmo de Larsen-McAllister para duas matrizes 4x4



Este processo pode ser descrito pelo algoritmo a seguir. Ele utiliza uma textura extra para armazenar o resultado acumulado, inicialmente contendo valores nulos. Esta operação extra pode ser evitada com o uso de buffers de acumulação disponíveis em algumas placas gráficas, ou ainda com o uso de renderização direto para outra textura, conforme citado na Seção 3.1. Além de usar apenas a primeira componente dos 4 canais presentes nos registradores do processador de fragmentos, para fins de simplificação o tamanho das matrizes neste algoritmo foi definido como sendo n.

```
1 for (k=0; k < n; k=k+tamanho_do_passo) {
2     for (i=0; i < n; i++) {
3         for (j=0; j < n; j++) {
4
5             // inicio do fragment shader
6             Reg3.r = 0
7             for (m=k; m < k+tamanho_do_passo; m++) {
```

```
8                   Reg1.r = lookup(i, m, A)
9                   Reg2.r = lookup(m, j, B)
10                  Reg3.r = Reg3.r + Reg1.r * Reg2.r
11              }
12              Reg4.r = lookup(i, j, F)
13              Reg0.r = Reg3.r + Reg4.r
14              // fim do fragment shader
15
16          }
17      }
18      Copia frame buffer para textura F
19  }
```

O código do programa de fragmentos (*fragment shader*) interno aos loops neste exemplo realiza uma iteração conforme um número pré-definido de passos. Este número identifica quantas operações de multiplicação serão feitas no código do *shader*, visto que ele é capaz de abrigar uma quantidade variável de instruções, dependendo do hardware. Entretanto, apesar do algoritmo indicar a existência de um loop na linha 7, ele deve ser desenrolado e implementado manualmente para cada operação de multiplicação, visto que grande parte do hardware gráfico atual não suporta instruções para o tratamento de loops. Hardwares recentes como a GeForce FX 6800 possuem tratamento de loops, porém com tamanho constante, limitado pelos processadores de vértice e de fragmento.

Em geral, pode-se aproveitar ainda a propriedade de matrizes que permite sua representação em blocos. O algoritmo anterior pode ser especializado para operar com blocos de mapas de texturas, conforme o algoritmo de múltiplos canais proposto por Hall (1), utilizado no desenvolvimento prático deste trabalho. Ele baseia-se na possibilidade de armazenar as coordenadas de até 4 matrizes dentro de um único canal `RGBA`, conforme

$$X = \begin{bmatrix} A_r & B_g \\ C_b & D_a \end{bmatrix}, Y = \begin{bmatrix} E_r & F_g \\ G_b & H_a \end{bmatrix} \tag{3.9}$$

Os elementos de uma matriz M, seguindo esta representação, podem ser denotados por M_i, onde $i \in r, g, b, a$. De acordo com esta notação, o produto $X_r Y_g$ é igual a AF. Segundo esta notação, é possível estender a técnica de multiplicação de matrizes com apenas uma componente de cor para utilizar os 4 canais disponíveis. O processo de multiplicação para a primeira linha e coluna de duas matrizes X e Y é descrito de acordo com

$$X_{rrbb} Y_{rgrg} = \begin{bmatrix} AE_r & AF_g \\ CE_b & CF_a \end{bmatrix} \tag{3.10}$$

Assim, o produto de duas matrizes 4×4 segundo esta notação pode ser descrito como segue:

$$\begin{aligned} X_{rrbb} Y_{rgrg} &= X_{rrbb} Y_{rgrg} + X_{ggaa} Y_{baba} \\ &= \begin{bmatrix} AE_r & AF_g \\ CE_b & CF_a \end{bmatrix} + \begin{bmatrix} BG_r & BH_g \\ DG_b & DH_a \end{bmatrix} \\ &= Z_{rgba} \end{aligned} \tag{3.11}$$

Para aplicar a fórmula derivada em 3.11, a operação de `lookup()` feita no algoritmo anterior deverá coletar os 4 valores vindos da matriz especificada como parâmetro. Assim,

a operação $Reg0$ = lookup(i, j, X) pode ser usada para recuperar as 4 coordenadas para as matrizes representadas em X, que são A_{ij}, B_{ij}, C_{ij} e D_{ij}.

Finalmente, a multiplicação em si é dada com o algoritmo a seguir:

```
1  for (k=0; k < n; k=k+tamanho_do_passo) {
2      for (i=0; i < n; i++) {
3          for (j=0; j < n; j++) {
4
5              // inicio do fragment shader
6              Reg3.r = 0
7              for (m=k; m < k+tamanho_do_passo; m++) {
8                  Reg1.r = lookup(i, m, X)
9                  Reg2.r = lookup(m, j, Y)
10                 Reg3.r = Reg3 + Reg1.rrbb * Reg2.rgrg
11                 Reg3.r = Reg3 + Reg1.ggaa * Reg2.baba
12             }
13             Reg4.r = lookup(i, j, F)
14             Reg0.r = Reg3 + Reg4
15             // fim do fragment shader
16
17         }
18     }
19     Copia frame buffer para textura F
20 }
```

4 TRABALHOS RELACIONADOS

Alguns trabalhos foram essenciais para a elaboração desta proposta. Os trabalhos que mais influenciaram decisões de design e de implementação estão contidos neste Capítulo, que trata de aspectos relevantes de cada um destes projetos. Com esta apresentação, será possível situar o trabalho proposto em relação aos trabalhos já existentes e em andamento.

4.0.4 libSIMD

Desenvolvida por Iain Nicholson [1], esta biblioteca tem como objetivo fornecer funções para auxiliar no processamento de algoritmos matemáticos. Ela utiliza como base de implementação extensões SIMD de CPUs, como o SSE, o SSE2 e o 3DNow!, além de implementar rotinas em software. O projeto é motivado pela dificuldade de alguns compiladores de gerar código otimizado para estes recursos.

A escolha dos recursos a serem usados é feita em tempo de compilação, com o uso de diretivas especiais para a geração do código. Enquanto esta decisão torna mais fácil a elaboração da arquitetura do software, ela traz um problema de portabilidade. Qualquer programa *linkado* estaticamente a esta biblioteca terá que ser recompilado caso seja executado em uma máquina que não tenha a extensão usada por ele em sua CPU.

A organização interna desta biblioteca, entretanto, é interessante. O corpo de cada função é definido em tempo de compilação, com diretivas do tipo `#ifdef`. Uma rotina de adição sobre dois vetores é definida pela libSIMD da seguinte forma:

```
1  void
2  simd_vector4_add(vector4 a, vector4 b, vector4 c)
3  {
4  #ifdef USE_3DNOW
5      // codigo em assembly para somar os vetores usando 3DNow!
6  #else if defined(USE_SSE)
7      // codigo em assembly para somar os vetores usando SSE
8  #else if defined(USE_SSE2)
9      // codigo em assembly para somar os vetores usando SSE2
10 #else
11     // executa a operacao via software, sem recursos especiais da CPU
12     c[0] = a[0] + b[0];
13     c[1] = a[1] + b[1];
14     c[2] = a[2] + b[2];
15     c[3] = a[3] + b[3];
16 #endif
```

[1] http://libsimd.sourceforge.net

```
17 }
```

A API fornecida pela libSIMD é composta por vários *kernels* semelhantes ao da adição sobre dois vetores. Em geral eles são capazes de operar sobre dois dados de entrada e produzir uma saída, que é escrita em um terceiro parâmetro para a função. Estas funções estão disponíveis para operarem sobre pequenas matrizes (2×2, 3×3 e 4×4) e arrays de até 4 componentes.

Fica a cargo do programador utilizar estas funções para construir suas próprias rotinas, como a multiplicação de matrizes ou de vetores de dimensões variadas.

4.0.5 Cg

Cg é uma linguagem de programação desenvolvida em conjunto pela NVIDIA e pela Microsoft. Seu objetivo é mapear os recursos disponibilizados pelo hardware gráfico em uma linguagem de alto nível, semelhante ao C (Kernighan e Ritchie 1978). A idéia de seguir a filosofia do C vai no sentido de ser uma linguagem orientada ao hardware e de ser uma linguagem de propósito geral, ao contrário de uma linguagem de propósito específico para computação gráfica como a RenderMan Shading Language, da Pixar (Upstill 1990). Esta sub-seção abordará algumas de suas características, enquanto maiores detalhes sobre sua especificação e exemplos de uso da linguagem podem ser encontrados na bibliografia (Fernando e Kilgard 2003).

A linguagem Cg inclui uma variedade de facilidades para suportar as diferentes arquiteturas de hardware gráfico, que apresentam diferentes níveis de funcionalidade. Esta capacidade foi dada à linguagem através da exposição destas diferenças ao programador. Cada processador programável da GPU é definido por um *profile* que especifica qual subconjunto da linguagem é suportado neste processador. Neste caso, um programa que tente usar recursos não suportados pelo seu hardware gráfico não é capaz de passar pela etapa de compilação (Mark et al. 2003).

Cg é construída com base nas APIs OpenGL e DirectX. Ela conta com o apoio de um compilador que é capaz de gerar código intermediário em uma linguagem assembly própria. Este código então passa por um processo de otimização, onde o *profile* do hardware escolhido é usado como parâmetro para a geração das instruções. O código final é repassado para a API gráfica específica, e será invocado no instante apropriado, de acordo com o programa escrito pelo usuário.

Ao processo de otimização cabe também alocar os registradores que serão necessários para que o programa seja mapeado com sucesso nos processadores da GPU. Esta tarefa é diretamente dependente do hardware, pois existe uma grande diferença entre as placas gráficas no número de instruções suportadas e da quantidade de registradores temporários, texturas, entradas para o programa e saídas geradas por ele. Mais uma vez, o uso dos *profiles* faz-se necessário, permitindo que o programa se ajuste à capacidade do hardware gráfico.

Algumas informações extras também são armazenadas nos *profiles*, impondo limitações sobre operações com ponteiros e arrays. Esta imposição é feita devido ao suporte limitado de operações de acesso dependentes de texturas. Eles só estão disponíveis quando são feitas amostragens de texturas, ou quando são lidos dados dos registradores uniformes da GPU – usados para padronizar a passagem de valores constantes (como parâmetros ou valores pré-calculados) entre as etapas do pipeline. Desta forma, um *profile* pode indicar quais são as operações que um determinado processador pode fazer e quais ele não pode.

Como decisão de design, Cg disponibiliza para o usuário a linguagem de assembly intermediária gerada para seus programas. Esta decisão permite que o programador faça

modificações neste código antes que o compilador gere o código específico para o processador, como otimizações manuais (*fine tuning*). Mantendo uma camada intermediária, é possível também invocar o compilador a partir de uma API de programação, além da linha de comando.

A linguagem Cg é também orientada à execução de *kernels*. Toda função que deve ser executada em um processador da GPU é escrita conforme a idéia de streams: dados de entrada são recebidos, computados pela função e passados adiante. Um exemplo da sintaxe é mostrado abaixo, com a semântica dos seus operadores explicada a seguir.

```
1  void
2  realiza_transformacao ( float4   objectPosition      : POSITION,
3                          float4   color               : COLOR,
4                          float4   decalCoord          : TEXCOORD0,
5                      out float4   clipPosition        : POSITION,
6                      out float4   outputColor         : COLOR,
7                      out float4   outputDecalCoord    : TEXCOORD0,
8                  uniform float    brightness ,
9              uniform float4x4  modelViewProjection)
10 {
11      clipPosition  = mul(modelViewProjection,  objectPosition );
12     outputColor =  brightness  *  color ;
13     outputDecalCoord = decalCoord;
14 }
```

Neste exemplo, nota-se que Cg suporta nativamente tipos de dados escalares (*float*), vetoriais (*float4*) e matriciais (*float4x4*). A função `mul` é uma função nativa da linguagem, que multiplica uma matriz por um vetor. Cg permite também a sobrecarga de operadores (Stroustrup 2000), dando ao usuário a possibilidade de utilizar a função `mul` para operar sobre outros tipos de dados, por exemplo.

O modificador `uniform` indica que um parâmetro não irá ter seu valor modificado sobre um lote de vértices ou fragmentos. Os identificadores POSITION, COLOR e TEXCOORD0 seguindo os parâmetros *objectPosition, color* e *decalCoord* especificam como estes parâmetros serão ligados aos recursos da API. No OpenGL, os comandos `glVertex` irão alimentar o identificador POSITION, `glColor` o identificador COLOR e `glMultiTexCoord` os identificadores TEXCOORDn.

O modificador `out` indica que os parâmetros *clipPosition, outputColor* e *outputDecalCoord* serão gerados como saída desta função. O indicador após o ponto-e-vírgula informa como o resultado alimentará os estágios de união de primitivas e de rasterização do pipeline gráfico.

É importante observar que Cg possui o mesmo conjunto de palavras reservadas que C e C++. Com isso, a medida em que o hardware gráfico vai adicionando suporte a novos recursos (como loops e saltos incondicionais), mapeamentos especiais podem ser feitos usando as palavras reservadas que ainda não foram utilizadas.

Cg tem sido utilizada principalmente para a escrita de *shaders*, que são bastante usados para acelerar o processo de renderização de imagens gráficas. Entretando alguns trabalhos têm utilizado a linguagem com a finalidade de utilizar o poder computacional das GPUs para outros fins, como a computação científica. Alguns exemplos nesta área incluem a simulação de nuvens (Harris 2003) e de fluídos (Li et al. 2004).

4.0.6 BrookGPU

Brook é uma linguagem de programação orientada a streams, desenvolvida pela universidade de Stanford. Ela é uma extensão da linguagem ANSI C, projetada para incorporar idéias de computação paralela de dados e de aritmética intensiva em uma linguagem eficiente. Ela foi inicialmente utilizada em processadores de streams como o Merrimac (Dally et al. 2003), também de Stanford, e o Imagine (Kapasi et al. 2002).

Brook para GPUs (BrookGPU) é uma implementação de Brook para o hardware gráfico, com o intuito de ser usada para computação de propósitos gerais. Como base para esta implementação, BrookGPU apresenta um compilador próprio e um sistema de abstração e virtualização de vários aspectos do hardware gráfico (Buck et al. 2004).

Os tipos de dados suportados por BrookGPU consistem em elementos do tipo *float* e tipos de dados vetoriais como os usados por Cg, como *float2*, *float3* e *float4*. Construções do tipo *struct* usando estes tipos de dados nativos são também permitidas, dando ao usuário a possibilidade da criação de seus próprios tipos.

Toda computação é especificada em torno de streams e de *kernels*. A execução de um *kernel* em uma stream invoca implicitamente um loop sobre todos os elementos de uma stream, executando o corpo do *kernel* sobre cada elemento. A palavra reservada `kernel` determina este tipo de função, assim como a palavra reservada `out` indica os parâmetros usados como retorno. O exemplo abaixo apresenta um breve uso de Brook para GPUs.

```
1  kernel void
2  multiplica_e_soma( float a, float4 x<>, float4 y<>, out float4 result<>)
3  {
4      result = a * x + y;
5  }
6
7  int
8  main(void)
9  {
10     float a;
11     float4 X[100], Y[100], Result[100];
12     float4 x[100], y[100], result[100];
13
14     /* inicializa a, X e Y */
15     ...
16
17     /* copia dados da memoria para a stream */
18     streamRead(x, X);
19     streamRead(y, Y);
20
21     /* executa o kernel em todos os elementos */
22     multiplica_e_soma(a, x, y, result);
23
24     /* copia os dados da sream para a memoria */
25     streamWrite( result, Result);
26     ...
27     return 0;
28 }
```

Todos os parâmetros recebidos por um *kernel* em BrookGPU são do tipo stream ou float, sendo permitido apenas o retorno de streams. No caso do uso de floats para a entrada de dados, eles são considerados valores constantes, e não podem ser modificados pelo *kernel*. Brook também faz um ajuste no tamanho das streams de dados: caso as streams de entrada e de saída tenham tamanhos diferentes, cada stream de entrada é alterada para que tenha o mesmo tamanho da de saída. Isto é feito as operações de repetição (de 123 para 111222333, por exemplo) e de *striding* (de 123456789 para 13579).

Conforme listado nas linhas 18, 19 e 25, BrookGPU exige que o programador explicite quando deseja obter os dados para uso em seu programa ou quando deseja usá-los para computá-los em *kernels*. Esta operação é feita com as funções `streamRead()`, que mapeia os dados para um formato que possa ser aproveitado pelo hardware gráfico, e pela `streamWrite()`, que copia o resultado de volta para a CPU.

A escolha dos recursos de hardware que serão usados fica a cargo de uma biblioteca de apoio, que é linkada ao programa compilado pelo usuário. A etapa de compilação conta com uma etapa de pré-processamento, onde o arquivo fonte na linguagem Brook é convertido para código na linguagem C++ e então compilado e linkado com esta biblioteca. A biblioteca fica encarregada de verificar o conteúdo da variável de ambiente `$BRT_RUNTIME`, que irá determinar sobre quais dispositivos o software irá executar. Os recursos de hardware atualmente contam com GPUs da NVIDIA e da ATI, além de permitir o mapeamento de operações sobre a CPU – porém sem utilizar extensões SIMD especiais.

BrookGPU utiliza a linguagem Cg como base para suas operações sobre o hardware gráfico. Todo o código escrito pelo usuário na linguagem Brook (isto é, os *kernels*) é mapeado pelo compilador de Brook em *shaders* de Cg. Estes *shaders* são então usados para gerar código OpenGL ou DirectX, que serão enviados para a GPU em tempo de execução do programa. Conforme o conteúdo da variável de ambiente `$BRT_RUNTIME`, um destes códigos será carregado, salvo quando utilizado o *backend* de CPU.

4.0.7 Sh

Sh é uma linguagem de meta-programação para GPUs programáveis modernas, desenvolvida na Universidade de Waterloo. O termo meta-programação significa que, ao invés de apenas executar as operações especificadas pelo programador, Sh coleta estas operações e cria uma representação intermediária delas, que é então processada por um otimizador e enviada a um compilador determinado.

Sh é construída sobre o C++ através de sobrecarga de operadores e da definição de tuplas especiais e de tipos de dados para representar matrizes. Com isto Sh evita precisar manter seu próprio compilador, como acontece com Cg e com Brook, além de não precisar doutrinar os usuários com a sintaxe e semântica de uma nova linguagem de programação. O uso de classes abstratas, templates, e vários outros recursos de C++ podem ser usados para desenvolver programas em Sh.

As tuplas especiais de Sh podem ser enviadas na forma de streams para a GPU, ou podem ser processadas pela CPU, sem que haja a transformação do código para *fragment shaders* ou *vertex shaders*. O suporte à GPU é fornecido com *backends* OpenGL, onde podem ser especializados o uso de instruções para o hardware da ATI ou para uso genérico, através das extensões ARB, e futuramente incorporará também a API de programação de hardware gráfico SMASH (McCool 2001), desenvolvida pela mesma Universidade. A escolha destes recursos é feita em tempo de compilação, com diretivas especiais para habilitar os *backends* que o usuário deseja utilizar.

Um dos aspectos mais interessantes de Sh é a sua capacidade de combinar e conectar shaders para criar novos shaders. Para isto, as funções executadas pela GPU (os shaders) são tratados como objetos, enquanto são definidos dois operadores para lidar com este tipo de objetos: o operador de conexão e o de combinação. O operador de conexão utiliza combinação funcional; as saídas de um shader são alimentadas como entradas para outro shader. O operador de combinação concatena os canais de entrada, saída e de computação dos dois shaders, gerando um shader completamente novo baseado nestes dois. Nota-se que o shader resultante pode ainda ser processado pelo otimizador de Sh, antes de ser enviado para o compilador específico (McCool et al. 2004).

Sh suporta ainda operações similares, que podem ser usada para manipular streams e aplicar *kernels* computacionais expressados como shaders sobre streams. Ao conectar um shader a uma stream, ele será executado para cada elemento desta stream.

O uso de Sh, entretanto, não abstrai termos de computação gráfica. Os tipos suportados pelo Sh explicitam vetores normais, de cor e de texturas, por exemplo. O shader escrito pelo programador também precisa ser explícito, informando em qual processador ele deverá ser executado (McCool e Toit 2004). O trecho de código abaixo mostra a declaração de um *fragment shader* em Sh, que deverá ser mapeado para executar no processador de fragmentos da GPU.

```
1  fsh = SH_BEGIN_PROGRAM("gpu:fragment") {
2      ShInputNormal3f nv;      // vetor normal
3      ShInputVector3f lv;      // vetor de luz
4      ShInputVector3f vv;      // vetor do observador
5      ShInputColor3f ec;       // irradiancia
6      ShInputTexCoord2f u;     // coordenada de textura
7      ShOutputColor3f fc;      // cor do fragmento
8
9      vv = normalize(vv);
10     lv = normalize(lv);
11     nv = normalize(nv);
12     ShVector3f hv = normalize(lv + vv);
13     fc = kd(u) * ec;
14     fc += ks(u) * pow(pos(hv|nv), spec_exp);
15 } SH_END;
```

5 DESENVOLVIMENTO DA BIBLIOTECA DE ACESSO

Este capítulo apresenta as características da biblioteca desenvolvida para acessar o hardware dedicado e utilizá-lo para fins de computação de problemas genéricos, representados por tipos de dados conforme visto no Capítulo 3. Nele são focados a interface de programação (API) disponibilizada para o usuário, a abstração no acesso aos recursos disponíveis na arquitetura e o modelo do sistema multi-camadas projetado.

A biblioteca considerou, para questões de implementação, as extensões SSE presentes em micro-processadores da arquitetura Intel e AMD, enquanto o uso do hardware gráfico foi limitado à plataforma NV30 da NVIDIA, programado através da biblioteca OpenGL. O foco em uma plataforma específica permitiu que fossem usadas extensões OpenGL otimizadas para esta arquitetura, representada no mercado pelas placas gráficas GeForce FX 5200 e superiores.

A interface de acesso disponível para o usuário foi concebida através de funções de alto nível, onde o usuário não precisa informar informações específicas de computação gráfica, como vértices ou fragmentos, tampouco registradores SSE que seu programa utilizará. Esta abordagem permite que qualquer usuário sem quaisquer conhecimentos sobre programação nestes dispositivos possa utilizá-los para mapear suas aplicações de cálculo.

5.1 Verificação do hardware dedicado disponível

A decisão de suportar diferentes dispositivos dedicados traz ao software o problema de como identificar o hardware disponível na arquitetura. Este problema pode ser atacado de duas maneiras. A primeira é através de diretivas de compilação, onde o usuário especifica para qual alvo o binário será gerado. Neste caso o hardware não chega a ser identificado – apenas assume-se que o hardware existe, e instruções específicas para as extensões solicitadas são adicionadas ao programa.

Quando um programa tenta executar instruções não suportadas pelo micro-processador, como uma chamada a uma instrução SSE quando o micro-processador não implementa esta extensão, o sistema operacional envia um sinal de "instrução ilegal" ao programa. Alguns sistemas operacionais permitem o tratamento deste tipo de sinal, evitando que o programa seja abortado. Com isso o programa pode tentar executar instruções específicas de cada extensão que ele deseja suportar, instalando um manipulador para sinais de instruções ilegais antes de executá-las. A determinação dos recursos suportados pelo micro-processador vêm então da execução ou não do tratador do sinal: caso ele seja chamado logo após a tentativa de execução de uma instrução específica, sabe-se que a extensão ao qual ela pertence não é suportada.

No entanto este tipo de sinal não costuma ser tratável em todos os sistemas operacionais modernos, o que leva a um problema de portabilidade neste método de determinação

de recursos. Isto leva ao segundo método de verificar o hardware disponível: realizando *probe* no hardware em tempo de execução.

Isto traz uma leve desvantagem, pois ao contrário de um binário onde os recursos já estão definidos em tempo de compilação, este método exige que uma rotina para detectar os periféricos seja disparada a cada execução do software. Por outro lado, isto permite que o software se ajuste às características oferecidas pelo hardware sem que ele precise ser recompilado, o que é bastante desejável.

Este trabalho foi desenvolvido considerando o ajuste de acordo com o hardware em tempo de execução. A verificação dos recursos é realizada em duas etapas, que consistem na identificação das extensões presentes no micro-processador e de possíveis placas gráficas.

5.1.1 Identificação de recursos SIMD na CPU

Arquiteturas x86 apresentam desde os micro-processadores Intel Pentium MMX uma instrução especial denominada CPUID (*CPU Identification*) (Corporation 2002). Esta instrução é uma evolução de um mecanismo de identificação de recursos disponíveis nos micro-processadores anteriores a este, e foi criada para permitir a identificação de novas e futuras extensões.

Por ser uma instrução recente e não estar presente em arquiteturas antigas, o suporte à execução da instrução CPUID exige cuidados especiais, pois a sua execução em um micro-processador que não a suporte pode vir a causar um erro de instrução ilegal reportado pelo sistema operacional. A solução para este problema veio com a modificação do comportamento do registrador de EFLAGS na arquitetura x86: quando o bit 21 deste registrador pode ser trocado via software, a instrução CPUID é suportada.

A instrução CPUID é separada em duas funções, onde diferentes informações sobre a CPU são retornadas. A primeira delas traz informações básicas sobre a CPU, como o nome do fabricante e a assinatura do micro-processador. A segunda traz informações estendidas, como o suporte a extensões como o MMX e o SSE. O usuário informa o conjunto de informações desejado através do registrador EAX, seguido da execução da instrução CPUID. O seguinte exemplo indica como estas informações podem ser obtidas:

```
1 ; Obtem as informacoes basicas do micro-processador
2 MOV EAX, 0x0
3 CPUID
4
5 ; Obtem as informacoes estendidas do micro-processador
6 MOV EAX, 0x1
7 CPUID
```

Após a solicitação do conjunto de informações estendidas, os registradores EDX e ECX contém as flags suportadas pelo processador, disponibilizadas em um mapa de bits. Algumas das informações relevantes encontram-se listadas na Tabela 5.1, das quais destacam-se os bits 23, 25 e 26. Através da verificação do valor destes bits é possível determinar se o processador suporta o uso destas extensões: a extensão é suportada caso o bit tenha o valor 1, ou não caso contrário.

Nesta tabela destaca-se também a possibilidade de identificar a existência de um *Time Stamp Counter* (Corporation 2003) (bit 4). O TSC é importante quando medidas de tempo precisas são necessárias, pois ele faz a amostragem do tempo baseado nos ciclos de clock da CPU, sem que hajam conversões de dados por camadas superiores de software.

Tabela 5.1: Algumas flags salvas no registrador EDX ao solicitar informações estendidas da CPU

Bit	Nome	Descrição quando Flag=1
0	FPU	A CPU tem uma unidade de ponto flutuante no chip
4	TSC	A instrução RDTSC é suportada para ler o contador de tempo da CPU
23	MMX	A tecnologia Intel MMX é suportada pela CPU
24	FXSR	A CPU suporta o uso de instruções rápidas para salvar/restaurar o contexto da FPU
25	SSE	Extensões SSE são suportadas pela CPU
26	SSE2	Extensões SSE2 são suportadas pela CPU

5.1.2 Identificação do hardware gráfico

Além de fornecer extensões para a programação de processadores de vértice e fragmentos, bibliotecas gráficas como a OpenGL e o DirectX permitem que os recursos do hardware gráfico sejam reportados através de funções disponibilizadas pela API. Com isto, aplicações tem a possibilidade de adaptar o uso de funções de acordo com as características de desempenho e de recursos do seu hardware.

O OpenGL disponibiliza uma função chamada `glGetString()`. Esta função retorna um ponteiro para uma string estática, onde seu conteúdo varia de acordo com o aspecto solicitado. As extensões suportadas pelo hardware conectado ao OpenGL são obtidas com o parâmetro `GL_EXTENSIONS`. A string retornada consiste em um array separado por espaços indicando as extensões suportadas, com componentes como

- `"ARB_fragment_program"`,
- `"ARB_vertex_program"`,
- `"ARB_texture_rectangle"`,
- `"ATI_fragment_shader"`,
- `"NV_fragment_program"`,
- `"NV_vertex_program"`,
- `"GL_ARB_imaging"`,
- ...

Assim como o OpenGL, a implementação do GLX fornece uma função em sua API para dar ao usuário informações sobre o cliente conectado à biblioteca. Assim como a `glGetString()`, a função `glXGetClientString()` retorna um array de strings. Quando utilizado o parâmetro `GLX_EXTENSIONS`, as extensões suportadas pelo cliente são retornadas, contendo dados como

- `"GLX_SGIX_pbuffer"`,
- `"GLX_SGIX_fbconfig"`,
- `"GLX_NV_float_buffer"`,
- ...

De acordo com a existência das extensões necessárias nestes array, como o suporte a *pbuffers* e à programação do processador de fragmentos, o hardware gráfico pode ser usado para os fins propostos.

Outra técnica também pode ser usada para identificar o hardware gráfico: a varredura do barramento PCI, que responde também por algum dispositivo que esteja presente no barramento AGP. Para cada dispositivo encontrado, seu ID é lido e comparado com uma lista pré-definida de ID's de placas gráficas conhecidas por apresentarem o suporte desejado. Caso o ID seja encontrado nesta lista, o hardware é aceito pela implementação. No caso dele não ser encontrado, não pode-se assumir nada sobre o hardware, pois a lista pode simplesmente não contê-lo (como no caso dela estar desatualizada).

Isto faria com que o primeiro método tivesse que ser utilizado para identificar os recursos do hardware gráfico, trazendo um atraso devido à busca mal sucedida. A varredura do barramento traz ainda uma desvantagem, pois uma biblioteca extra seria necessária apenas para realizar esta tarefa.

Além disto, a própria OpenGL exporta variáveis globais como a `GL_VENDOR`, `GL_RENDERER` e a `GL_VERSION`, que descrevem o fabricante e a identificação da placa gráfica. Estas informações podem ser usadas para comparar o hardware gráfico com uma lista pré-definida de dispositivos, da mesma forma sugerida para quando feito a varredura do barramento.

Estas características fazem com que o uso da API da biblioteca gráfica seja mais adequada para esta função.

5.2 Camada de abstração do hardware

O propósito da biblioteca é utilizar o hardware dedicado disponível para executar as operações solicitadas pelo usuário, de acordo com a interface de programação fornecida a ele. Isto significa que uma abstração deve ser feita no acesso aos recursos disponíveis, visto que não é possível saber *a priori* quais são os dispositivos dedicados que se encontrarão no hardware.

Para satisfazer este requisito, as técnicas de detecção dos recursos disponíveis na arquitetura são necessárias. Após definidas as extensões SIMD presentes na CPU e o hardware gráfico utilizado, estas informações são armazenadas em um mapa de bits, onde cada bit sinaliza a presença ou ausência de algum destes recursos. Após esta identificação, algum destes recursos pode ser escolhido para processar os dados fornecidos pelo usuário. Caso nenhum deles seja identificado na arquitetura, as operações devem recorrer à implementações em software.

Esta abordagem foi concebida com a definição de funções específicas em software (sem o uso de extensões especiais) e com o uso de extensões SSE e da GPU. As funções implementadas em software e no SSE apresentam um *kernel* necessário para iterar sobre os dados de entrada, que são geralmente recebidos através de um loop. Cada *kernel* é responsável por uma operação específica sobre estes dados, como a soma, multiplicação, divisão, subtração e logaritmo, entre outros.

De acordo com o dispositivo dedicado escolhido pelo usuário, uma destas funções será invocada. Para permitir a chamada de diferentes funções a partir de um mesmo protótipo, o uso de ponteiros para funções foi adotado. Estes ponteiros são inicializados no instante em que é determinado o dispositivo que irá processar as operações, sem uma interação explícita com o usuário. No caso de não ser identificado nenhum dispositivo dedicado na arquitetura, todos os ponteiros para estas operações serão apontados para as implementações em software. Esta concepção é visualizada no exemplo abaixo, onde uma interface única de acesso é usada, sendo trocada de acordo com o recurso solicitado pelo usuário:

```
1 use_dispositivo (SSE);
2  for ( i=0; i<n; i++)
3      soma_componentes(A_i, B_i, C_i)  ⟶  soma_componentes_SSE(A_i, B_i, C_i)
4
5 use_dispositivo (CPU);
6  for ( i=0; i<n; i++)
7      soma_componentes(A_i, B_i, C_i)  ⟶  soma_componentes_CPU(A_i, B_i, C_i)
```

Nota-se que existe um sobrecusto na chamada de uma função para *cada iteração* do loop, que pode afetar o desempenho final da aplicação. Este sobrecusto é causado pela necessidade de manipular o registro de ativação na pilha de execução do programa para cada chamada de função executada (Sebesta 1993). Para evitar este problema, funções que encapsulam a operação toda também são disponibilizadas, aliviando a chamada excessiva de um *kernel*. Neste caso a função consistirá de um loop, com as mesmas operações invocadas pelo *kernel* presentes em seu corpo. Diferentes implementações são fornecidas, de forma que ponteiros para função também possam ser usados para invocá-las:

```
1 use_dispositivo (SSE);
2  soma(A, B, C)    ⟶    soma_SSE(A, B, C)
3
4 use_dispositivo (GPU);
5  soma(A, B, C)    ⟶    soma_GPU(A, B, C)
6
7 use_dispositivo (CPU);
8  soma(A, B, C)    ⟶    soma_CPU(A, B, C)
```

A inicialização destes ponteiros é feito pela função de inicialização da biblioteca. Nesta função, o usuário tem a possibilidade de informar quais são os dispositivos que ele deseja usar em seu programa. O retorno desta função é uma máscara que descreve os dispositivos que estarão disponíveis ao programa, combinando os recursos encontrados com os recursos solicitados pelo usuário. O fluxograma indicado na Figura 5.1 mostra o processo de atribuição dos ponteiros durante a inicialização da biblioteca.

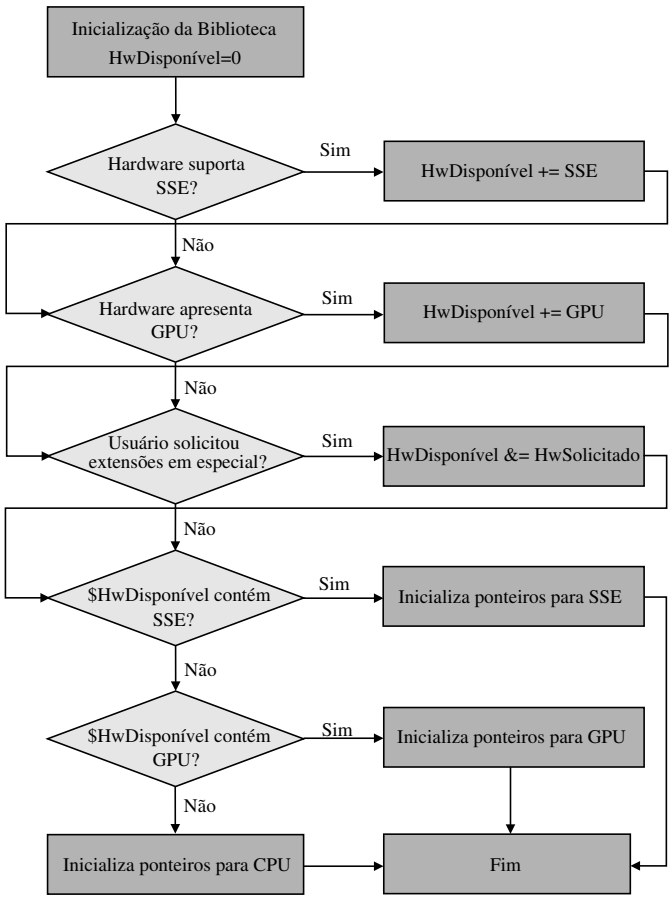
A decisão de trabalhar com ponteiros para funções traz ainda uma vantagem interessante: é possível também implementar apenas *parcialmente* o suporte a um hardware dedicado específico. O papel dos ponteiros neste caso será o de utilizar as funções mapeadas para este hardware, recorrendo a outro recurso para as funções ainda não mapeadas a ele.

Como decisão de design da biblioteca, a implementação das funções mapeadas para o hardware gráfico deixa visível para o usuário o código utilizado pelo processador de fragmentos da GPU. Este código, carregado em tempo de execução, pode ser modificado pelo usuário para incluir instruções mais eficientes que possam vir a estar presentes em placas gráficas futuramente. Esta decisão vai de encontro com a política usada pelo Cg, que permite que o usuário trabalhe com o código em um assembly intermediário para otimizar o assembly final que será usado pela GPU, conforme apresentado na Seção 4.0.5.

5.3 Interface com o usuário

A interface de programação (API) consiste de rotinas para a inicialização e término da biblioteca e de funções para mapear as operações desejadas sobre os recursos disponíveis. Estas funções foram definidas conforme segue:

Figura 5.1: Fluxograma de inicialização da biblioteca



- `int rocket_init(int flags)`
  É a função de inicialização da biblioteca, onde são alocadas estruturas internas como o contexto de renderização OpenGL. Esta rotina também realiza a inicialização dos ponteiros para funções, conforme o fluxograma indicado na Figura 5.1. As flags aceitas por esta função incluem `GPU`, `SSE`, `CPU` e `AUTO`. Esta última é uma flag onde todos os bits tem valor 1, implicando que nenhum dispositivo seja descartado pela biblioteca. Alguns nomes foram reservados para uso futuro, como `SSE`, `AMD_3DNOW` e `DSP`, representando outros hardwares dedicados que poderão ser explorados pela biblioteca.

- `void rocket_finish()`
  Libera os recursos alocados pela biblioteca, como o *pbuffer* (caso tenha sido usada a extensão de GPU) e estruturas auxiliares.

- `int rocket_switch(int flags)`
  Esta função permite que o usuário troque de extensão em tempo de execução. Por exemplo, caso a biblioteca tenha sido inicializada para operar sobre a GPU e deseja-se utilizar as extensões SSE, a chamada à esta função com a flag `SSE` irá trocar os ponteiros para funções para as implementações SSE. O valor de retorno desta função é a nova extensão utilizada, permitindo que o usuário saiba se sua solicitação foi atendida com sucesso.

- `matrix_t *new_matrix(int rows, int columns)`
  Função para criar uma matriz de *rows* linhas e *columns* colunas. Ela retorna um ponteiro para uma estrutura alocada em memória, contendo informações manipuladas pela biblioteca, como as dimensões solicitadas pelo usuário, as dimensões da matriz que foram realmente alocadas, a matriz propriamente dita e um identificador de textura, para o caso dela ser mapeada para a GPU. O primeiro campo desta estrutura é um byte, indicando que esta estrutura armazena uma matriz.

- `int free_matrix(matrix_t *matrix)`
  Libera os recursos alocados pela matriz *matrix*. Esta função retorna 0 caso consiga liberá-los com sucesso, ou propaga o número do erro, caso ocorra algum.

- `vector_t *new_vector(int lenght)`
  Aloca um vetor de tamanho *lenght*. Esta função retorna um ponteiro para uma estrutura em memória, contendo as mesmas informações que a estrutura *matrix_t*. O primeiro campo desta estrutura é um byte, indicando que esta estrutura armazena um vetor.

- `int free_vector(vector_t *vector)`
  Libera os recursos alocados pelo vetor *vector*. Esta função retorna 0 caso consiga liberá-los com sucesso, ou propaga o número do erro, caso ocorra algum.

- `int rocket_add(void *A, void *B, void *C)`
  Define através de uma interface única a operação de adição sobre dois tipos de dados A e B, que podem ser duas matrizes ou dois vetores, salvando o resultado no dado C, que deve ter sido previamente alocado pelo usuário. O retorno da função é 0 caso a operação tenha ocorrido com sucesso, ou o valor do erro propagado, caso contrário.

  Esta função acessa primeiramente o primeiro byte dos parâmetros A e B, identificando o tipo dos dados envolvidos na operação. Caso sejam duas matrizes, a função interna `rocket_add_matrices()` é invocada, Caso sejam dois vetores, a função interna `rocket_add_vectors()` é executada. O tipo de dados do parâmetro C deve corresponder ao dos dados de entrada, e as dimensões ou tamanho dos três parâmetros devem ser os mesmos.

  As duas funções internas são representadas por ponteiros para funções, que poderão apontar para as funções `rocket_add_matrices_CPU()`, `rocket_add_matrices_GPU()` ou `rocket_add_matrices_SSE()`, a exemplo da operação sobre matrizes.

- `int rocket_subtract(void *A, void *B, void *C)`
  Define a subtração sobre os parâmetros A e B, salvando o resultado em C. O comportamento interno e os valores de retorno desta função equivalem-se ao da função `rocket_add()`, com excessão às funções internas executadas, que a exemplo da operação sobre vetores, utiliza funções como `rocket_subtract_vectors_CPU()`, `rocket_subtract_vectors_GPU()` e `rocket_subtract_vectors_SSE()`.

- `int rocket_divide(void *A, void *B, void *C)`
  Divide os componentes correspondentes dos elementos em A pelos elementos em B, salvando o resultado no elemento correspondente em C. Operando da mesma

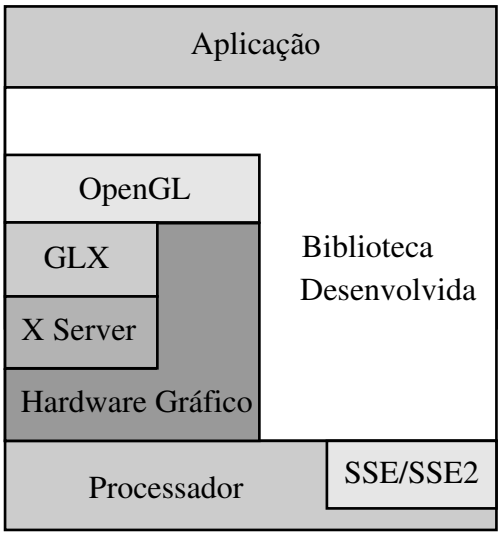
forma que as funções anteriores, esta função admite operações sobre vetores e matrizes, retornando um valor de erro caso sejam fornecidos tipos de dados diferentes como parâmetro.

- `int rocket_multiply(void *A, void *B, void *C)`
  Esta função realiza a multiplicação sobre dois tipos de dados A e B, salvando o resultado na variável C, que deve ter sido previamente alocada pelo usuário. Quando os tipos envolvidos nesta operação são matrizes, a função `rocket_multiply_matrices()` é invocada, ou a função `rocket_multiply_vectors()` caso os tipos sejam vetores. Ambas são ponteiros para funções, que irão utilizar a extensão habilitada no momento da chamada para efetuar o processamento.

  No caso da operação sobre matrizes, a função interna `rocket_multiply_matrices()` verifica ainda se as dimensões das matrizes satisfazem a equação $(n \times m)(m \times p) = (n \times p)$. Caso não satisfaçam, a operação não pode ser realizada, e um valor de erro é retornado para o usuário. Caso a operação ocorra com sucesso, o valor 0 é retornado, ou o número do erro é propagado no retorno da função.

- `int rocket_abs(void *A)`
  Para uma matriz ou vetor de entrada, retorna este vetor ou matriz modificado, contendo apenas os valores absolutos de cada um de seus elementos.

Figura 5.2: A biblioteca desenvolvida sobre as camadas de software e hardware



Como visão geral da interface de acesso fornecida ao usuário, a biblioteca pode ser representada conforme a Figura 5.2: uma única camada de programação é disponibilizada, adaptando-se a diferentes recursos de hardware disponíveis na arquitetura.

6 MEDIDAS DE DESEMPENHO

Em se tratando de uma metodologia para a utilização de hardware dedicado para auxiliar no processamento de cálculo efetivo, medidas de desempenho são essenciais para tomar decisões sobre que dispositivos utilizar para determinadas tarefas. Este Capítulo demonstra algumas operações implementadas na biblioteca desenvolvida, com experimentos e resultados obtidos para cada uma delas.

6.1 Experimentos

A biblioteca foi avaliada em um computador AMD Athlon XP 2000+, com capacidade de 512MB de memória RAM. Esta CPU apresenta as extensões SIMD SSE e 3DNow!, sendo que foram aproveitadas suas extensões SSE para medidas de desempenho (cálculo de ponto flutuante com precisão simples). O hardware gráfico utilizado foi a GeForce FX 5200 da NVIDIA, em um barramento AGP 4x. Os aplicativos de teste e a biblioteca foram compilados com o GCC 3.4.1, na plataforma GNU/Linux, com o kernel 2.6.8.1.

Os experimentos tinham como objetivo determinar os dispositivos que processavam em menor tempo as operações solicitadas. Foram mapeadas as operações de multiplicação, divisão, soma e de valor absoluto em matrizes de ordens de 200×200, 400×400, 600×600, 800×800, 1000×1000, 1200×1200, 1400×1400, 1600×1600, 1800×1800 e 2000×2000.

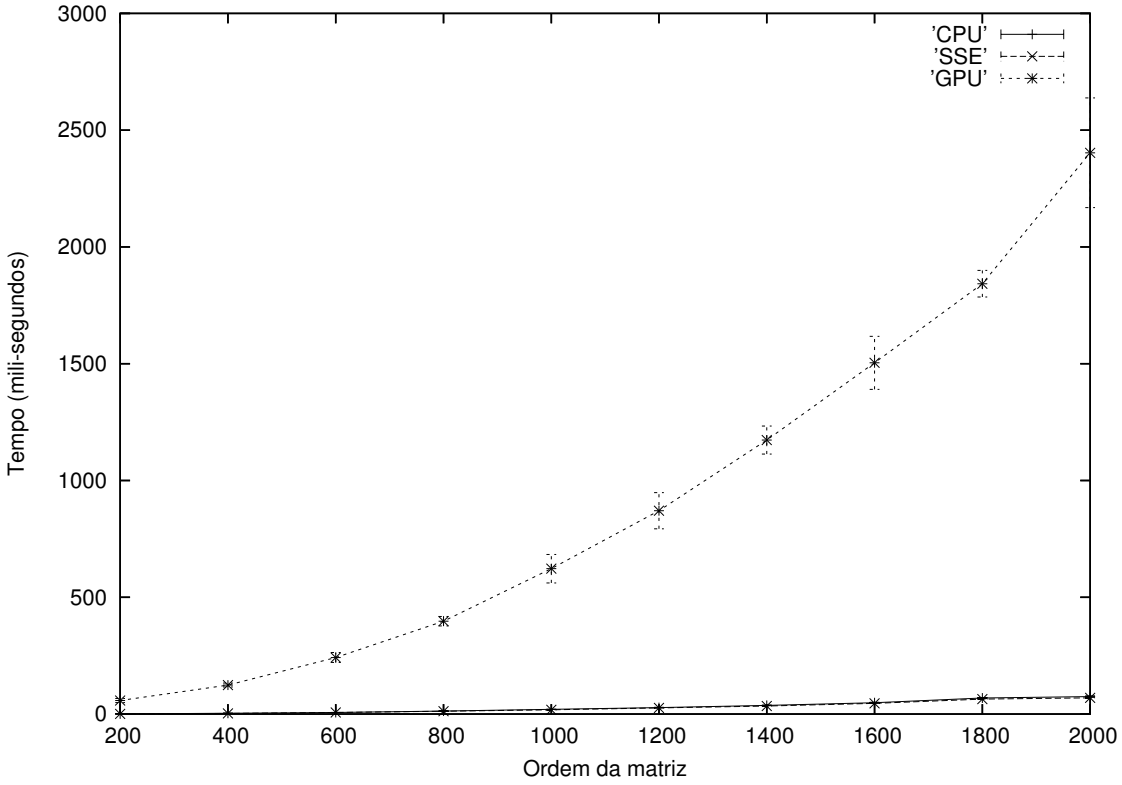
As medidas foram realizadas após uma série de 100 iterações, de onde foram tiradas a média e o desvio padrão. A tomada de tempo teve início no momento em que os loops via software eram chamados, com término após o processamento das operações exigidas neste loop. No caso da GPU, foi ainda computado o tempo necessário para enviar os dados para a GPU na forma de texturas e para compilar o *fragment shader*. Estes passos extras são exigidos quando é realizada uma computação sobre dados que ainda não se encontram na memória da GPU.

6.2 Resultados

As operações envolvendo instruções simples da CPU e com o SSE mostraram-se equivalentes em todos os experimentos realizados, com uma diferença maior apenas na operação de multiplicação de matrizes. Esta semelhança pode ser constatada nas Figuras 6.1,6.2 e 6.3, representando as operações de divisão, valor absoluto e adição, respectivamente.

Estas 3 operações, por apresentarem pouca carga de trabalho para cada elemento das matrizes (A_{ij}/B_{ij} para a divisão, $A_{ij} + B_{ij}$ para a adição, "*se* $A_{ij} < 0$ *então* $A_{ij} = A_{ij} * -1.0$" para o valor absoluto), não exigem muito poder computacional. O uso do

Figura 6.1: Operação de divisão de elementos de duas matrizes



SSE, neste caso, tomou como vantagem a operação sobre até 4 valores com o uso de uma única instrução, apenas.

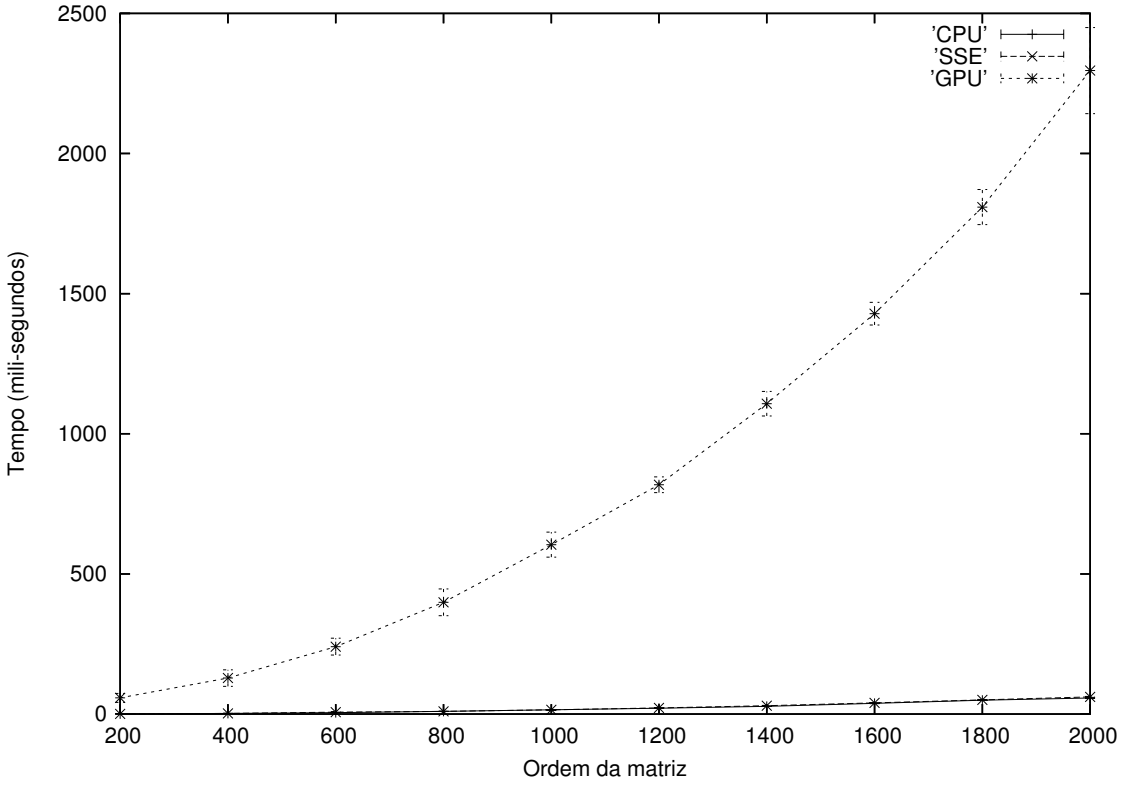
A principal causa do bom desempenho nas operações sobre a CPU vem do fato de que elas são compostas por loops desenrolados. Aliado ao nível alto de otimização usado no compilador GCC, estas instruções executadas na FPU acabam utilizando instruções otimizadas, diminuindo a disparidade entre instruções simples da CPU/FPU e o SSE. Esta disparidade tende a diminuir ainda a medida em que a velocidade do clock do processador aumenta, visto que estas operações ficaram abaixo do tempo de 2,5 segundos.

A execução destas operações sobre a GPU consumiu um tempo semelhante, independente da operação. O gráfico destas 3 operações sobre a GPU pode ser vistos na Figura 6.4. Nota-se que nos pontos onde há uma diferença de tempo visível, o desvio padrão também o é (indicado pelas barras verticais sobre cada ponto amostrado).

O crescimento do tempo na GPU é dado principalmente pela necessidade da transferência dos dados no barramento, como as matrizes de textura, e para a leitura do resultado. Além disso, ainda é necessário manipular o buffer de renderização (*pbuffer*) e instruir o OpenGL sobre as coordenadas de matriz. Estes tempos constantes, que precisam ser executados sempre quando uma operação for mapeada para a GPU, acabam ocultando os custos com a operação em si – principalmente quando elas exigem poucas instruções, como as envolvidas nestas 3 operações em questão.

Quando uma operação é realizada na GPU sobre dados que já se encontram em sua memória, o tempo necessário para realizar o processamento é significativamente menor. Em situações onde diversas operações são feitas sobre os mesmos dados de entrada, basta fazer o upload dos shaders atualizados para a GPU e realizar os loops via software. Estes

Figura 6.2: Operação de valor absoluto em elementos de uma matriz



loops são encarregados de amostrar regiões das texturas, que têm seus valores lidos pelo shader. Apenas após o término de todas as operações desejadas, o resultado pode vir a ser lido do frame buffer. O tempo de processamento efetivo na GPU – sem considerar a transferência de dados com a CPU – é apresentado na Figura 6.5.

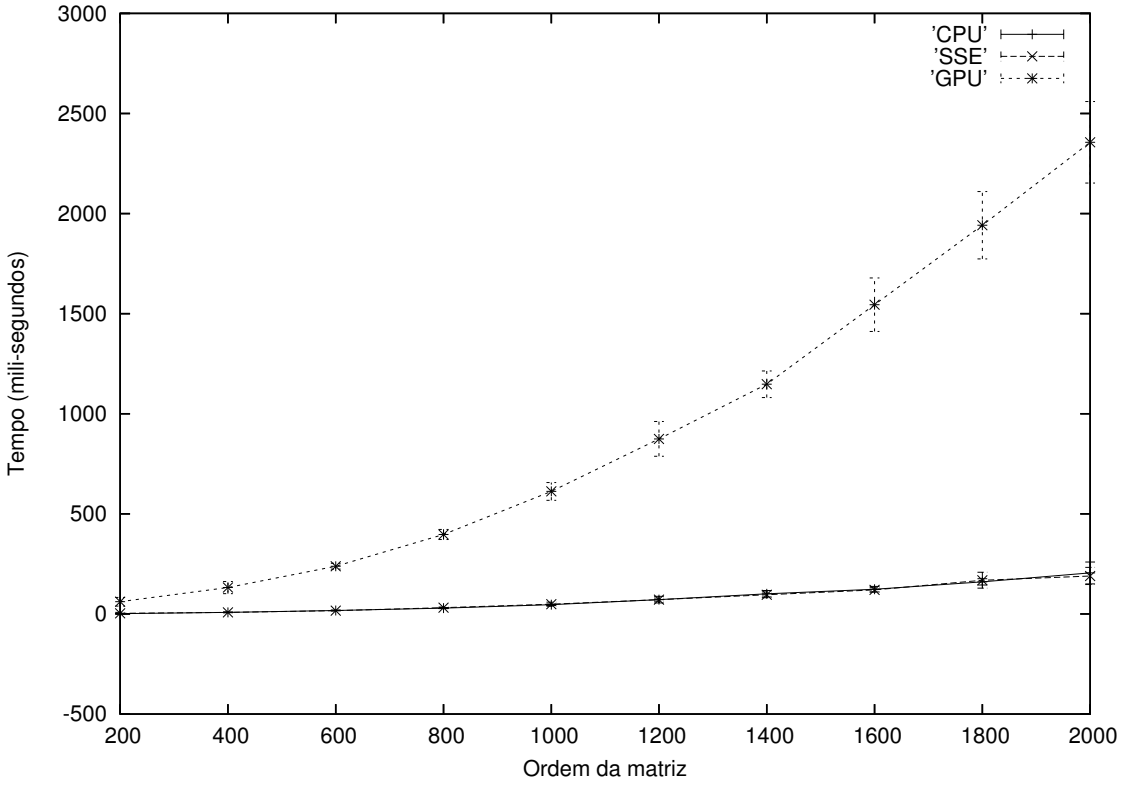
Este tempo também pode ser visualizado em relação ao tempo de computação na CPU e com o SSE, conforme indicado na Figura 6.6 para a operação de soma. Esta comparação é interessante para demonstrar a partir de que ponto o processamento pela GPU é mais eficiente que pela CPU. Pode-se ver que para matrizes pequenas, mesmo desconsiderando as taxas de transferência, é mais vantajoso fazer a operação em CPU.

A operação de multiplicação de matrizes, por sua vez, mostrou-se bastante interessante. Como envolve bastante processamento de ponto flutuante, a execução sobre a CPU e SSE mostrou-se custosa. O gráfico na Figura 6.7 mostra a curva do tempo gasto para este cálculo. Nesta operação o uso do SSE mostra-se mais eficiente que as operações sobre a CPU durante boa parte das execuções. As medidas de tempo na linha indicada por `GPU-2` referem-se à operação de multiplicação na GPU sem considerar a tranferência e sincronização de dados com a CPU.

Com o aumento na complexidade das operações exigidas pelo processador de fragmentos na GPU, o tempo de processamento desta operação foi maior em relação à soma, divisão e valor absoluto. Além disso, a CPU fica responsável por manter controle sobre os loops para o posicionamento das coordenadas das matrizes de textura, influenciando no tempo de processamento final.

Os resultados obtidos com estas computações serviram como base para melhor determinar os dispositivos a serem usados nas operações pela biblioteca. Esta determinação

Figura 6.3: Operação de soma entre elementos de duas matrizes



é feita quando o usuário não indica o recurso que deseja explorar pela sua aplicação, conforme o fluxograma indicado na Figura 5.1.

Desta forma, quando um recurso de GPU é encontrado na arquitetura e o usuário pede para que os recursos sejam escolhidos pela biblioteca, os ponteiros para a multiplicação de matrizes são configurados para usar o mapeamento pela GPU.

Os resultados também serviram para melhor definir as prioridades de uso dos dispositivos pela biblioteca. Com isso, o uso da CPU é priorizado em relação ao da GPU, que é usada apenas quando solicitado pelo usuário, ou conforme a inicialização em modo automático pela biblioteca. Com isso, os hardwares dedicados são usados naquelas operações que melhor se sobressaem, trazendo os melhores mapeamentos possíveis para a aplicação – sem a intervenção do programador.

Figura 6.4: Operações de soma, divisão e valor absoluto na GPU

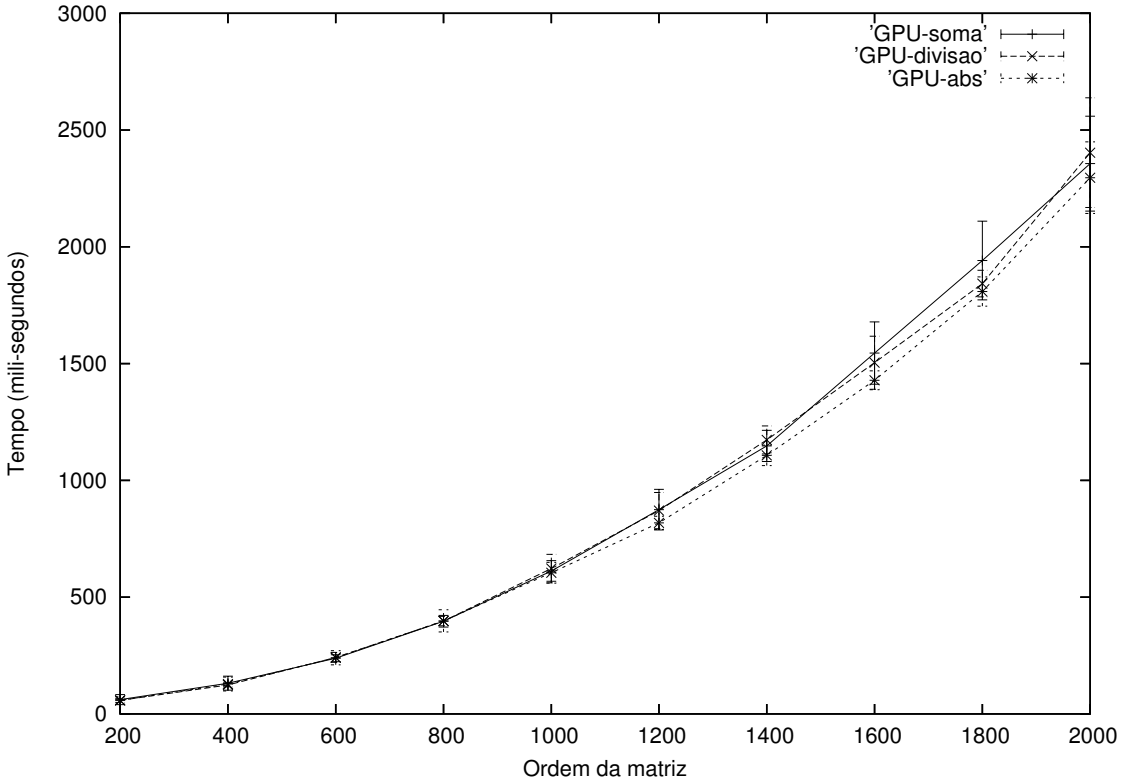


Figura 6.5: Operações na GPU desconsiderando transferências de dados com a CPU

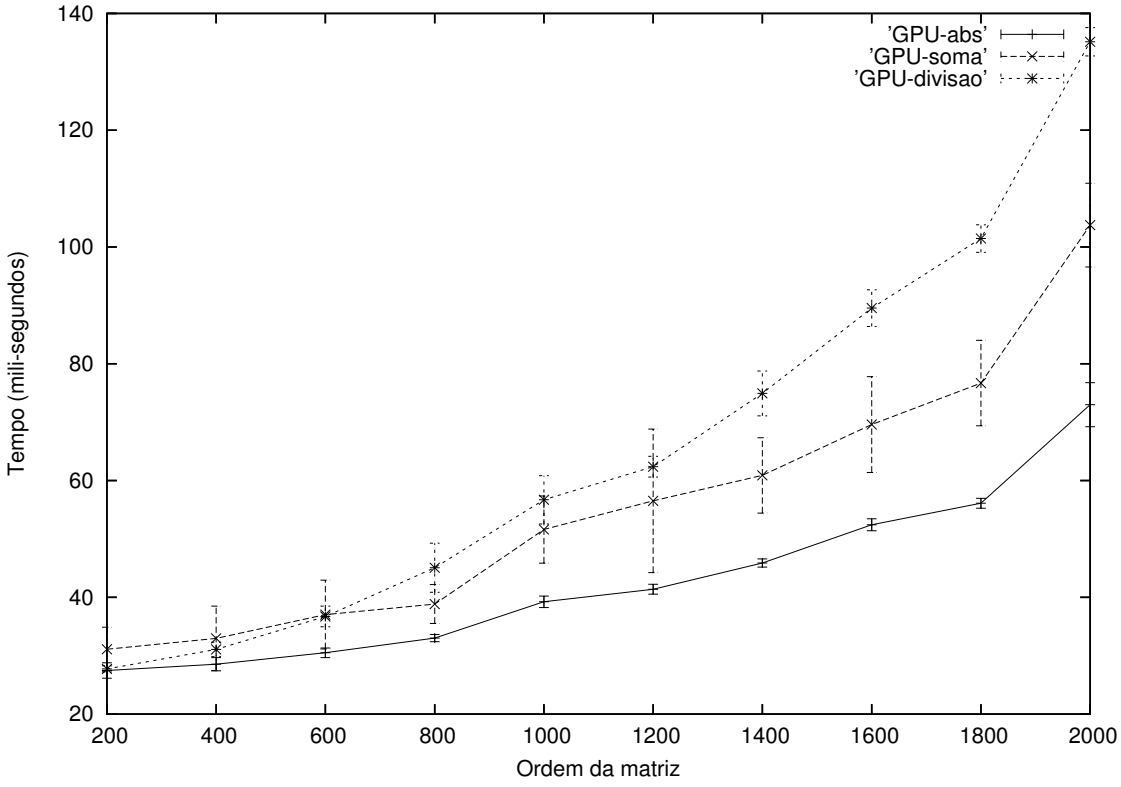


Figura 6.6: Operação de soma desconsiderando transferências de dados entre GPU e CPU

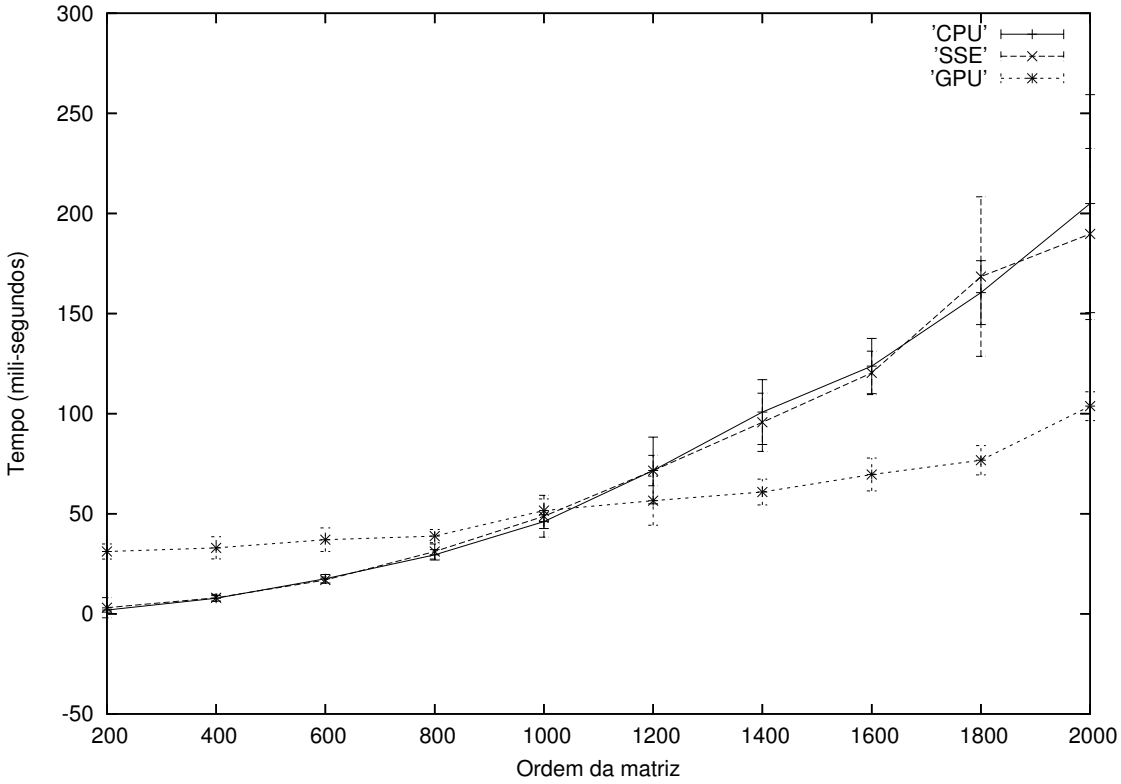
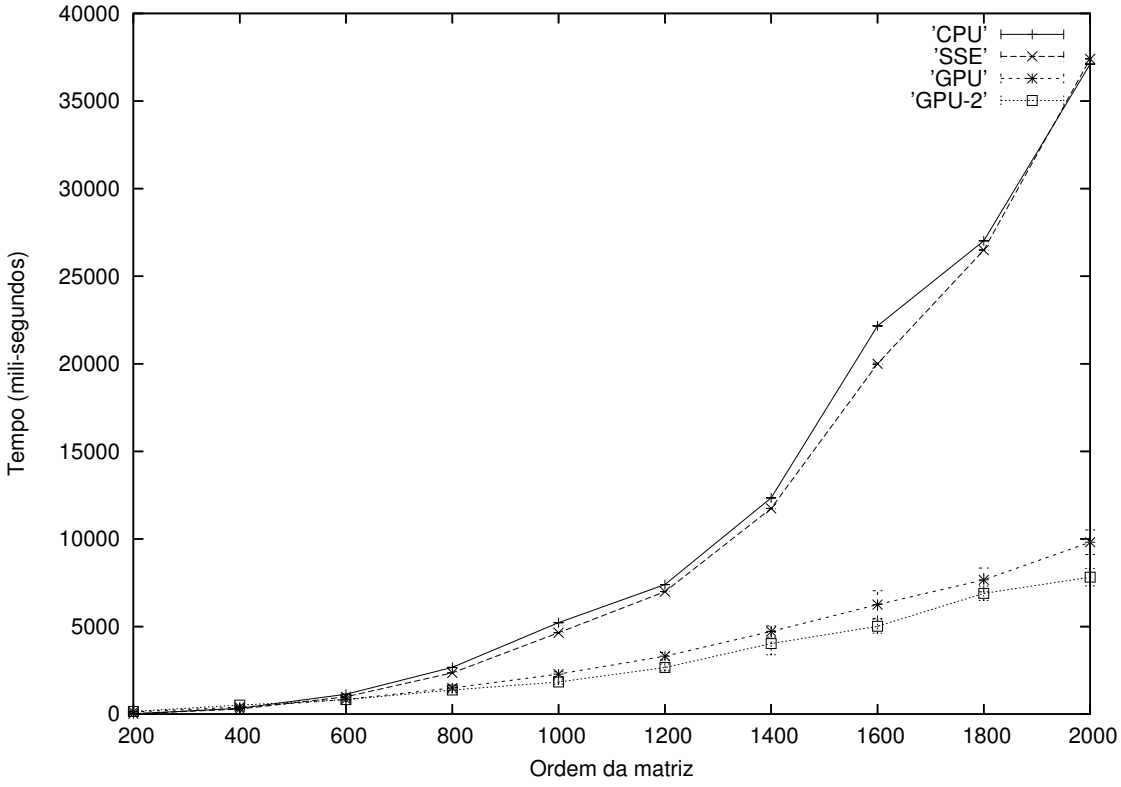


Figura 6.7: Operação de multiplicação de duas matrizes



7 CONCLUSÕES

Este trabalho apresentou o uso de hardware dedicado para a solução de problemas genéricos de computação, representáveis por estruturas de dados como vetores e matrizes. Para tanto, foram discutidas técnicas de programação, de mapeamento desta classe de problemas para o hardware dedicado e uma proposta de uso eficiente destes dispositivos.

O trabalho permitiu identificar, a partir de um conjunto pré-definido, os melhores recursos de hardware disponibilizados em uma arquitetura. Para isto foram feitas análises de desempenho de operações implementadas sobre cada um dos dispositivos definidos previamente.

Com interfaces de alto nível que expõem ao programador apenas as operações que ele deseja realizar, o mapeamento sobre estes dispositivos é feito automaticamente. Isto evita que o programador precise explicitar a alocação de registradores da CPU ou de texturas para o hardware gráfico.

A detecção automática do hardware disponível, unido à uma API de alto nível, traz ao usuário uma característica importante: a portabilidade de sua aplicação. Como a biblioteca define em tempo de execução os recursos que serão utilizados, é garantido que seu programa irá executar em qualquer arquitetura com suporte a ponto flutuante. A portabilidade neste caso vem em dois fatores: a portabilidade de código e a portabilidade de desempenho, que é obtida com a adaptação à arquitetura de hardware pela biblioteca.

O uso de GPUs trouxe a este trabalho a questão da possibilidade de seu uso para a computação genérica, visto que ele é um dispositivo especializado. O mapeamento de operações a este dispositivo é muitas vezes não óbvio e não trivial de ser implementado. Esta característica tende a mudar nos próximos anos, dado o uso cada vez mais freqüente de GPUs para processamento genérico[1].

A disponibilidade de APIs de programação de abstração a estes dispositivos facilita o uso do hardware dedicado para estes fins. APIs de alto nível permitem ainda que seja adicionado suporte a outros dispositivos, como placas de som programáveis e outras extensões SIMD de processadores, ou ainda a otimização de instruções já existentes. Isto tudo pode ser realizado sem retirar ou modificar a interface disponibilizada para o usuário.

Esta extensão a novos dispositivos é tema de trabalho futuro, assim como a adição de novos procedimentos para a biblioteca, como a multiplicação de matrizes e vetores, a transformada de Fourier e o cálculo de valores mínimos e máximos, entre outros. Acredita-se que com os avanços de programabilidade de dispositivos dedicados, como o hardware gráfico, seja possível ampliar ainda mais o escopo de seu uso, viabilizando a sua utilização direta através de suporte nativo pelo sistema operacional.

[1] Ver http://www.gpgpu.org

REFERÊNCIAS

[Amanatides 1987]AMANATIDES, J. Realism in Computer Graphics: A Survey. *IEEE Computer Graphics and Applications*, v. 1, p. 44–56, jan. 1987.

[Ati 2003]ATI, M. D. Programmability Features of Graphics Hardware. *ACM SIGGRAPH 2003*, abr. 2003. ACM SIGGRAPH Course Notes #11, 2003.

[Bailey 1988]BAILEY, D. Extra-High Speed Matrix Multiplication on the Cray-2. *SIAM Journal on Scientific and Statistical Computing*, v. 9, n. 3, p. 603–607, May 1988. Disponível em: <citeseer.ist.psu.edu/bailey88extrahigh.html>.

[Bajaj et al. 2004]BAJAJ, C. et al. SIMD Optimization of Linear Expressions for Programmable Graphics Hardware. *Computer Graphics Forum*, v. 23, p. 1–18, 2004.

[Bolz et al. 2003]BOLZ, J. et al. Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid. *ACM SIGGRAPH 2003*, v. 1, ago. 2003.

[Brown 2002]BROWN, P. *ARB_vertex_program extension*. [S.l.], jun. 2002. ARB Extension #26.

[Buck et al. 2004]BUCK, I. et al. Brook for GPUs: Stream Computing on Graphics Hardware. *ACM SIGGRAPH 2004*, v. 1, ago. 2004.

[Comba et al. 2003]COMBA, J. L. D. et al. Computation on GPUs: From a Programmable Pipeline to an Efficient Stream Processor. In: *Revista de Informática Teórica e Aplicada - RITA*. [S.l.]: Instituto de Informática da UFRGS, 2003. v. 10, p. 41–70.

[Coppersmith e Winograd 1982]COPPERSMITH, D.; WINOGRAD, S. On the Asymptotic Complexity of Matrix Multiplication. *SIAM Journal on Computing*, v. 11, p. 472–492, 1982.

[Corporation 1994]CORPORATION, I. *Intel387 SX Math Coprocessor*. jan. 1994. Datasheet.

[Corporation 2002]CORPORATION, I. *Intel Processor Identification and the CPUID Instruction*. 2002. Application Note.

[Corporation 2003]CORPORATION, I. *IA-32 Intel Architecture Software Developer's Manual*. 2003. Instruction Set Reference.

[Corporation 2004]CORPORATION, I. *IA-32 Intel Architecture Software Developer's Manual*. 2004. Basic Architecture.

[Corporation 2003]CORPORATION, N. *NVIDIA OpenGL Extension Specifications*. [S.l.], jun. 2003.

[Dally et al. 2003]DALLY, W. J. et al. Merrimac: Supercomputing with streams. In: *SC'03*. Phoenix, Arizona: [s.n.], 2003.

[Devices 2000]DEVICES, I. A. M. *3DNow! Technology Manual*. 2000. Application Note.

[Fernando e Kilgard 2003]FERNANDO, R.; KILGARD, M. J. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. [S.l.]: Addison-Wesley Pub Co, 2003. ISBN 0321194969.

[Foley et al. 1996]FOLEY, J. et al. *Computer Graphics. Principles and Practice. 2nd Edition in C*. [S.l.]: Addison-Wesley, 1996. FOL j 96:1 1.Ex. ISBN 0-201-84840-6.

[Foundation 2002]FOUNDATION, F. S. *Using the GNU Compiler Collection (GCC)*. [S.l.]: Free Software Foundation, 59 Temple Place - Suite 330, Boston, MA 02111-1307 USA, 2002.

[1]HALL, J. D.; CARR, N. A.; HART, J. C. *Cache and Bandwidth Aware Matrix Multiplication on the GPU*. Dept. of Computer Science, mar. 2003. Tech Report UIUCDCS-R-2003-2328.

[Harris 2003]HARRIS, M. J. *Real-Time Cloud Simulation and Rendering*. [S.l.], set. 2003. Tech Report #TR03-040.

[Hoge 1998]HOGE, S. FX8010 - A DSP Chip Architecture for Audio Effects. *First COST-G6 Workshopon Digital Audio Effects - DAFX'98*, nov. 1998.

[Inc. 1999]INC., C. L. *Processor with Instruction Set for Audio Effects*. [S.l.], jan. 1999. WIPO patent: WO 9901814 (A1), Jan. 14, 1999.

[Kapasi et al. 2002]KAPASI, U. J. et al. The Imagine Stream Processor. *IEEE International Conference on Computer Design*, p. 282–288, set. 2002. Disponível em: <citeseer.ist.psu.edu/kapasi02imagine.html>.

[Kapasi et al. 2003]KAPASI, U. J. et al. Programmable Stream Processors. *IEEE Computer Society*, v. 1, p. 54–62, ago. 2003.

[Kernighan e Ritchie 1978]KERNIGHAN, B. W.; RITCHIE, D. M. *The C Programming Language*. Englewood Cliffs, New Jersey: Prentice-Hall, 1978.

[Krüger e Westermann 2003]KRüGER, J.; WESTERMANN, R. Linear Algebra Operators for GPU Implementation of Numerical Algorithms. *SIGGRAPH 2003*, p. 908–916, jul. 2003.

[Larsen e McAllister 2001]LARSEN, S.; MCALLISTER, D. Fast Matrix Multiplies using Graphics Hardware. *Supercomputing 2001*, v. 1, nov. 2001. Disponível em: <http://www.sc2001.org/papers/pap.pap313.pdf>.

[Li et al. 2004]LI, W. et al. *GPU-Based Flow Simulation with Complex Boundaries*. [S.l.], nov. 2004.

[Mark et al. 2003]MARK, W. R. et al. Cg: A System for Programming Graphics Hardware in a C-like Language. *ACM SIGGRAPH 2003*, v. 22, jul. 2003.

[Marr 2002]MARR, D. e. a. *Hyper-Threading Technology Architecture and Microarchitecture: A Hypertext History*. fev. 2002. Intel Technology Journal.

[McCool 2001]MCCOOL, M. D. SMASH: A Next-Generation API for Programmable Graphics Accelerators. *SIGGRAPH 2001*, abr. 2001.

[McCool e Toit 2004]MCCOOL, M. D.; TOIT, S. D. *Metaprogramming GPUs with Sh*. [S.l.]: AK Peters, Ltd., 2004. ISBN 1568812299.

[McCool et al. 2004]MCCOOL, M. D. et al. Shader Algebra. *SIGGRAPH 2004*, 2004.

[Moreland e Angel 2003]MORELAND, K.; ANGEL, E. The FFT on a GPU. *Graphics Hardware*, 2003.

[Oat 2002]OAT, C. *Rendering to an off-screen buffer with WGL_ARB_pbuffer*. [S.l.], 2002. Http://citeseer.ist.psu.edu/539700.html.

[P754 1985]P754, I. T. *ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic*. [S.l.], ago. 12 1985. A preliminary draft was published in the January 1980 issue of IEEE Computer, together with several companion articles. Available from the IEEE Service Center, Piscataway, NJ, USA.

[Pan 1980]PAN, V. Y. New Fast Algorithms for Matrix Operations. *SIAM Journal on Computing*, v. 9, p. 321–342, 1980.

[Peleg e Weiser 1996]PELEG, A.; WEISER, U. MMX Technology Extension to the Intel Architecture. *IEEE Micro*, ago. 1996.

[Poddar e Womack 2001]PODDAR, B.; WOMACK, P. *WGL_ARB_render_texture extension*. jul. 2001. ARB Extension #20.

[Purcell et al. 2002]PURCELL, T. J. et al. Ray Tracing on Programmable Graphics Hardware. *ACM SIGGRAPH*, 2002.

[Sebesta 1993]SEBESTA, R. W. *Concepts of Programming Languages*. [S.l.]: Benjamin/-Cummings Publishing Company,Inc., 1993. ISBN 0-8053-7130-3.

[Strassen 1969]STRASSEN, V. Gaussian Elimination is not Optimal. *Numerical Mathematics*, v. 13, p. 354–356, 1969.

[Stroustrup 2000]STROUSTRUP, B. *The C++ Programming Languages*. [S.l.]: Addison-Wesley, 2000.

[Thakkar e Huff 1999]THAKKAR, S. T.; HUFF, T. The Internet Streaming SIMD Extensions. *Intel Technology Journal Q2*, 1999.

[Upstill 1990]UPSTILL, S. *The RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics*. [S.l.]: Addison Wesley Professional, 1990. ISBN 0201508680.