Pontifícia Universidade Católica do Rio de Janeiro

Paulo Roberto Pereira de Souza Filho

Auxílio a Portabilidade de Código em Aplicações de Alto Desempenho

PUC-Rio - Certificação Digital Nº 1421599/CA

Dissertação de Mestrado

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre pelo Programa de Pós-graduação em Informática do Departamento de Informática do Centro Técnico e Científico da PUC-Rio.

Orientador: Profª. Noemi de La Rocque Rodriguez

Rio de Janeiro
Março de 2016

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

Paulo Roberto Pereira de Souza Filho

Auxílio a Portabilidade de Código em Aplicações de Alto Desempenho

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre pelo Programa de Pós-graduação em Informática do Departamento de Informática do Centro Técnico e Científico da PUC-Rio. Aprovada pela Comissão Examinadora abaixo assinada.

Profª. Noemi de La Rocque Rodriguez
Orientador
Departamento de Informática — PUC–Rio

Prof. Jairo Panetta
ITA

Prof. Waldemar Celes Filho
Departamento de Informática — PUC-Rio

Prof. Roberto Ierusalimschy
Departamento de Informática — PUC-Rio

Prof. Marcio da Silveira Carvalho
Coordenador Setorial do Centro Técnico Científico — PUC–Rio

Rio de Janeiro, 31 de março de 2016

PUC-Rio - Certificação Digital Nº 1421599/CA

Todos os direitos reservados. É proibida a reprodução total ou parcial do trabalho sem autorização da universidade, do autor e do orientador.

Paulo Roberto Pereira de Souza Filho

Graduou–se em Engenharia da Computação pela Pontifícia Universidade Católica do Paraná (PUC–PR). Trabalha há mais de 13 anos na Petrobras com aplicações de alto desempenho.

PUC-Rio - Certificação Digital Nº 1421599/CA

Ficha Catalográfica

Souza Filho, Paulo Roberto Pereira de

Auxílio a portabilidade de código em aplicações de alto desempenho / Paulo Roberto Pereira de Souza Filho; orientador: Noemi de La Rocque Rodriguez. — 2016.

117 f. : il. (color.); 30 cm

Dissertação (mestrado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2016.

Inclui bibliografia.

1. Informática – Teses. 2. Computação de alto desempenho. 3. Arquiteturas heterogêneas. 4. Vetorização explícita. I. Rodriguez, Noemi de La Rocque. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Agradecimentos

Aos meus Professores Noemi Rodriguez e Jairo Panetta pelo apoio e incentivo para a realização deste trabalho.

À Petrobras e e à PUC–Rio, pelos auxílios concedidos, sem os quais este trabalho não poderia ter sido realizado.

À minha esposa, que me acompanhou todo esse tempo com apoio direto e indireto.

Aos meus pais, família e amigos, que me apoiaram mesmo com minha ausência na vida familiar.

A todos os colegas, professores e funcionários do Departamento de Informática da PUC-Rio, pelo companheirismo, aprendizado e auxílio.

PUC-Rio - Certificação Digital Nº 1421599/CA

Resumo

Souza Filho, Paulo Roberto Pereira de; Rodriguez, Noemi de La Rocque. **Auxílio a Portabilidade de Código em Aplicações de Alto Desempenho**. Rio de Janeiro, 2016. 117p. Dissertação de Mestrado — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Atualmente na computação de alto desempenho existem diversas opções de arquiteturas de diversos fabricantes, algumas sendo heterogêneas como por exemplo CPU+GPU. Este trabalho tem como objetivo implementar maneiras de codificar aplicações de alto desempenho contemplando alguns tipos de arquiteturas, incluindo algumas heterogêneas, garantindo a portabilidade em uma grande porção do código mas mantendo o desempenho e a capacidade de fazer otimizações específicas a cada arquitetura. Implementamos a biblioteca HLIB que gerencia as primitivas de arquiteturas heterogêneas do tipo CPU+GPU, APU e CPU+Phi e que também funciona em arquiteturas homogêneas tradicionais. Implementamos o OpenVec, uma ferramenta para gerar, de forma portável, código vetorial explícito. Contemplando as principais arquiteturas SIMD dos últimos 17 anos, tais como ARM Neon, Intel SSE até AVX-512 e IBM VSX. Demonstramos o uso combinado dessas duas ferramentas com aplicações de alto desempenho, que demandam mais de um petaflop.

PUC-Rio - Certificação Digital Nº 1421599/CA

Palavras–chave

Computação de alto desempenho; Arquiteturas heterogêneas; Vetorização explícita.

Abstract

Souza Filho, Paulo Roberto Pereira de; Rodriguez, Noemi de La Rocque (Advisor). **Support for Code Portability in High Performance Computing Applications**. Rio de Janeiro, 2016. 117p. MSc. Dissertation — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Today's platforms are becoming increasingly heterogeneous. A given platform may have many different computing elements in it: CPUs, coprocessors and GPUs of various kinds. This work propose a way too keep some portion of code portable without compromising the performance along different heterogeneous platforms. We implemented the HLIB library that handles the preparation code needed by heterogeneous computing, also this library transparently supports the traditional homogeneous platform. To address multiple SIMD architectures we implemented the OpenVec, a tool to help compiler to enable SIMD instructions. This tool provides a set of portable SIMD intrinsics and C++ operators to get a portable explicit vectorization, covering SIMD architectures from the last 17 years like ARM Neon, Intel SSE to AVX-512 and IBM Power8 Altivec+VSX. We demonstrated the combination use of this strategy using both tools with petaflop HPC applications.

PUC-Rio - Certificação Digital Nº 1421599/CA

Keywords

High performance computing (HPC); Heterogeneous architectures; Explicit vectorization; SIMD; CUDA; OpenCL; hStreams.

Sumário

1 Introdução **12**
1.1 Organização 14

2 OpenVec **15**
2.1 Implementação 20
2.1.1 O Tipo Vetorial 21
2.1.2 Alinhamento e Alocação de Memória 22
2.1.3 Intrínsecos Vetoriais 23
2.1.4 Vetorização de código condicional 27
2.1.5 Otimização da vetorização de código condicional 31
2.1.6 Tratamento da cauda dos vetores 33
2.1.7 Reduções 34
2.2 Discussão 35

3 Biblioteca de Arquiteturas Heterogêneas **37**
3.1 Programação de Arquiteturas Heterogêneas 37
3.2 Implementação 43
3.2.1 Inicializar / finalizar dispositivo 44
3.2.2 Alocar / desalocar memória no dispositivo 45
3.2.3 Memória para transferências entre CPU e dispositivo 47
3.2.4 Zerar memória alocada no dispositivo 48
3.2.5 Cópia de memória CPU⇔dispositivo e dispositivo⇔dispositivo 48
3.2.6 Filas de execução concorrentes 50
3.2.7 Passagem de Mensagem entre Dispositivos 54
3.3 Avaliação 55
3.4 Discussão 56

4 Aplicações de Alto Desempenho **58**
4.1 Migração Reversa no Tempo RTM 58
4.2 Inversão Completa do Campo de Onda FWI 61
4.3 Observações 62
4.4 Exemplo de Uso HLIB 63
4.5 Exemplo de Uso OpenVec 65
4.6 Avaliação de Desempenho OpenVec 67
4.7 Uso Combinado HLIB + OpenVec 68
4.8 Experimentos 69
4.9 Discussão 71

5 Conclusões **72**

6 Referências Bibliográficas **75**

A Documentação OpenVec **84**
A.1 Como Compilar Código OpenVec 84
A.2 Operadores 84

PUC-Rio - Certificação Digital Nº 1421599/CA

A.3 Constantes 85
A.4 Funções 86

B Documentação HLIB **108**
B.1 Sub-rotinas 108

PUC-Rio - Certificação Digital Nº 1421599/CA

Lista de figuras

1.1 Aplicação de alto desempenho em três camadas. 13

2.1 Evolução dos produtos Intel para desktop. A largura SIMD se refere ao número de elementos de precisão simples por operação. Em 1996 existia apenas um núcleo e um *float* por instrução, em 2016 vários núcleos e 16 *floats* por instrução. Extraímos os dados da página ark.intel.com. Extrapolamos o *clock* e o número de núcleos para 2016. 16
2.2 (a) Adição de vetores escalar. (b) Adição de vetores em uma unidade SIMD de 4 elementos. Definimos o número de iterações do *loop* com `n=40` que é múltiplo da largura da unidade SIMD. 17
2.3 Formas implícitas e explícitas de vetorização de código. 18
2.4 Aplicação livre de chamadas proprietárias utilizando o OpenVec, com seus diversos *backends*. 19
2.5 Mapeamento dos elementos `float` no array `x[ ]` do tipo `ov_float`, em uma arquitetura com 4 floats por vetor SIMD. 20
2.6 Operações intermediárias na vetorização de código condicional. Neste exemplo, a condição do `if` é verdadeira apenas para o segundo e o terceiro elementos. 28
2.7 Criação da máscara de bits em um processador com uma unidade SIMD com largura `OV_FLOAT_WIDTH=4`. 30

3.1 Esquema de conexão em uma arquitetura heterogênea CPU+CPU. Desempenho de conexão baseado na seguinte configuração: CPU Xeon E5-4669 v3, GPU Nvidia K80 conectada via PCIe 16X de 3ª geração. 37
3.2 Comparativo da evolução do poder computacional em GFlops/s entre Nvidia e Intel. Computação heterogênea em verde, computação homogênea em azul. 38
3.3 Comparativo entre o grau de portabilidade e desempenho potencial. OpenCL otimizado se refere a múltiplos *kernels* OpenCL otimizados para cada arquitetura. O desempenho é potencial, fica a cargo do desenvolvedor. 41
3.4 Aplicação em azul, biblioteca heterogênea em vermelho e as bibliotecas *runtime* de cada arquitetura em verde. 42
3.5 A HLIB implementa uma interseção de funcionalidades de diversas APIs. 43
3.6 Uso do dispositivo ao longo do tempo. Execuções de *kernel* em azul, cópias de memória em verde e nada sendo executado em vermelho. (a) um exemplo de execução onde o tempo de cópia é menor que o tempo do *kernel*. (b) um exemplo de execução onde o tempo de cópia é igual ao tempo do *kernel*. 53
3.7 Uso das unidades de execução do dispositivo ao longo do tempo. Execuções de *kernel* em azul, cópias de memória em verde e nada sendo executado em vermelho. 53

PUC-Rio - Certificação Digital Nº 1421599/CA

3.8 Comportamento de dois HLIB_MPI_sendRecv para trocar as bordas com dois processos MPI vizinhos. Eixo do tempo na horizontal. Execuções de *kernel* em azul, cópias de memória em verde, comunicação MPI em amarelo e nada sendo executado em vermelho. Mensagem quebrada em 6 pedaços. e_1=$pedaço_1$ da esquerda do processo corrente, ev_1=$pedaço_1$ do vizinho da esquerda, d_1=$pedaço_1$ da direita do processo corrente, vd_1=$pedaço_1$ do vizinho da direita. 55

4.1 Levantamento sísmico marítimo. O navio puxa cabos com os sensores (hidrofones). Abaixo do casco do navio temos a fonte sísmica que é um canhão de ar comprimido. Parte da energia que a fonte produz é refletida pelas camadas geológicas do fundo do mar. 59
4.2 Estêncil para calcular derivadas espaciais de 16ª ordem. O cálculo da derivada no ponto central (em vermelho) depende dos pontos vizinhos (em azul). 60
4.3 Fluxograma do método FWI. 62
4.4 Troca de bordas do algoritmo de propagação de onda 3D por diferenças finitas, que é utilizado pelas aplicações RTM e FWI. Em verde claro os pontos das bordas necessários aos vizinhos do $processo_i$. Os pontos do centro que não são dependência para nenhum outro processo, em verde escuro. O $processo_i$ invoca HLIB_MPI_sendRecv duas vezes, conforme as setas. 64
4.5 (a) um estêncil de 25 pontos (RAIO=4), obtemos o ponto central em vermelho com uma soma ponderada de todos os pontos. (b) 4 estêncis computados simultaneamente em uma arquitetura SIMD com 4 elementos por vetor. 65
4.6 Cálculo simultâneo de 16 elementos em uma arquitetura SIMD com 16 elementos por vetor, como o Intel Xeon Phi. 67
4.7 Desempenho do estêncil de 16ª ordem com OpenVec+OpenMP e com o código de referência, que utiliza intrínsecos AVX-2 e OpenMP. Ambiente do teste: processador Xeon E5-2697v3 @ 2.60 GHz com 14 núcleos, compilador da Intel versão 15.0.1.133 e gcc 4.4.7. 68
4.8 Camadas da RTM com HLIB+OpenVec. A RTM utiliza a HLIB para fazer o gerenciamento da computação heterogênea. Codificamos os *kernels* das arquiteturas hStreams e regular com o OpenVec. 68
4.9 Camada de arquitetura da RTM. As setas indicam a direção de invocação. Por exemplo, o *driver* do código regular invoca o *kernel* OpenVec. 69

PUC-Rio - Certificação Digital Nº 1421599/CA

Lista de tabelas

2.1 Largura das unidades SIMD em cada arquitetura suportada pelo OpenVec. 22
2.2 Intrínsecos OpenVec para precisão simples. 26
2.3 Intrínsecos de comparação para precisão simples e operadores relacionais (apenas C++). Esses intrínsecos retornam uma máscara de bits com os bits ligados nos elementos onde a comparação é verdadeira. 30
2.4 Intrínsecos de redução para máscaras de comparação entre objetos do tipo `ov_float`. Esses intrínsecos também estão disponíveis para precisão dupla (sufixo `"d"`). 31
2.5 Intrínsecos de redução para o testar se os elementos de um objeto `ov_float` são negativos. Esses intrínsecos também estão disponíveis para precisão dupla (sufixo `"d"`). 32
2.6 Funções de redução matemática, com $n =$ `OV_FLOAT_WIDTH-1`. O OpenVec implementa as mesmas funções para precisão dupla com o sufixo d. 35

3.1 Primitivas HLIB da computação heterogênea. 40
3.2 Inicialização interna da HLIB em diferentes arquiteturas. 45

4.1 Estimativa da quantidade de operações de ponto flutuante para uma migração RTM. O total de operações é o produto das demais linhas da tabela. As dimensões comprimento, largura e profundidade compreendem um volume discretizado em uma grade regular 3D. 61
4.2 Experimentos com diversos dispositivos e APIs. A primeira coluna se refere ao número do experimento, e a última ao ano de lançamento do dispositivo. 70

A.1 Flags de compilação para diferentes compiladores e arquiteturas. 84

PUC-Rio - Certificação Digital Nº 1421599/CA

1
Introdução

As aplicações de alto desempenho podem comprometer orçamentos de dezenas de milhões de reais entre aquisições de supercomputadores e gastos operacionais com CPDs. O maior supercomputador divulgado, o Tianhe-2 com capacidade de 33.86 petaflops[1], tem um custo estimado de US$ 390 milhões e consome aproximadamente US$ 24 milhões por ano com energia elétrica[1].

Os desenvolvedores das aplicações que executam neste ambiente de super computação têm a difícil decisão de escolher o nível de otimização de código a ser usado, pois um ganho de desempenho, mesmo que pequeno, pode trazer uma grande economia de recursos. Em alguns casos, essa decisão envolve a escolha da linguagem de programação antes do início do projeto. Muitas opções de otimização são limitadas a um determinado fabricante ou arquitetura e normalmente não são portáveis a outros tipos e modelos de supercomputadores.

Nos últimos anos, as arquiteturas heterogêneas, que utilizam a CPU em conjunto com um acelerador, ganharam espaço na lista dos 500 maiores supercomputadores do mundo[3], sendo que aparecem nos dois primeiros lugares nessa lista. Esse tipo de arquitetura torna ainda mais difícil a decisão do desenvolvedor de definir o nível de otimização a ser aplicado em um determinado código.

Determinados níveis de otimização deixam o código menos legível e dependente de um determinado fornecedor de *hardware*. Com isso fica mais difícil adicionar novas características e mover o código para outras arquiteturas e outros fornecedores. Além disso, no caso de arquiteturas heterogêneas, existe uma complexidade adicional de manter dois códigos: um para arquitetura homogênea tradicional e outro para a arquitetura heterogênea. Por exemplo, ao escolher codificar em Fortran ou C, o desenvolvedor garante que seu código irá abranger a maior parte dos supercomputadores. Mas o desempenho em uma arquitetura heterogênea com GPUs Nvidia fica muito distante do desempenho potencial de se codificar diretamente em CUDA[4]. Por outro lado, ao utilizar CUDA o desenvolvedor limita seu código a apenas um tipo de arquitetura.

Acreditamos que a estratégia de dividir a aplicação em camadas ajuda a manter uma parte do código portável, por exemplo. Propomos dividir a aplicação de alto desempenho heterogênea em três camadas: uma camada superior agnóstica de arquitetura, uma camada com as primitivas de gerenciamento da computação heterogênea e outra camada com o código que efetivamente executa no dispositivo secundário, que em geral é a porção de código da aplicação

[1]Custos de energia elétrica baseados no mercado dos EUA[2].

PUC-Rio - Certificação Digital Nº 1421599/CA

que se deseja acelerar com o uso de um dispositivo secundário, conforme a Figura 1.1.

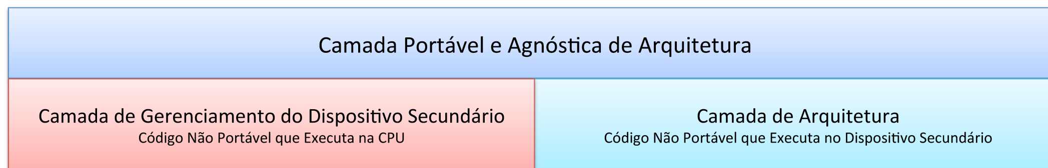

Figura 1.1: Aplicação de alto desempenho em três camadas.

O objetivo deste trabalho é prover ferramentas para o auxílio à portabilidade de código em aplicações de alto desempenho com o foco no desempenho, e propor uma estratégia para organizar essas aplicações em três camadas bem distintas. Uma das nossas ferramentas implementa toda a camada de gerenciamento (em vermelho) da Figura 1.1. Ao dividir a aplicação em três camadas o desenvolvedor isola suas otimizações específicas de cada arquitetura na camada de arquitetura, onde cada arquitetura fica isolada em arquivos fonte distintos. Nossa ferramenta implementa a camada de gerenciamento, e o resto do código da aplicação fica isolado em uma outra camada portável e agnóstica de arquitetura.

Implementamos duas ferramentas que trazem um balanceamento entre portabilidade e desempenho.

A primeira ferramenta explora o uso das unidades vetoriais dos processadores, como a unidade do processador Xeon Phi do supercomputador Tianhe-2. Essa ferramenta habilita o desenvolvedor a expor seu código de uma forma que o compilador produza instruções vetoriais.

As unidades vetoriais são capazes de processar vários elementos simultaneamente em paralelo com apenas uma instrução. Esse tipo de unidade é chamada SIMD (*single instruction multiple data*) e é utilizada por CPUs, APUs e GPUs. A maioria dos supercomputadores atuais possui unidades SIMD.

O modo de expor paralelismo do tipo SPMD (*single program multiple data*) entre diferentes nós e entre diferentes núcleos está bem definido, com padrões abertos como o MPI e o OpenMP. Mas atualmente não existe um padrão aberto para expressar um paralelismo do tipo SIMD de forma explícita. Nossa ferramenta provê funções vetoriais SIMD portáveis que são mapeadas diretamente para funções SIMD específicas de cada arquitetura. Ao utilizar nossa ferramenta de vetorização explícita, o desenvolvedor não se refere a característica alguma da unidade SIMD. Com isso, o mesmo código pode ser compilado para qualquer largura SIMD, inclusive para uma arquitetura escalar com apenas um elemento por instrução.

PUC-Rio - Certificação Digital Nº 1421599/CA

A segunda ferramenta que implementamos auxilia o desenvolvedor na programação de arquiteturas heterogêneas como CPU+GPU. Na computação heterogênea o desenvolvedor pode gastar muito tempo em um código que prepara o dispositivo para execução. Esse código de preparação é o responsável por tarefas como inicializar o dispositivo, alocar e povoar sua memória e enfileirar tarefas a serem executadas no dispositivo. Nossa ferramenta é uma biblioteca com primitivas básicas da computação heterogênea. Essas primitivas implementam de forma portável a porção de código que faz a preparação dos dispositivos secundários. Ao utilizar nossa biblioteca o desenvolvedor pode focar no desenvolvimento do código que efetivamente executa no dispositivo. Essa ferramenta trata a arquitetura homogênea como uma arquitetura heterogênea, permitindo que o desenvolvedor mantenha apenas uma lógica de gerenciamento que contempla arquiteturas homogêneas e heterogêneas.

Como o desenvolvedor somente utiliza chamadas específicas/proprietárias no código que executa no dispositivo, esse código fica naturalmente isolado em uma camada de *software*, que chamamos de *camada de arquitetura*. Com isso, o resto do código da aplicação fica neutro em relação à arquitetura, isolado em outra camada de *software* que invoca a nossa biblioteca.

Para demonstrar o uso desta estratégia com as duas ferramentas combinadas, discutimos a implementação da aplicação de migração reversa no tempo[5]. As equipes de processamento sísmico utilizam essa aplicação para gerar uma imagem de subsuperfície que auxilia a tomada de decisão do posicionamento de um poço de petróleo, que pode custar US$ 50 milhões. Em áreas geológicas complexas como o pré-sal, essa migração posiciona os eventos geológicos de forma mais correta que outras migrações que demandam menos poder computacional. Uma única execução da migração reversa no tempo pode ocupar uma máquina de um petaflop por três meses.

PUC-Rio - Certificação Digital Nº 1421599/CA

1.1 Organização

Esta dissertação está organizada da seguinte forma. Nos Capítulos 2 e 3 apresentamos nossas duas ferramentas de auxílio à portabilidade de código em aplicações de alto desempenho. No Capítulo 4 apresentamos aplicações de alto desempenho que demandam grandes recursos computacionais, e discutimos o uso combinado das nossas ferramentas nessas aplicações. Em seguida, concluímos esta dissertação no Capítulo 5. No final do texto, nos Anexos A e B disponibilizamos a documentação das duas ferramentas que implementamos.

2
OpenVec

Há mais de uma década os fabricantes de processadores não podem mais aumentar o desempenho dos processadores simplesmente aumentando a frequência: o *power wall*[6], um limite de potência por área, inviabilizou o aumento da frequência de operação dos processadores. O aumento indiscriminado da frequência levaria a temperaturas insustentáveis. Desde 2003 os processadores não dependem do aumento da frequência de operação como fonte primária de aumento de desempenho. Eles ficam mais largos a cada nova geração, podendo executar mais operações simultaneamente.

Esse aumento na largura do processador pode ocorrer de várias formas, sendo que as principais são o aumento do número de núcleos, como por exemplo um processador *quad-core*, unidades vetoriais SIMD (*single instruction multiple data*), como a unidade Neon do processador ARM que podemos encontrar em *smartphones*, e o uso de múltiplas unidades de execução em paralelo como *load, store, integer* e SIMD.

Do ponto de vista do programador, duas tendências ficam claras[7]: aplicações *multi-thread* e paralelismo ao nível de instrução ILP (*instruction level paralelism*) com instruções SIMD. Existem várias formas populares e bem estabelecidas de explorar os diversos núcleos de processamento, incluindo padrões abertos, tais como MPI, OpenMP, TBB, pThreads e OpenCL. Entretanto não existe nenhum padrão maduro ou amplamente utilizado para explorar a vetorização de maneira explícita. Recentemente o padrão OpenMP 4.0 trouxe algumas características dedicadas a vetorização. Apesar de alguns compiladores fazerem um bom trabalho de vetorização automática e da notação de *array* presente em algumas linguagens como Fortran 90, bem como das extensões de anotação de código presente em alguns compiladores, não é possível para o programador mapear diretamente instruções vetoriais de forma portável. A notação de *array*, apesar de portável e padrão, fornece apenas uma dica ao compilador de que aquelas operações podem ser vetorizadas.

Apesar do modelo de programação de múltiplos núcleos estar muito mais maduro e padronizado, nos últimos quinze anos o aumento no desempenho dos sistemas de computação se deu, tanto pelo uso de unidades SIMD, como pelo aumento do número de núcleos. A Figura 2.1 mostra a evolução dos processadores Intel para *desktop* com relação ao *clock*, número de núcleos e a largura da unidade vetorial SIMD (capacidade de processar mais de um elemento por instrução).

As arquiteturas como as GPUs, que privilegiam unidades SIMD em detrimento das outras unidades, têm um desempenho por Watt maior em

PUC-Rio - Certificação Digital Nº 1421599/CA

relação às CPUs tradicionais, que gastam mais espaço de silício com caches e execução fora de ordem.

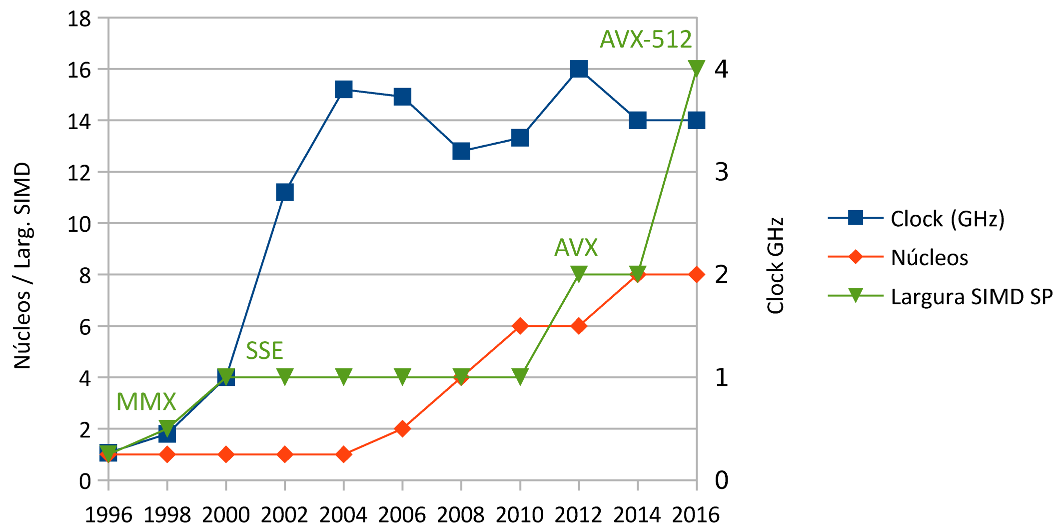

Figura 2.1: Evolução dos produtos Intel para desktop. A largura SIMD se refere ao número de elementos de precisão simples por operação. Em 1996 existia apenas um núcleo e um *float* por instrução, em 2016 vários núcleos e 16 *floats* por instrução. Extraímos os dados da página ark.intel.com. Extrapolamos o *clock* e o número de núcleos para 2016.

A Intel lançou as tecnologias MMX e SSE antes de lançar seu processador *dual-core*. A unidade SIMD MMX processa dois *floats* simultaneamente e a SSE quatro *floats*. O processador Intel Haswell, lançado em 2013, dobrou o número de unidades SIMD por núcleo em relação à geração anterior, sendo capaz de processar 2x8 FMAs (*fused multiply and add)* simultaneamente por núcleo. A Intel anunciou para 2016 o lançamento da unidade vetorial AVX-512 com 16 elementos por instrução.

Como o *power wall* limita o aumento de frequência de operação dos processadores, imaginamos, que cada vez mais, o aumento de desempenho venha da utilização de instruções vetoriais. Por exemplo, um processador Intel Core i7-5960X é capaz de processar 128 (8x2x8) *floats* simultaneamente com seus 8 núcleos, cada um com duas unidades vetoriais de 8 elementos (AVX 2). Atualmente a maioria dos processadores possui unidades SIMD, incluindo os processadores de *smartphones* e *tablets*.

Para extrair todo o potencial desse tipo de processador, é necessário que o compilador gere instruções vetoriais[8]. Embora se tenha avançado nas técnicas de vetorização automática[9, 10, 11, 12, 13, 14, 15, 16, 17], os compiladores ainda não vetorizam alguns códigos[18, 19, 9, 8, 20]. Uma das opções para solucionar esse problema é o uso de SIMD *intrinsics*[21, 22, 23, 24], que são funções fornecidas pelo compilador que são mapeadas diretamente em uma

PUC-Rio - Certificação Digital Nº 1421599/CA

ou mais instruções SIMD do processador. Neste trabalho utilizaremos o termo *intrínsecos* para descrever SIMD *intrinsics*.

Diversos trabalhos reportam ganhos de desempenho com a utilização de intrínsecos[25, 26, 27, 28, 29, 30, 31, 32, 33]. O grande problema dessa abordagem é que esses intrínsecos são específicos para cada arquitetura. Um código que utiliza intrínsecos Intel SSE, por exemplo, não extrai todo o potencial de um processador Intel AVX, apesar de ambos serem desenvolvidos pelo mesmo fabricante, e um código AVX não executa em um processador SSE. Com isso, uma recodificação é necessária para cada arquitetura.

Para exemplificar, vamos discutir o caso de uma adição simples de vetores, como a mostrada na Figura 2.2a. Podemos mapear a adição de vetores da Figura 2.2a para uma instrução SIMD que soma vários elementos simultaneamente em paralelo, conforme a Figura 2.2b.

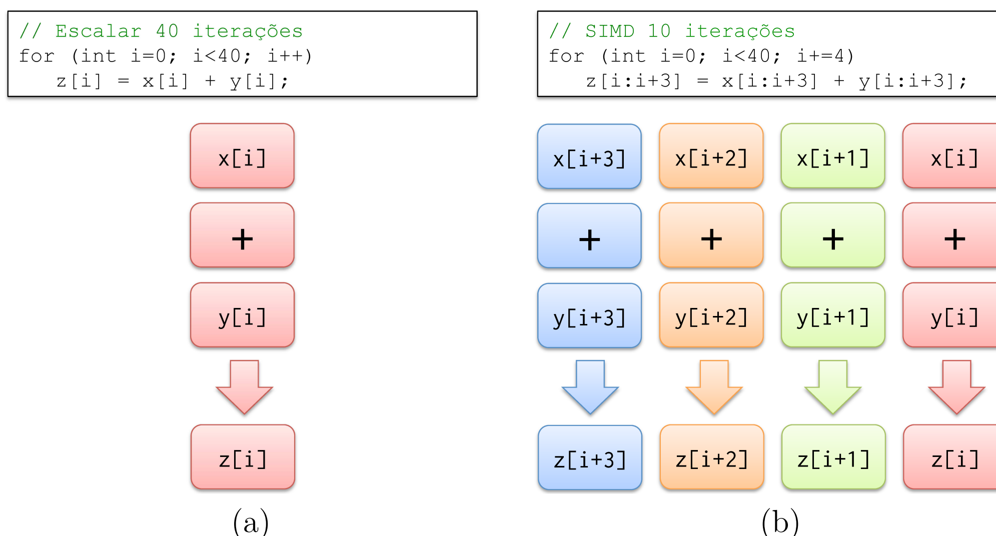

Figura 2.2: (a) Adição de vetores escalar. (b) Adição de vetores em uma unidade SIMD de 4 elementos. Definimos o número de iterações do *loop* com `n=40` que é múltiplo da largura da unidade SIMD.

Atualmente a unidade SIMD com mais elementos pertence ao processador Xeon Phi, com 16 elementos por instrução. Essa largura de 16 operandos por instrução habilita um ganho teórico de desempenho de 16 vezes, tendo em vista que cada instrução processa 16 elementos de cada vez. Na prática é difícil obter 100% do ganho teórico pois a velocidade da memória pode não ser capaz de alimentar todos esses operandos simultâneos. De forma simplificada, a relação do desempenho atingido em relação ao pico teórico fica limitada pela *intensidade computacional* da aplicação. Williams[31] define a intensidade computacional como a razão entre a quantidade de operações de ponto flutuante e a quantidade de acessos à memória, utilizando a unidade $\frac{flops}{word}$.

PUC-Rio - Certificação Digital Nº 1421599/CA

Para lidar com a falta de portabilidade dos intrínsecos proprietários, desenvolvemos o **OpenVec**[34, 35], que apresentamos a seguir. O OpenVec é uma ferramenta para geração explícita de código vetorial SIMD para ser aplicada nos *hot spots* de uma aplicação, com o objetivo de obter um desempenho similar ao codificado com intrínsecos proprietários mas mantendo a aparência e a portabilidade de um código de alto nível.

A Figura 2.3 mostra as diversas formas de vetorizar um código. Algumas formas são implícitas, como por exemplo o uso de diretivas (`#pragma`) para guiar o compilador na vetorização de um *loop*.

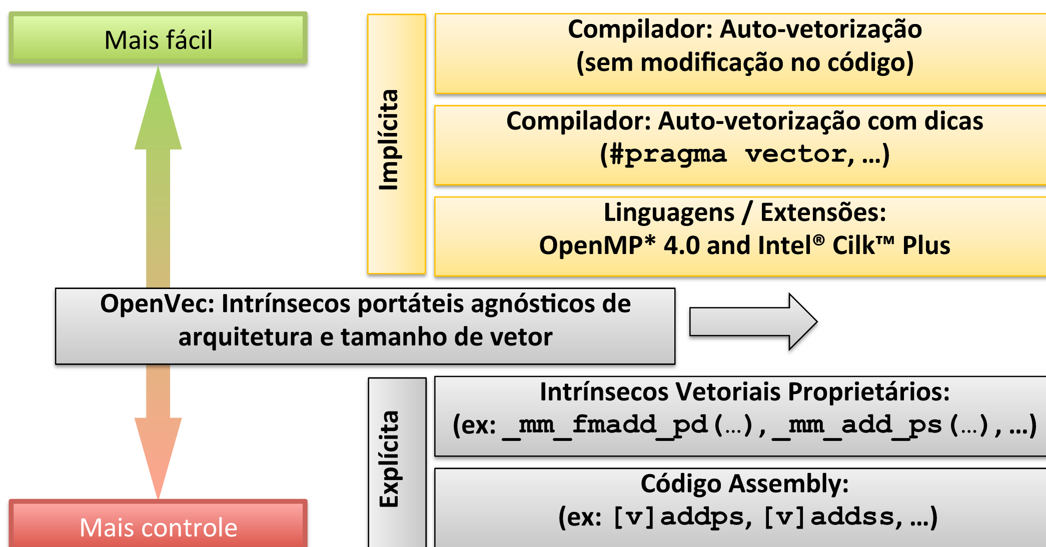

Figura 2.3: Formas implícitas e explícitas de vetorização de código.

O OpenVec define um tipo vetorial e o suporte aos intrínsecos de hardware para as linguagens C e C++, através de macros e sobreposição de operadores (*operator overload*) no caso de C++. Ele mapeia as APIs proprietárias de cada arquitetura/compilador para uma API comum e portável.

Também geramos código escalar para os casos onde a arquitetura não possui instruções SIMD ou para uma arquitetura não suportada. Com isso garantimos que um código OpenVec seja compilado em qualquer arquitetura por qualquer compilador padrão C/C++. O *backend* escalar do OpenVec é o responsável por gerar esse código.

O OpenVec tem como público alvo desenvolvedores que desejam utilizar intrínsecos SIMD para acelerar *hot spots* em diversos tipos e gerações de processadores mantendo o reuso e a portabilidade de código. Testamos nossa implementação em diversos processadores - ARM Neon, Intel SSE/AVX/AVX-512/IMCI e IBM Power8 VSX - e com diversos compiladores - gcc, llvm e Intel. Nossa implementação utiliza apenas *header files* com o preprocessador

PUC-Rio - Certificação Digital Nº 1421599/CA

C e no caso de C++ também utilizamos sobreposição de operadores. Também testamos o backend escalar que funciona com qualquer compilador.

A Figura 2.4 mostra a aplicação em uma camada livre de chamadas proprietárias e os diversos *backends* do OpenVec. Organizamos a implementação de cada arquitetura (*backend*) em *header files* separados. O *header file* principal `openvec.h` define o *backend* em tempo de compilação. O desenvolvedor usuário do OpenVec inclui em seu código apenas o *header file* principal.

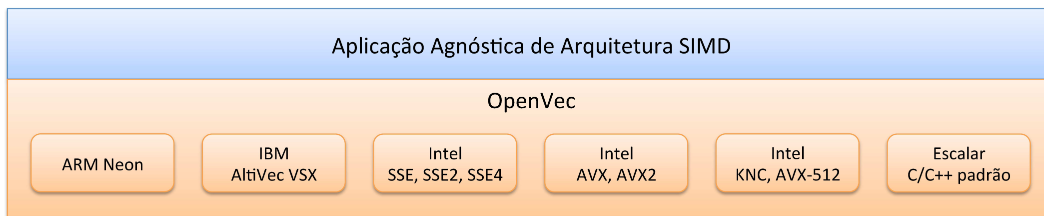

Figura 2.4: Aplicação livre de chamadas proprietárias utilizando o OpenVec, com seus diversos *backends*.

A Listagem 2.1 mostra a função SAXPY da BLAS, implementada de forma escalar e com o OpenVec. Essa função multiplica elemento a elemento um vetor por um escalar, e soma o resultado com um outro vetor.

Listagem 2.1: SAXPY escalar e com OpenVec C++ e C.

```
#include"openvec.h"


// SAXPY Escalar
void saxpy(int n, float a, float *x, float *y)
{
  for (int i=0; i<n; i++)
     y[i] = a*x[i] + y[i];
}




// SAXPY OpenVec C++
void vsaxpy(const int n, const float a, const ov_float* x,
            ov_float* y)
{
  int const nv=(n-1)/OV_FLOAT_WIDTH; // numero de vetores SIMD
  for (int i=0; i<=nv; i++)
     y[i] = a*x[i] + y[i];
}
```

PUC-Rio - Certificação Digital Nº 1421599/CA

```
/* SAXPY OpenVec C */
void vsaxpy(const int n, const float a, const ov_float* x,
            ov_float* y)
{
  int const nv=(n-1)/OV_FLOAT_WIDTH;/* numero de vetores SIMD*/
  /* promove o escalar a para vetor */
  ov_float const va=ov_setf(a);
  for (int i=0; i<=nv; i++)
     /* invoca intrinseco multiply and add */
     y[i] = ov_maddf(va, x[i], y[i]);
}
```

Na Listagem 2.1, os arrays `x[ ]` e `y[ ]` são declarados com o tipo vetorial `ov_float` que contém, por exemplo, 4 floats (128 bits) em um processador IBM Power 8. Cada *backend* implementa a constante `OV_FLOAT_WIDTH`, que define a largura da unidade SIMD para `floats` (precisão simples).

A Figura 2.5 mostra o mapeamento dos vinte primeiros elementos do array de floats da Listagem 2.1 em um array de vetores SIMD do tipo `ov_float`.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
x[0]				x[1]				x[2]				x[3]				x[4]			

Figura 2.5: Mapeamento dos elementos `float` no array `x[ ]` do tipo `ov_float`, em uma arquitetura com 4 floats por vetor SIMD.

O *header file* `openvec.h`. define a operação `a*x[i]` (multiplicação de escalar por vetor) da Listagem 2.1 com um *overload* de operador C++.

Podemos observar na Listagem 2.1 que a versão do código que utiliza C é mais complexa que a versão C++. Em C, precisamos promover o escalar para vetor e invocar um intrínseco para efetuar a multiplicação e soma.

A Seção 2.1.6 trata o caso onde o número de elementos do *loop* (variável `n` da Listagem 2.1) não é múltiplo da largura do vetor SIMD.

2.1 Implementação

Implementamos o OpenVec apenas com macros padrão do preprocessador C, garantindo com isso o mínimo de *overhead* pois os intrínsecos OpenVec são substituídos por intrínsecos proprietários do compilador pelo preprocessador C. Em C++ também utilizamos a sobreposição de operadores. O OpenVec não é uma biblioteca, e sim um *header file* que mapeia os intrínsecos proprietários do compilador para uma nomenclatura comum e agnóstica de arquitetura.

Criamos *backends* para as arquiteturas SIMD a seguir:

PUC-Rio - Certificação Digital Nº 1421599/CA

- ARM Neon
- Intel SSE/SSE2/SSE4/AVX/AVX-2/AVX-512/IMCI
- IBM Power8 VSX
- Escalar, compatível com qualquer compilador C/C++

2.1.1 O Tipo Vetorial

O OpenVec define tipos vetoriais que o compilador mapeia para registradores vetoriais. Esses tipos contém múltiplos elementos e seu tamanho varia conforme a largura SIMD de cada arquitetura. Definimos a constante `OV_FLOAT_WIDTH` para cada *backend* com o número de valores de ponto flutuante da unidade SIMD (precisão simples) do processador. No *backend* escalar `OV_FLOAT_WIDTH`=1.

O tipo `ov_float` define uma variável que contém `OV_FLOAT_WIDTH` valores de ponto flutuante em precisão simples (`floats`). Por exemplo, em um processador ARM Neon o tipo `ov_float` contém 4 floats, em um processador Intel AVX `ov_float` contém 8 floats. Utilizamos esse tipo para declarar variáveis da mesma forma que os tipos nativos, como `float` e `double`. A seguir um exemplo de declaração de uma variável vetorial de `floats`:

```
ov_float minhaVar;
```

Cada arquitetura define um tipo vetorial diferente, e cada uma possui uma largura SIMD. Por exemplo, em um processador ARM Neon mapeamos o tipo `ov_float` para o tipo proprietário `float32x4_t` que contém 4 floats, enquanto em um processador Intel AVX mapeamos o mesmo `ov_float` para o tipo `__m256` que contém 8 floats.

Apos instanciar uma variável com o tipo vetorial, todas as operações serão executadas simultaneamente em todos os elementos da variável. Por exemplo, em um processador Intel SSE com `OV_FLOAT_WIDTH`=4, a soma $x+y$ com duas variáveis do tipo `ov_float`, executa em paralelo, somando cada float de x com o respectivo float de y.

Definimos as operações entre variáveis escalares e vetoriais repetindo o valor escalar para todas as posições do vetor. Por exemplo, a soma $x + 1$, com x sendo do tipo `ov_float`, executa em paralelo da seguinte forma em um processador Intel SSE:

$$x_0 + 1; x_1 + 1; x_2 + 1; x_3 + 1;$$

Implementamos operações vetoriais com números de ponto flutuante de 32 e 64 bits, com os tipos `ov_float` e `ov_double` e as respectivas constantes `OV_FLOAT_WIDTH` e `OV_DOUBLE_WIDTH`.

PUC-Rio - Certificação Digital Nº 1421599/CA

A Tabela 2.1 mostra a largura SIMD para todas as arquiteturas suportadas pelo OpenVec.

Arquitetura SIMD	OV_FLOAT_WIDTH	OV_DOUBLE_WIDTH
Intel Xeon Phi AVX-512	16	8
Intel AVX/AVX-2	8	4
Intel SSE2/SSE4	4	2
Intel SSE	4	1
ARM Neon	4	1
IBM Power Altivec	4	1
Escalar	1	1

Tabela 2.1: Largura das unidades SIMD em cada arquitetura suportada pelo OpenVec.

2.1.2 Alinhamento e Alocação de Memória

PUC-Rio - Certificação Digital Nº 1421599/CA

Em todas as arquiteturas SIMD existe uma penalidade de desempenho se uma estrutura não está alinhada na memória. Para executar um `load` ou `store` sem penalidade de desempenho, o endereço base da posição de memória deve estar no alinhamento natural do vetor. Por exemplo, em um processador Xeon Phi, ao carregar uma variável `ov_float` da memória, a posição de memória deve estar alinhada da seguinte forma:

`OV_ALIGN` = `OV_FLOAT_WIDTH x sizeof(float) = 16x4 = 64bytes`

No OpenVec, definimos a constante `OV_ALIGN`, usando o cálculo acima, para indicar qual é a restrição de alinhamento de cada arquitetura.

Implementamos as seguintes funções de alocação de memória que garantem que o primeiro elemento alocado está alinhado em `OV_ALIGN` bytes:

```
void *ov_malloc(size_t size);
void ov_free(void *ptr);
void *ov_calloc(size_t count, size_t size);
```

Uma memória alocada com as funções `ov_malloc` e `ov_calloc` deve ser desalocada somente com a função `ov_free`.

Como o desenvolvedor tem a informação de que o primeiro elemento está alinhado, ele pode, em alguns casos, garantir que os demais acessos à memória serão alinhados, por exemplo, fazendo com que a dimensão contígua em memória de seus arrays seja múltipla de `OV_FLOAT_WIDTH`. Em um array alinhado, o primeiro elemento está sempre alinhado e temos um endereço alinhado a cada `OV_FLOAT_WIDTH` elementos.

Tratar o alinhamento de memória traz um ganho de desempenho[36]. Com o OpenVec é possível informar o compilador se o acesso é alinhado ou não. A Seção 2.1.3 detalha este procedimento.

Para auxiliar o desenvolvedor no caso onde o tamanho do *array* não é múltiplo de `OV_FLOAT_WIDTH`, as funções de alocação adicionam `OV_ALIGN` bytes na quantidade de bytes alocados. Dessa forma o desenvolvedor não precisa testar se o tamanho do array é múltiplo de `OV_FLOAT_WIDTH`, basta deixar o código "invadir" as ultimas posições de memória, como mostra a Listagem 2.1.

Em todos os *backends*, com exceção do escalar, nossa implementação invoca a função `posix_memalign`, que aloca memória alinhada. Por conveniência, o *header file* `openvec.h` substitui as funções padrão `malloc`, `calloc` e `free` pelas funções equivalentes do OpenVec. Dessa forma, o desenvolvedor não precisa modificar o código para obter o beneficio da alocação alinhada. No *backend* escalar, para manter o máximo de compatibilidade, nossa implementação invoca as funções padrão `malloc`, `calloc` e `free`.

2.1.3 Intrínsecos Vetoriais

A programação com intrínsecos vetoriais SIMD requer que o desenvolvedor utilize funções (intrínsecos) que na maioria dos casos se referem a uma instrução SIMD do processador. Além disso o desenvolvedor deve substituir o uso dos tipos básicos como `float` e `double` por tipos vetoriais que contêm múltiplos elementos. A codificação de algumas operações triviais é diferente na programação com intrínsecos, por exemplo, não é possível utilizar um `if-then-else`, que deve ser substituído por um intrínseco. Em alguns casos, para acessar a memória é necessário utilizar intrínsecos de `load` e `store`.

Implementamos diversas funções que mapeiam os intrínsecos proprietários dos compiladores. As chamadas OpenVec normalmente fazem um mapeamento direto utilizando macros, conforme a definição a seguir do nosso *backend* Intel AVX:

```
#define ov_sqrtf _mm256_sqrt_ps
```

Em C++ o uso de algumas funções matemáticas básicas é desnecessário, pois existe o *overload* de operadores, conforme exemplo a seguir:

```
ov_float a,b,c;
// codigo C++ com overload do operador +
c = a + b;
/* codigo C com chamada explicita de adicao */
c = ov_addf(a, b);
```

PUC-Rio - Certificação Digital Nº 1421599/CA

Em um código C/C++ com os tipos básicos da linguagem, o compilador gera automaticamente as operações `load` e `store`. No OpenVec é possível programar com `load` e `store` implícitos (gerados pelo compilador) e explícitos.

O desenvolvedor pode declarar seus arrays com um tipo básico como `float` e utilizar LD/ST explícitos ou mapear os arrays para um tipo vetorial como o `ov_float` resultando em LD/ST implícitos. Todo acesso implícito deve ser alinhado. Essa restrição também existe para os tipos básicos, por exemplo, um acesso a um `float` deve estar alinhado na memória em 4 bytes. Se o desenvolvedor faz um acesso implícito a um tipo básico ou vetorial em um endereço desalinhado, o programa termina com o sinal `SIGSEGV` (*segmentation fault*). A Listagem 2.2, mostra o uso do LD/ST implícito e explícito:

Listagem 2.2: Duas implementações de SAXPY com LD/ST explícitos e implícitos.

```
void saxpy_exp(int n, float a, float *x, float *y)
{
  for (int i=0; i<n; i+=OV_FLOAT_WIDTH)
  {
    ov_float vy = ov_ldf(&y[i]); // Load explicito alinhado
    ov_float vx = ov_ldf(&x[i]); // Load explicito alinhado
    vy = a*vx + vy;
    ov_stf(&y[i], vy);  // Store explicito alinhado
  }
}

void saxpy_imp(int n, float a, ov_float *x, ov_float *y)
{
  int const nv = ((n-1)/OV_FLOAT_WIDTH);
  for (int i=0; i<=nv; i++)
  {
    y[i] = a*x[i] + y[i]; // LD/ST implicitos
  }
}
```

O desempenho dos acessos alinhados é maior que o desempenho dos acessos desalinhados[36]. Mas se o desenvolvedor não pode assumir com segurança que o endereço está alinhado, ele deve utilizar o acesso explícito desalinhado. Se um array do tipo `ov_float` tem o primeiro elemento alinhado, todos os outros elementos `ov_float` do *array* estão alinhados.

Implementamos funções que testam se um endereço está alinhado ou não. Se o desenvolvedor não tem o controle da alocação de memória, ele pode escrever duas versões de código: uma alinhada e outra não. A Listagem 2.3

PUC-Rio - Certificação Digital Nº 1421599/CA

apresenta a invocação de duas versões da função SAXPY. Neste exemplo, o desenvolvedor testa o alinhamento de memória dos *arrays* `x[]` e `y[]`.

Listagem 2.3: Teste de alinhamento de memória.

```
// Se pelo menos um array nao esta alinhado
// Invoca a versao desalinhada
if ((ov_not_alignedf(x) || ov_not_alignedf(y))
{
  saxpy_not_aligned(n, a, x ,y);
}
else // Senao invoca a versao alinhada
{
  saxpy_aligned(n, a, x ,y);
}
```

O desenvolvedor deve tomar cuidado com a semântica do acesso implícito. Por exemplo, no *array* do tipo `ov_float` a indexação `x[i+1]` aponta para o próximo vetor e não para o próximo `float`.

A Tabela 2.2 mostra todas as funções OpenVec para precisão simples, que recebem o sufixo `"f"`. As funções que operam no tipo `ov_double` recebem o sufixo `"d"`.

PUC-Rio - Certificação Digital Nº 1421599/CA

Intrínseco SIMD	Operação / Descrição
`ov_addf(x,y)`	$x + y$
`ov_subf(x,y)`	$x - y$
`ov_mulf(x,y)`	$x \times y$
`ov_divf(x,y)`	$\frac{x}{y}$
`ov_ldf(endereço)`	Load alinhado
`ov_uldf(endereço)`	Load desalinhado
`ov_stf(endereço,x)`	Store alinhado
`ov_storeuf(endereço,x)`	Store desalinhado
`ov_stream_stf(endereço,x)`	Stream store alinhado
`ov_nt_storef(endereço,x)`	Store não temporal alinhado
`ov_setzerof(x)`	Zera todos floats de x
`ov_getzerof()`	Retorna um vetor de zeros
`ov_setf(escalar)`	Retorna um vetor com o *escalar* repetido em todas as posições
`ov_maxf(x,y)`	Retorna o máximo por elemento
`ov_minf(x,y)`	Retorna o mínimo por elemento
`ov_sqrtf(x)`	$\sqrt{x}$
`ov_rsqrtf(x)`	$\frac{1}{\sqrt{x}}$
`ov_sqrf(x)`	x^2
`ov_rcpf(x)`	$\frac{1}{x}$
`ov_floorf(x)`	$\lfloor x \rfloor$
`ov_ceilf(x)`	$\lceil x \rceil$
`ov_maddf(x,y,z)`	$x \times y + z$
`ov_msubf(x,y,z)`	$x \times y - z$
`ov_absf(x)`	$\lvert x \rvert$

Tabela 2.2: Intrínsecos OpenVec para precisão simples.

Alguns intrínsecos OpenVec não têm uma instrução de hardware equivalente em todas as arquiteturas, por isso implementamos alguns intrínsecos OpenVec com um ou mais intrínsecos de hardware, ou em último caso, com código escalar. Por exemplo, implementamos o intrínseco multiplicar e somar `ov_maddf` no *backend* Intel SSE com dois intrínsecos proprietários:

```
#define ov_maddf(a,b,c) _mm_add_ps(c, _mm_mul_ps(a,b))
```

O entendimento das funções matemáticas é trivial, por exemplo a função `ov_sqrtf(x)` calcula em paralelo a raiz quadrada para todos os `floats` de `x`.

Os desenvolvedores que utilizam o OpenVec podem modificar e criar novas funções, baseadas em necessidades de desempenho e precisão. Em

PUC-Rio - Certificação Digital Nº 1421599/CA

alguns *backends* não existe a instrução divisão vetorial, e varias instruções são necessárias para efetuar uma divisão. O exemplo a seguir cria uma divisão customizada, mais rápida e menos precisa que a divisão padrão do OpenVec.

```
#define fast_div(x,y) ov_mulf(x,ov_rcpf(y)) // x × 1/y
```

Os compiladores permitem relaxar a precisão para obtenção de maior desempenho com *flags* de otimização, mas somente a nível de arquivo de código fonte. O OpenVec traz a habilidade de customizar instruções ao nível de operação. Por exemplo, em um código com mil linhas o desenvolvedor pode customizar apenas uma divisão, em uma determinada porção do código.

2.1.4 Vetorização de código condicional

O conceito básico da vetorização é executar a mesma instrução em todos os elementos do vetor, mas em muitos casos existe um caminho divergente causado por um `if-then-else`. Nestes casos é preciso computar o código do `if` e o código do `else` para todos os elementos e juntar o resultado computado de cada código em um vetor.

A Figura 2.6 apresenta as operações intermediárias executadas pelo processador em um código SIMD condicional do tipo `if-then-else`. Em uma etapa inicial, é necessário computar os seguintes operandos da Figura 2.6:

- Máscara de bits (em verde) com os bits ligados nos elementos onde a condição do `if` é verdadeira
- Máscara de bits inversa (em laranja) com os bits ligados nos elementos onde a condição do `if` é falsa (`else`)
- Um vetor contento o resultado do bloco de código do `then` (em azul)
- Um vetor contendo o resultado do bloco de código do `else` (em vermelho)

Para agregar os resultados do `then` e do `else` em um único vetor, aplicamos uma operação `AND` entre a máscara do `if` (em verde) e o resultado do `then` (em azul), esta operação zera os elementos onde a condição do `if` é falsa. Também aplicamos uma operação `AND` entre a máscara inversa (em laranja) e o resultado do `else` (em vermelho), mantendo apenas os elementos onde a condição do `else` é verdadeira. Na última etapa, agregamos os resultados das duas operações `AND` com uma operação `OR`, resultando em um vetor com o `then` aplicado nos elementos onde a condição do `if` é verdadeira e o `else` aplicado nos elementos onde a condição do `if` é falsa.

PUC-Rio - Certificação Digital Nº 1421599/CA

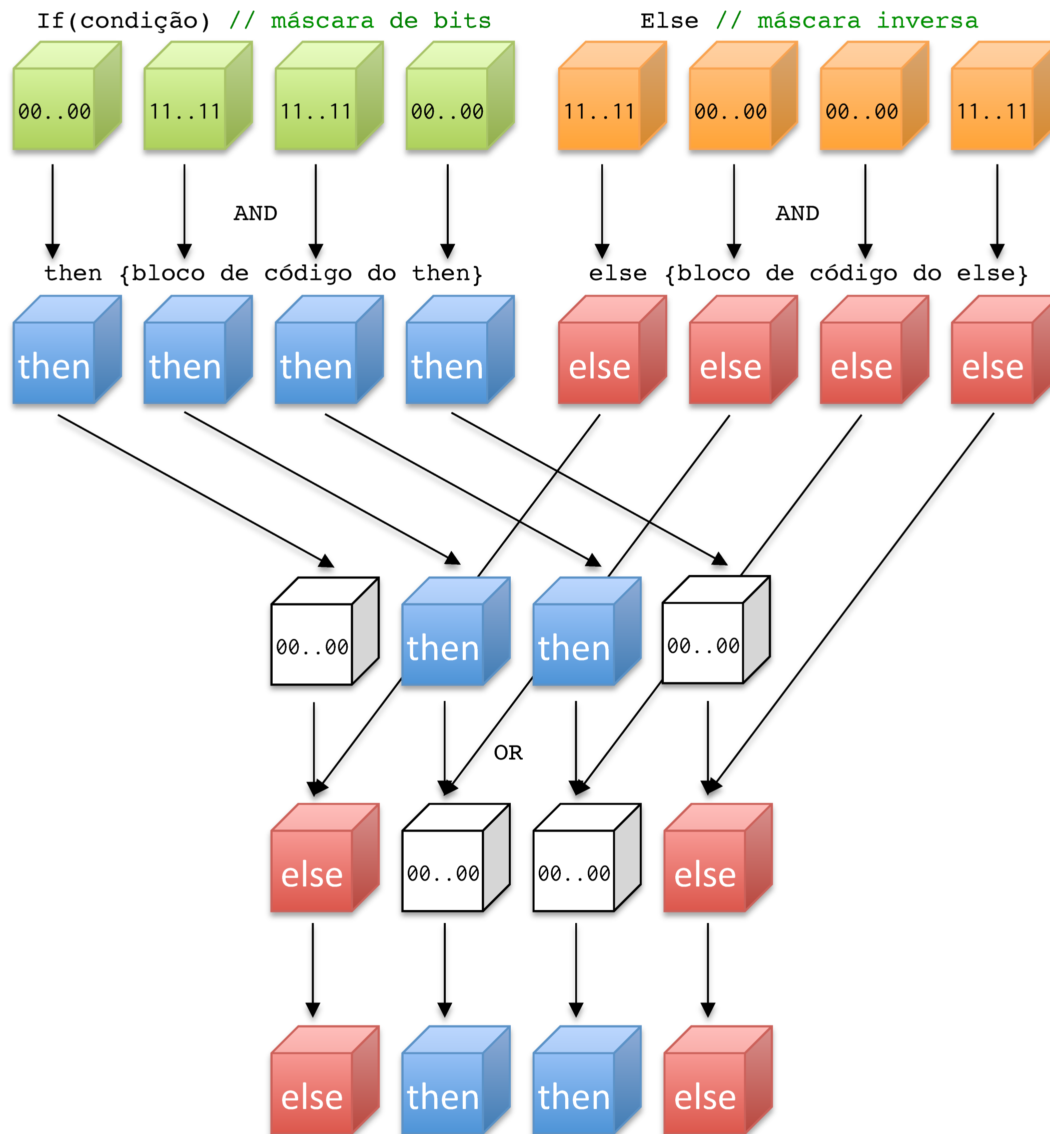

Figura 2.6: Operações intermediárias na vetorização de código condicional. Neste exemplo, a condição do `if` é verdadeira apenas para o segundo e o terceiro elementos.

PUC-Rio - Certificação Digital Nº 1421599/CA

A Listagem 2.4 mostra um simples `if-then-else` em um código escalar e a Listagem 2.5 mostra o código C++ equivalente que vetoriza o `if-then-else` com a função `ov_conditionalf`.

Listagem 2.4: Simples `if-then-else` escalar.

```
void sfunc(int n, float *x, float *y, float *z)
{
  for (int i=0; i<n; i++)
    if (x[i]>y[i]) z[i] = x[i]-y[i];
    else           z[i] = y[i]+x[i];
}
```

Listagem 2.5: Vetorização do `if-then-else` com a função `ov_conditionalf`.

```
void vfunc(int n, ov_float *x, ov_float *y, ov_float *z)
{
  int const nv=(n-1)/OV_FLOAT_WIDTH;
  for (int i=0; i<=nv; i++)
    z[i] = ov_conditionalf(x[i]>y[i], x[i]-y[i], y[i]+x[i]);
}
```

Implementamos em C++ o *overload* de todos os operadores relacionais, conforme a Tabela 2.3. A operação `x[i]>y[i]` da Listagem 2.5 gera a máscara de bits, que é utilizada para juntar os resultados do `if-then-else`.

Com o OpenVec, não podemos codificar comandos `if` diretamente. A função `ov_conditionalf` recebe no primeiro argumento a máscara com os bits ligados para os elementos onde o resultado do teste é verdadeiro (em verde na Figura 2.6). O segundo argumento é a expressão a ser computada no caso do teste ser verdadeiro (*then*) e o terceiro argumento é a expressão a ser computada no caso do teste ser falso (*else*). O retorno dessa função é um vetor com o resultado da expressão '*then*' nos elementos onde o `if` é verdade e com o resultado da expressão '*else*' nos demais elementos.

Nem todos os processadores SIMD possuem uma instrução equivalente ao `ov_conditionalf`. Em algumas arquiteturas, necessitamos de múltiplas instruções para codificar um `if`. Estas instruções executam as operações da Figura 2.6. A seguir apresentamos nossa implementação para algumas arquiteturas:

```
// Backend escalar
#define ov_conditionalf(mask, x, y)  ((mask) ? (x) : (y))

// Backend ARM Neon
#define ov_conditionalf(mask, x, y) \
    vbslq_f32(vreinterpretq_u32_f32(mask), x, y)

// Backends Intel SSE, SSE2, SSE4
#define ov_conditionalf(mask, x, y) \
    _mm_or_ps(_mm256_and_ps(mask, x), _mm_andnot_ps(mask, y))

// Backend Intel AVX
#define ov_conditionalf(mask, x, y) \
    _mm256_blendv_ps(y, x, mask)
```

Em C, como não é possível fazer o *overload* dos operadores de comparação, criamos funções de comparação que retornam a máscara de bits. A Tabela 2.3 mostra essas funções, que também estão disponíveis em C++.

PUC-Rio - Certificação Digital Nº 1421599/CA

Intrínseco de Comparação	Operador Relacional C++	Descrição
`ov_eqf(x,y)`	`==`	$x = y$
`ov_nef(x,y)`	`!=`	$x \neq y$
`ov_gtf(x,y)`	`>`	$x > y$
`ov_gef(x,y)`	`>=`	$x \geq y$
`ov_ltf(x,y)`	`<`	$x < y$
`ov_lef(x,y)`	`<=`	$x \leq y$

Tabela 2.3: Intrínsecos de comparação para precisão simples e operadores relacionais (apenas C++). Esses intrínsecos retornam uma máscara de bits com os bits ligados nos elementos onde a comparação é verdadeira.

A Figura 2.7 mostra em detalhes a criação da máscara pela operação `x>y` (C++) e a função `ov_gtf(x,y)`. A comparação é feita em paralelo, de forma independente para cada elemento dos operandos vetoriais `x,y`.

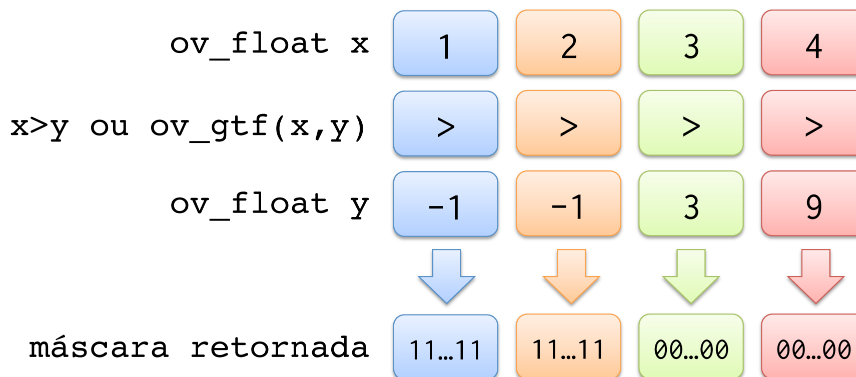

Figura 2.7: Criação da máscara de bits em um processador com uma unidade SIMD com largura `OV_FLOAT_WIDTH=4`.

Os operadores e as funções de comparação da Tabela 2.3 retornam o tipo `ov_maskf` ou `ov_maskd` (precisão dupla). Mapeamos esses tipos para os tipos proprietários de cada arquitetura.

O desenvolvedor não deve assumir que o tamanho do tipo `ov_maskf` é igual ao tamanho do tipo `ov_float`. Em algumas arquiteturas a máscara utiliza apenas um bit por elemento e em outras utiliza a mesma quantidade de bits do tipo que está sendo comparado. Por exemplo, em uma arquitetura Intel SSE uma variável vetorial do tipo `ov_float` possui 128 bits (4x32) e a máscara também possui 128 bits, enquanto a arquitetura Intel AVX-512 utiliza apenas 16 bits para representar uma máscara resultado de uma comparação de 16 *floats* (512 bits).

PUC-Rio - Certificação Digital Nº 1421599/CA

2.1.5 Otimização da vetorização de código condicional

Em alguns casos, em apenas alguns elementos a execução entra no caminho do código do `else` ao invés do `then`. Este comportamento é muito comum em simulações numéricas, onde determinadas operações devem ser aplicadas apenas nas bordas do modelo. Neste caso a vetorização com máscara pode ser muito ineficiente pois ambos os blocos de código do `then` e do `else` são executados. Mapeamos intrínsecos que verificam se todos os elementos de um vetor SIMD vão percorrer o caminho do `then` ou do `else`, para que apenas um dos dois códigos seja executado e não seja necessário aplicar a máscara.

Criarmos duas funções básicas, uma que verifica se todos os bits da máscara estão ligados, e outra que verifica se pelo menos um bit está ligado. Essas funções fazem a redução de todos os elementos da máscara para um valor escalar, que pode ser verdadeiro ou falso, conforme a Tabela 2.4.

Intrínseco de redução	Descrição
`ov_allf(máscara)`	Retorna um valor diferente de zero se a máscara é verdade para todos os elementos
`ov_anyf(máscara)`	Retorna um valor diferente de zero se a máscara é verdade para pelo menos um elemento

Tabela 2.4: Intrínsecos de redução para máscaras de comparação entre objetos do tipo `ov_float`. Esses intrínsecos também estão disponíveis para precisão dupla (sufixo "d").

Com essas funções podemos otimizar a vetorização de códigos condicionais, testando se todos os elementos entram no bloco do `if` ou do `else`, se todos os elementos entram no mesmo caminho de código então não é necessário aplicar a máscara, a Listagem 2.6 mostra a otimização do código vetorial da Listagem 2.5.

Listagem 2.6: Otimização da vetorização do `if-then-else` com a função `ov_allf`.

```
void vfunc(int n, ov_float *x, ov_float *y, ov_float *z)
{
  int const nv=(n-1)/OV_FLOAT_WIDTH;
  for (int i=0; i<=nv; i++)
    if (ov_allf(x[i]>y[i])
      z[i] = x[i]-y[i];
    else
      z[i] = ov_conditionalf(x[i]>y[i], x[i]-y[i], y[i]+x[i]);
}
```

PUC-Rio - Certificação Digital Nº 1421599/CA

No código da Listagem 2.6, se todos os elementos do vetor SIMD `x[i]` são maiores que os respectivos elementos de `y[i]`, então o código do else não é computado e nenhuma máscara é aplicada. Senão, a versão mais lenta com a aplicação da máscara é executada, computando ambos os blocos de código do `if` e do `else`, mostrados na Listagem 2.4.

O tipo de retorno das funções da Tabela 2.4 varia para cada *backend*, mas o desenvolvedor pode assumir que o valor retornado pode ser utilizado na avaliação de um `if`, conforme o exemplo a seguir:

```
if (ov_allf(mascara)) ...
```

Decidimos não incorporar as funções de redução de máscara na função `ov_conditionalf`. Acreditamos que cabe ao o desenvolvedor decidir se o `then` ocorre com mais frequência que o `else`, ou vice versa.

Algumas arquiteturas possuem uma instrução que permite testar se os elementos de um vetor SIMD são negativos ou não, testando apenas o bit do sinal, sem a necessidade de gerar uma máscara de comparação. Implementamos os intrínsecos da Tabela 2.5 que tiram proveito dessa funcionalidade. Esses intrínsecos testam se todos ou se alguns elementos de um vetor SIMD são negativos ou não.

PUC-Rio - Certificação Digital Nº 1421599/CA

Intrínseco de redução	Descrição
`ov_any_lt_0f(x)`	Retorna um valor diferente de zero se pelo menos um elemento de $x < 0$
`ov_any_ge_0f(x)`	Retorna um valor diferente de zero se pelo menos um elemento de $x \geq 0$
`ov_all_lt_0f(x)`	Retorna um valor diferente de zero se todos os elementos de $x < 0$
`ov_all_ge_0f(x)`	Retorna um valor diferente de zero se todos os elementos de $x \geq 0$

Tabela 2.5: Intrínsecos de redução para o testar se os elementos de um objeto `ov_float` são negativos. Esses intrínsecos também estão disponíveis para precisão dupla (sufixo `"d"`).

Nos *backends* onde essas instruções de teste de bit do sinal não existem, implementamos esses intrínsecos com a combinação de uma função de comparação com uma função de redução, conforme o exemplo a seguir:

```
#define ov_any_lt_0f(x) ov_anyf(ov_ltf(x, ov_getzerof()))
```

Como no *backend* escalar só existe um elemento por vetor, as funções de redução condicional do tipo `ov_any` e `ov_all` são implementadas da mesma forma. A seguir a implementação escalar das funcções `ov_any_lt_0f` e `ov_all_lt_0f`:

```
#define ov_any_lt_0f(x) ((x)<0.0f)
#define ov_all_lt_0f ov_any_lt_0f
```

2.1.6
Tratamento da cauda dos vetores

Até o momento, em todos os exemplos de código que apresentamos, não levamos em conta se o número de iterações dos *loops* era múltiplo de `OV_FLOAT_WIDTH`. Nesta Seção tratamos o caso onde o número de iterações não é múltiplo de `OV_FLOAT_WIDTH`.

Uma estratégia para garantir que os *loops* tenham o número de iterações múltiplo de `OV_FLOAT_WIDTH` é fazer com que a dimensão dos *arrays* que é contígua em memória seja múltipla de `OV_FLOAT_WIDTH`. Essa estratégia também garante um alinhamento de memória (e também é utilizada para otimizar o acesso à memória na programação geral em GPUs, com indexação a partir de um endereço base alinhado e *offsets* múltiplos de CUDA *warp size*[37] ou `CL_KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE` no caso de OpenCL[38]).

Se alocamos um *array* com a função `ov_malloc`, então podemos deixar o *loop* "invadir"os últimos elementos do vetor na última iteração, pois a função `ov_malloc` aloca `OV_FLOAT_WIDTH` elementos adicionais. Na Listagem 2.1, podemos notar que a última iteração processa elementos adicionais. Esta condição ocorre quando o número `n` de iterações não é múltiplo de `OV_FLOAT_WIDTH`.

Essa estratégia de invadir os últimos elementos pode ser vista na Listagem 2.1 e contempla a maioria dos casos de uso. Mas, em alguns casos como por exemplo, na atualização de apenas uma faixa dos elementos de um *array*, os limites dos *loops* devem ser respeitados para a corretude do código.

A Listagem 2.7 mostra uma cópia de memória SIMD entre dois *arrays* do tipo `float`, onde o limite n do *loop* é respeitado.

Criamos a constante `OV_FLOAT_TAIL` com o valor `OV_FLOAT_WIDTH-1`. Podemos ver na Listagem 2.7 que o primeiro *loop* subtrai `OV_FLOAT_TAIL` elementos do limite n do *loop*, garantindo que o limite n elementos não é excedido.

PUC-Rio - Certificação Digital Nº 1421599/CA

Listagem 2.7: Exemplo de uma cópia de floats entre *arrays*.

```
void memcpy_f(float *dst, float const *src,
              int const n)
{
  int i=0; /* inicializa o contador do loop */

  /* Primeiro loop: percorre OV_FLOAT_WIDTH elementos por
     iteracao, sem invadir na ultima iteracao */
  for (; i < n-OV_FLOAT_TAIL; i+=OV_FLOAT_WIDTH)
  {
    ov_float vx = ov_ldf(&src[i]); // load alinhado
    ov_stf(&dst[i], vx);           // store alinhado
  }

  /* Segundo loop: percorre o resto dos elementos
     utilizando um codigo escalar */
  for (; i < n; i++) dst[i]=src[i];
}
```

PUC-Rio - Certificação Digital Nº 1421599/CA

2.1.7 Reduções

Operações de redução são frequentes em aplicações de alto desempenho, onde é comum fazer operações entre os elementos de uma variável vetorial, como por exemplo calcular a soma de todos os elementos de um *array*. O OpenVec provê funções matemáticas de redução como a função `ov_all_sumf` que recebe uma variável SIMD do tipo `ov_float` e retorna um único `float` como resultado, conforme a demonstração abaixo para uma arquitetura com 8 elementos por vetor SIMD:

$$retorno \leftarrow x_0 + x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7$$

A Listagem 2.8 mostra um código que soma todos os elementos de um *array*, tratando o caso em que o número de elementos do *array* não é múltiplo de `OV_FLOAT_WIDTH`. Este código também faz o tratamento de cauda.

Listagem 2.8: Soma de todos os elementos de um *array*

```
float soma_array(int const n, float *x)
{
  int i=0; // primeiro indice do loop

  // Inicializa com zero a soma parcial SIMD
  ov_float tmp=ov_zerof;
```

```
  // Soma um vetor SIMD por iteracao
  for (; i<n-OV_FLOAT_TAIL; i+=OV_FLOAT_WIDTH)
    tmp += ov_ldf(&x[i]);

  float vsum = ov_all_sumf(tmp); // Reducao

  // O segundo loop soma o resto dos elementos
  for (; i<n; i++) vsum += x[i];

  return vsum;
}
```

Os resultados da soma da Listagem 2.8 podem variar entre as diferentes arquiteturas, porque a ordem em que os `floats` são somados influencia o resultado. Por exemplo, somar um valor muito pequeno a um valor muito grande não modifica o valor muito grande nos casos onde a diferença entre eles é maior que a precisão da mantissa.

Nossa implementação corrente não implementa as funções de redução matemática de forma otimizada. Elas são implementadas com um código escalar para todas as arquiteturas. Mas tendo em vista que essas funções são normalmente utilizadas para juntar os resultados após um *loop*, o impacto no desempenho é irrelevante.

A Tabela 2.6 mostra as funções de redução matemática.

Função	Operação
`ov_all_sumf(x)`	$x_0 + x_1 + x_2 + ... + x_n$
`ov_all_prodf(x)`	$x_0 \times x_1 \times x_2 \times ... \times x_n$
`ov_all_maxf(x)`	$max(x_0, x_1, x_2, ..., x_n)$
`ov_all_minf(x)`	$min(x_0, x_1, x_2, ..., x_n)$

Tabela 2.6: Funções de redução matemática, com $n =$ `OV_FLOAT_WIDTH-1`. O OpenVec implementa as mesmas funções para precisão dupla com o sufixo d.

2.2 Discussão

Nossa implementação mantem o "*look and feel*" da programação proprietária com intrínsecos, mas de maneira portável. Para manter o desempenho não adicionamos nenhum grau de abstração, tornando talvez, o seu uso mais difícil.

Como o OpenVec trata apenas a vetorização, o desenvolvedor deve utilizar uma outra ferramenta para utilizar todos os núcleos da CPU. A Seção

PUC-Rio - Certificação Digital Nº 1421599/CA

4.6 apresenta uma avaliação de desempenho do OpenVec em conjunto com o OpenMP.

Em C++, nosso *overload* de operadores facilitou muito a vetorização explícita em relação aos intrínsecos proprietários.

Poderemos no futuro, adicionar ao OpenVec todos os intrínsecos de todas as arquiteturas suportadas. E prover ao desenvolvedor uma forma de verificar se o intrínseco OpenVec é nativo ou emulado. Esta verificação é feita em tempo de de compilação (ex: `#ifdef OV_NATIVE_SHUFFLE`). Então, o desenvolvedor pode escrever algumas versões de código e combinar esta funcionalidade com uma estratégia de *off-line autotuning* (em tempo de compilação)[39].

Esta abordagem traz a desvantagem da manutenção de múltiplas versões de código. E traz a vantagem da possibilidade de atingir todo o desempenho potencial de uma arquitetura, tendo em vista que todos os seus intrínsecos nativos estão disponíveis.

Recomendamos que quando possível, o desenvolvedor utilize a *keyword* `restrict` para informar ao compilador que os ponteiros não estão em alias. O uso do `restrict` pode ajudar o compilador a fazer algumas otimizações[40]. Como o `restrict` não faz parte do padrão C original, essa *keyword* pode variar entre compiladores. Fornecemos a *keyword* `ov_restrict` que mapeia para o `restrict` específico de todos os compiladores que utilizamos neste trabalho.

Disponibilizamos o OpenVec sob a licença MIT[41]. O projeto pode ser baixado no endereço https://github.com/OpenVec/OpenVec.

PUC-Rio - Certificação Digital Nº 1421599/CA

3
Biblioteca de Arquiteturas Heterogêneas

Após portar aplicações de imageamento sísmico para arquiteturas heterogêneas tais como IBM Cell[21] e Nvidia CUDA[42][43][44], notamos que um grande esforço de desenvolvimento é gasto no gerenciamento dos dispositivos secundários, ou seja, em um código de preparação do dispositivo para poder executar as operações de interesse. Isso nos motivou a implementar uma biblioteca que provê as tarefas usuais na programação em um ambiente heterogêneo, composto por um processador principal e um ou mais dispositivos auxiliares de processamento. Por questões históricas, implementamos esta biblioteca em Fortran 90. Neste capítulo descrevemos esta biblioteca.

O modelo de computação heterogênea, muitas vezes possui dois tipos de memória: a memória principal, que é conectada ao processador (CPU) e que neste contexto definimos como memória da CPU ou *host memory* e a memória de cada dispositivo. A Figura 3.1 mostra um exemplo de arquitetura heterogênea do tipo CPU+GPU.

PUC-Rio - Certificação Digital Nº 1421599/CA

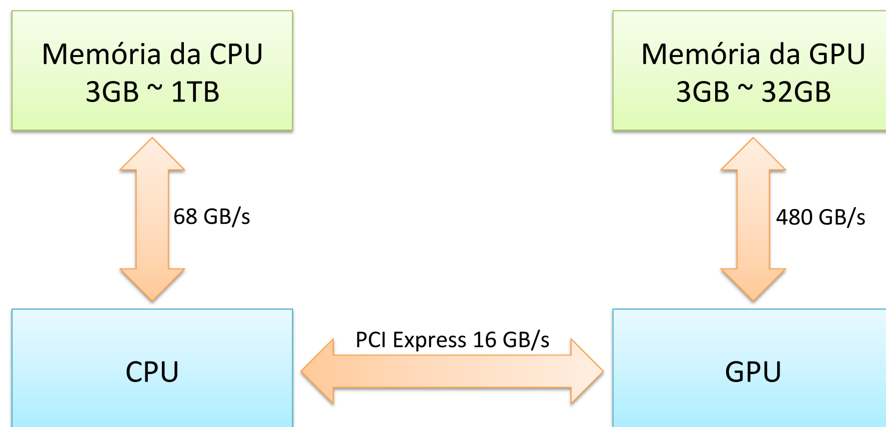

Figura 3.1: Esquema de conexão em uma arquitetura heterogênea CPU+CPU. Desempenho de conexão baseado na seguinte configuração: CPU Xeon E5-4669 v3, GPU Nvidia K80 conectada via PCIe 16X de 3ª geração.

3.1
Programação de Arquiteturas Heterogêneas

Para tirar proveito de uma arquitetura heterogênea, o programador deve dividir sua aplicação em dois tipos de código, um que é executado no processador principal (CPU) e outro que é executado em um dispositivo secundário, como por exemplo GPU. Esses dispositivos também são chamados de aceleradores e não funcionam de forma independente, precisando estar conectados a um processador principal. Neste trabalho usaremos o termo CPU para definir o processador principal. Apesar dos dispositivos secundários possuírem uma

ou mais unidades de processamento, usaremos o termo dispositivo para definir o dispositivo auxiliar, que é diferente do processador principal.

O interesse em utilizar esses dispositivos vem de fatores variados[45], como maior desempenho, economia de energia e espaço ou simplesmente melhor custo-benefício. A Figura 3.2 (fonte: CUDA C Programming Guide[46]) compara a evolução do poder computacional das placas gráficas Nvidia (dispositivo secundário) com as CPUs Intel.

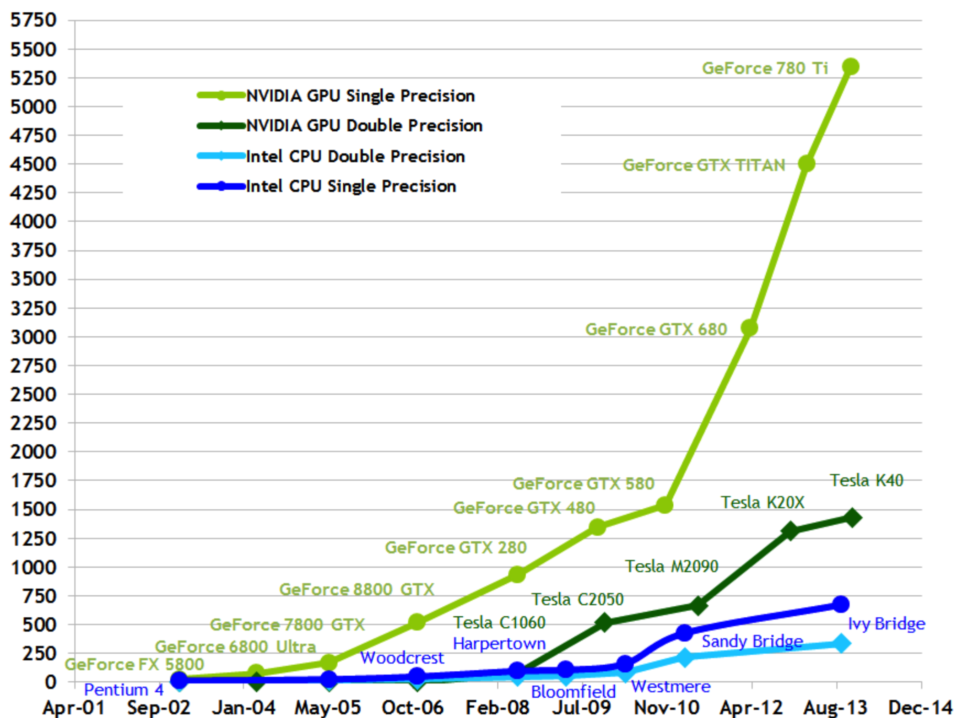

Figura 3.2: Comparativo da evolução do poder computacional em GFlops/s entre Nvidia e Intel. Computação heterogênea em verde, computação homogênea em azul.

O código que é executado na CPU compreende a inicialização da aplicação, E/S e a interface com usuário. Na computação heterogênea, o código da CPU também faz o gerenciamento dos dispositivos, como por exemplo alocação de memória no dispositivo. Os dispositivos executam pequenas porções de código, tipicamente chamados de *kernels*.

Para executar um *kernel* no dispositivo, o desenvolvedor precisa inicializar o dispositivo, alocar uma área de memória no dispositivo e provavelmente fazer a inicialização desta área, copiando o conteúdo da memória principal (memória da CPU) para essa memória. A memória alocada no dispositivo é passada como argumento no momento da invocação do *kernel*. O código que executa na CPU é o responsável por fazer a invocação de um *kernel* no dispositivo, a inicialização do dispositivo, a alocação / inicialização da memória do dispositivo e por controlar as filas de submissão. Diversos trabalhos[47, 48, 49]

PUC-Rio - Certificação Digital Nº 1421599/CA

consideram esse código de gerenciamento que executa na CPU como sendo um *"boilerplate code"*.

Como existe uma memória para a CPU e outra memória para o dispositivo, o desenvolvedor deve escolher com cuidado onde colocar cada estrutura de dados. Sua escolha deve se basear na velocidade e na capacidade de cada memória e na velocidade da conexão entre elas. A conexão entre a CPU e o dispositivo é feita pelo barramento PCI Express[1] que atualmente é o gargalo deste tipo de arquitetura, pois sua banda passante é muito menor que as bandas das memórias da CPU e do dispositivo.

O tempo gasto nas transferências CPU⇒dispositivo e dispositivo⇒CPU inviabiliza a implementação eficiente de muitos algoritmos. Um determinado *kernel* pode executar em menos tempo no dispositivo se comparado com sua versão que executa na CPU, mas o tempo gasto para transferir os argumentos de entrada e saída pode cancelar o ganho de desempenho que este *kernel* traria. Entretanto, é possível adicionar uma concorrência interna ao dispositivo e entre dispositivos, sobrepondo computação e transferências, minimizando o impacto do tempo de transferência.

Durante o processo de desenvolvimento, o desenvolvedor gasta muito tempo com o código de gerenciamento, que faz a preparação para a execução no *kernel*, como por exemplo, cópias entre as memórias da CPU e do dispositivo. Usualmente não é possível isolar esse código de gerenciamento do dispositivo em uma parte específica do código. Além disso é preciso tratar o código da arquitetura convencional, que só executa na CPU. Com isso o código fica com tratamentos de diferentes arquiteturas espalhados por diferentes trechos, ficando difícil de ler e manter.

O *kernel* normalmente é o *hot spot* da aplicação i.e. o trecho de código com o qual se pretende reduzir o tempo de execução. Mas em alguns casos, o desenvolvedor gasta mais esforço[50] e linhas de código gerenciando dispositivos e mantendo duas ou mais versões do código do que atacando o problema de aumentar o desempenho do *hot spot* da aplicação.

Diferentemente dos *kernels*, que podem ser isolados em uma invocação de função, o código de gerenciamento aparece em diferentes partes/camadas da aplicação, o que torna difícil desacoplar esse código da aplicação. O processo de adaptação e *tuning* de um código para uma arquitetura heterogênea como CUDA ou OpenCL pode ser demorado, envolver uma significativa reorganização do código e ficar mais suscetível a erros[51].

[1]PCI Express (Peripheral Component Interconnect Express) é o barramento padrão para conexão de dispositivos, tais como, placas de rede, controladoras de disco, placas gráficas (GPUs) e aceleradores.

PUC-Rio - Certificação Digital Nº 1421599/CA

Abordagens como OpenMP 4.0[52], OpenACC[53], OCCA[54] e HMPP[55], que utilizam um único código portável com anotações, ainda não atingem o desempenho obtido ao codificar diretamente com a API mais adequada a cada dispositivo[56, 57, 58, 59]. Ou seja, para se obter o maior desempenho em uma determinada arquitetura é preciso utilizar a API que melhor se adapta àquela arquitetura. Muitas vezes essa API é proprietária, como é o caso das APIs Nvidia CUDA e IBM AltiVec *intrinsics*.

Para permitir que o desenvolvedor se concentre no *hot spot* da aplicação, implementamos a biblioteca **HLIB**[60, 61] que gerencia as primitivas básicas da computação heterogênea mas que também funciona, de forma transparante, com arquiteturas homogêneas tradicionais (somente CPU). A biblioteca implementa os aspectos da computação heterogênea, como configurações e inicializações, mas deixa a cargo do desenvolvedor implementar o *kernel* de forma nativa, por exemplo, em CUDA, ou utilizando OpenMP em conjunto com SIMD *intrinsics* da Intel.

A biblioteca HLIB tem como público alvo desenvolvedores que desejam utilizar computação heterogênea utilizando APIs que não comprometem o desempenho. O ganho com o uso da biblioteca é aumentar o reuso de código, já que seu uso não traz nenhum ganho de desempenho, pois cabe ao desenvolvedor codificar os *kernels* de forma nativa para cada arquitetura.

A biblioteca fornece a liberdade de codificar os *kernels* na API que apresenta o maior potencial de desempenho para determinada arquitetura. O foco do desenvolvimento passa a ser o *hot spot* da aplicação, pois com a HLIB todo o resto do código de gerenciamento que executa na CPU é portável e agnóstico de arquitetura.

A HLIB implementa todo o código de gerenciamento do dispositivo, que compreende as primitivas da Tabela 3.1.

Primitivas
Inicializar / finalizar dispositivo
Alocar / desalocar memória no dispositivo
Alocar / desalocar memória para transferências CPU⇔dispositivo
Zerar memória alocada no dispositivo
Cópia de memória CPU⇔dispositivo e dispositivo⇔dispositivo
Concorrência entre computação e cópias de memória
Passagem de mensagem entre dispositivos

Tabela 3.1: Primitivas HLIB da computação heterogênea.

A principal diferença entre a nossa abordagem e trabalhos similares[62, 63, 64] é que, na nossa abordagem, cabe ao desenvolvedor enfileirar os *kernels* e invocar explicitamente as transferências entre a memória da CPU e a memória do dispositivo. Como apenas mapeamos funcionalidades já existentes, o uso da

PUC-Rio - Certificação Digital Nº 1421599/CA

PUC-Rio - Certificação Digital Nº 1421599/CA

biblioteca não restringe a capacidade de atingir o desempenho possível em cada arquitetura e mantém uma porção crítica do código portável e reutilizável. Podemos classificar o grau de portabilidade na computação heterogênea em três níveis:

- Máximo: um código para todas as arquiteturas
- Médio: um código de gerenciamento e múltiplos *kernels* para várias arquiteturas
- Mínimo: múltiplos *kernels* e códigos de gerenciamento para várias arquiteturas

A Figura 3.3 faz um comparativo entre o desempenho e o grau de portabilidade. Com a HLIB existe um único código de gerenciamento mas é necessário portar o *kernel* para cada arquitetura. Com relação ao desempenho uma aplicação que utiliza a HLIB se localiza próxima às APIs nativas, pois o *hot spot* é codificado da mesma forma na HLIB e na API nativa. Classificamos com o grau médio de portabilidade uma aplicação que utiliza a HLIB.

Figura 3.3: Comparativo entre o grau de portabilidade e desempenho potencial. OpenCL otimizado se refere a múltiplos *kernels* OpenCL otimizados para cada arquitetura. O desempenho é potencial, fica a cargo do desenvolvedor.

A Figura 3.4 mostra a organização de uma aplicação utilizando a HLIB. A biblioteca implementa todas a primitivas da Tabela 3.1.

Na Figura 3.4, a porção em vermelho representa a biblioteca heterogênea HLIB com os *backends* para as diversas plataformas suportadas: CUDA, OpenCL, hStreams e código regular homogêneo. As bibliotecas de *runtime* de cada arquitetura são mostradas em verde. A aplicação, em azul, contém duas camadas bem distintas: a camada de cima, que contém um código agnóstico de arquitetura e invoca a biblioteca heterogênea, e a camada de baixo que possui *kernels* com otimizações específicas para cada arquitetura.

Esses *kernels* são codificados de forma que sua invocação tenha a mesma interface Fortran 90: a aplicação invoca o *kernel* OpenCL da mesma forma

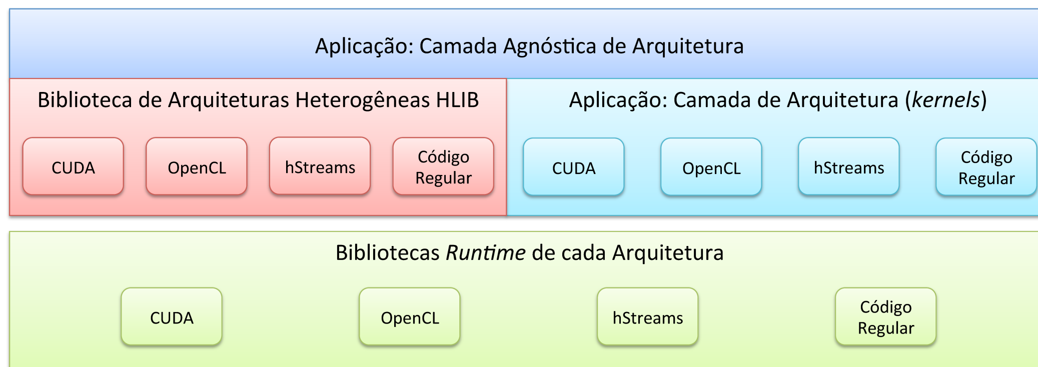

Figura 3.4: Aplicação em azul, biblioteca heterogênea em vermelho e as bibliotecas *runtime* de cada arquitetura em verde.

que invoca o *kernel* CUDA. Não existe nenhuma chamada CUDA ou qualquer chamada exclusiva de uma arquitetura espalhada pelo código da aplicação. Todas as chamadas proprietárias estão contidas nos *kernels*.

Em derterminadas APIs, como CUDA e OpenCL, a invocação dos *kernels* é feita utilizando sintaxes distintas e não pode ser feita diretamente por um código Fortran 90. Com isso, é necessário que o desenvolvedor crie um código *driver* com a mesma interface para todas as arquiteturas. O *driver* faz a invocação do kernel com a sintaxe apropriada de cada API. A Listagem 3.1 mostra o código de um *driver* escrito em CUDA para a função SAXPY. Este driver fica em um arquivo CUDA e é compilado com o compilador da Nvidia (`nvcc`).

Listagem 3.1: Exemplo de um código *driver*

```
// Assinatura de funcao compativel com Fortran
extern"C"
void saxpy_driver_(void *contexto, cudaStream *fl1,
                   int *n, float *a, float **x, float **y)
{
  // O objeto contexto nao e' necessario em CUDA,
  // mas e' necessario em outros backends como OpenCL
  // Objeto Fortran HLIB_filat_t == cudaStream* em CUDA

  // Sintaxe de invocacao CUDA com 256 threads por bloco
  // Kernel enfileirado na fila fl1
  saxpy_cuda<<<((*n)+255)/256, 256, 0, *fl1>>>(*n, *a, *x, *y);
}
```

A camada agnóstica de arquitetura, que executa na CPU, invoca o *driver*. E o *driver* invoca o *kernel* no dispositivo. A seguir temos um exemplo da invocação Fortran do *driver* da Listagem 3.1:

```
CALL Saxpy_driver(contexto, fila, n, a, x, y)
```

PUC-Rio - Certificação Digital Nº 1421599/CA

Esta invocação contempla todas as arquiteturas e faz parte da camada agnóstica. Cabe ao desenvolvedor escrever o *driver* e o *kernel* para cada arquitetura de interesse.

3.2 Implementação

Implementamos a biblioteca heterogênea de forma que um único código de gerenciamento possa ser utilizado da mesma forma, em arquiteturas homogêneas e heterogêneas. Implementamos *backends* para as seguintes arquiteturas:

- Nvidia CUDA
- OpenCL
- Intel hStreams
- CPU regular (arquitetura homogênea com C e Fortran)

A biblioteca é exposta com um módulo Fortran 90, e implementa uma interseção de funcionalidades de três APIs: Nvidia CUDA, OpenCL e Intel hStreams, conforme a Figura 3.5.

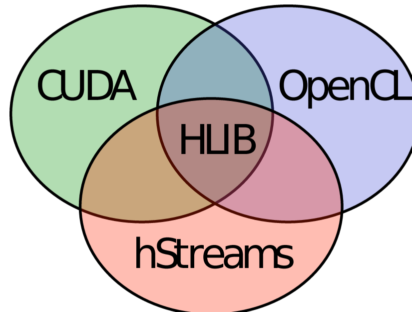

Figura 3.5: A HLIB implementa uma interseção de funcionalidades de diversas APIs.

Muitas decisões de implementação foram feitas para manter a compatibilidade com as APIs suportadas. Por exemplo, para manter a compatibilidade com o OpenCL, a memória alocada no dispositivo é um objeto opaco e não um ponteiro. Como outro exemplo, as filas de execução são sempre FIFO para garantir a compatibilidade com o `cudaStream` da Nvidia.

As funcionalidades da HLIB são muito próximas das funcionalidades das APIs dos *backends*. Em muitos casos, nossos *backends* apenas invocam uma única função correspondente da API da arquitetura. Por exemplo, invocamos a função `cudaMemcpyAsync` para implementar a primitiva de cópia de memória em CUDA.

PUC-Rio - Certificação Digital Nº 1421599/CA

A interseção de funcionalidades da Figura 3.5 foi suficiente para implementarmos as primitivas da Tabela 3.1. Não houve perda de desempenho nestas primitivas, pois existe um mapeamento direto para as funcionalidades das APIs dos *backends*.

Abordamos na Seção 3.4 o uso de primitivas específicas de uma arquitetura, isto é, primitivas que não estão contidas na interseção de APIs da Figura 3.5.

3.2.1 Inicializar / finalizar dispositivo

O primeiro passo ao se utilizar a HLIB é inicializar o dispositivo criando um contexto. Ao inicializar um dispositivo um objeto é retornado com o contexto criado com o dispositivo.

O contexto é um descritor do dispositivo, e após a inicialização fica associado a um único dispositivo. O contexto é utilizado durante a execução das primitivas da Tabela 3.1 para informar ao *runtime* da arquitetura qual dispositivo deverá executar a primitiva. Como exemplo, em OpenCL o contexto contém um objeto nativo do tipo `cl_context`. No momento da invocação o desenvolvedor informa o número do dispositivo que deseja utilizar, pois uma máquina pode ter mais de um dispositivo secundário. O código a seguir mostra a interface de criação de contexto com dispositivo:

```
SUBROUTINE HLIB_CriaContexto(dispNum, contexto, ierro)
                         ! Numero do dispositivo
  INTEGER,               INTENT(IN)  :: dispNum
                         ! Contexto retornado
  TYPE(HLIB_contexto_t), INTENT(OUT) :: contexto
                         ! Codigo de erro
  INTEGER,               INTENT(OUT) :: ierro
```

Implementamos o tipo `HLIB_contexto_t` como um objeto opaco de Fortran, ou seja, um objeto onde todos os atributos são privados.

Todas as demais sub-rotinas da biblioteca recebem como argumento o contexto, atuando no dispositivo associado a ele. Quando o desenvolvedor criar um código *driver* (que invoca um *kernel*), este também deve receber o contexto como argumento.

O processo de inicialização compreende verificar quantos dispositivos (`ndisp`) existem na máquina e associar ao contexto o dispositivo `dispNum MOD ndisp`, onde `dispNum` é o número do dispositivo passado para a sub-rotina `HLIB_CriaContexto`. Em alguns backends como o OpenCL, também é necessário criar uma fila de execução padrão para as operações internas

PUC-Rio - Certificação Digital Nº 1421599/CA

da biblioteca. Em CUDA, existem chamadas síncronas que não necessitam de uma fila (`cudaStream`). O comportamento dessa fila de execução padrão será descrito na Seção 3.2.6.

Internamente, o processo de inicialização é diferente para cada *backend*, a tabela 3.2 descreve as operações executadas em cada um:

Operação \ *Backend*	CUDA	hStreams	OpenCL
Obter plataforma		`hStreams_Init`	`clGetPlatformIDs`
Obter `ndisp`	`cudaGetDeviceCount`	`hStreams_GetNumPhysDomains`	`clGetDeviceIDs`
Associar dispositivo	`cudaSetDevice`	`hStreams_AddLogDomain`	`clCreateContext`
Criar fila padrão		`hStreams_StreamCreate`	`clCreateCommandQueue`

Tabela 3.2: Inicialização interna da HLIB em diferentes arquiteturas.

Ao final da aplicação, o desenvolvedor deve destruir o contexto com o dispositivo utilizando a sub-rotina a seguir:

```
SUBROUTINE HLIB_DestroiContexto(contexto, ierro)
  TYPE(HLIB_contexto_t), INTENT(INOUT) :: contexto
  INTEGER,               INTENT(OUT)   :: ierro
```

3.2.2
Alocar / desalocar memória no dispositivo

Os arrays necessários à execução de um *kernel* devem ser explicitamente alocados no dispositivo. As memórias da CPU e do dispositivo são distintas e normalmente possuem um desempenho diferente, conforme a Figura 3.1.

Na computação heterogênea existem dois modelos distintos de alocação de memória, alocação explícita e implícita. Por exemplo, o OpenACC e o OpenMP fazem a alocação de memória no dispositivo de forma implícita e em CUDA e OpenCL cabe ao desenvolvedor alocar explicitamente a memória no dispositivo. Adotamos a estratégia de alocação explícita: na HLIB o desenvolvedor explicitamente aloca a memória no dispositivo.

Para alocar memória em um dispositivo associado a um contexto, o programador deve invocar a sub-rotina a seguir:

```
SUBROUTINE HLIB_AlocaMemDisp(contexto, tam, zerar, memDisp,
                             ierro)
                           ! Dispositivo
  TYPE(HLIB_contexto_t), INTENT(IN)  :: contexto
                           ! Numero de elementos
  INTEGER(HLIB_KINDMEM), INTENT(IN)  :: tam
                           ! .TRUE. para inicializar com zero
  LOGICAL,               INTENT(IN)  :: zerar
```

PUC-Rio - Certificação Digital Nº 1421599/CA

```
                        ! Area alocada de memoria
  TYPE(HLIB_real_t),    INTENT(OUT) :: memDisp
                        ! Codigo de erro
  INTEGER,              INTENT(OUT) :: ierro
```

Essa sub-rotina aloca a área de memória no dispositivo e devolve um objeto opaco, como por exemplo o `HLIB_real_t` que mapeia para o tipo básico `float` no dispositivo. Como no OpenCL, a memória alocada no dispositivo é um objeto opaco, implementamos da mesma forma para manter a compatibilidade com o OpenCL.

Em CUDA e hStreams, a memória no dispositivo é um ponteiro como outro qualquer, aumentando a flexibilidade de programação, pois a aritmética de ponteiro e a indexação ocorrem da mesma forma que em C/C++. Mas para manter a interseção de APIs da Figura 3.5, definimos nossa API da mesma forma que a API menos flexível, neste caso OpenCL. Os *backends* CUDA, hStreams e regular armazenam o ponteiro para a memória do dispositivo dentro do objeto opaco.

A biblioteca armazena no objeto a quantidade de memória alocada, assim é possível verificar invasão de memória em operações de cópia. Definimos alguns tipos básicos para alocação de memória no dispositivo:

- `HLIB_real_t`
- `HLIB_double_t`
- `HLIB_int32_t`
- `HLIB_int64_t`

Com exceção do *backend* regular, em todos os outros a implementação é feita invocando a função correspondente para alocação de memória no dispositivo como `cudaMalloc` em CUDA. O *backend* regular invoca um simples `malloc` em C, pois só existe a memória principal.

A memória do dispositivo é desalocada com a sub-rotina a seguir:

```
SUBROUTINE HLIB_DesalocaMemDisp(contexto, memDisp, ierro)
                        ! Dispositivo associado
  TYPE(HLIB_contexto_t), INTENT(IN)    :: contexto
                        ! Memoria a ser desalocada
  TYPE(HLIB_real_t),    INTENT(INOUT) :: memDisp
                        ! Codigo de erro
  INTEGER,              INTENT(OUT)   :: ierro
```

A biblioteca também verifica se o contexto foi criado e se a memória foi alocada antes de ser desalocada.

PUC-Rio - Certificação Digital Nº 1421599/CA

3.2.3 Memória para transferências entre CPU e dispositivo

Na HLIB, as transferências entre CPU e dispositivo são explícitas, assim como em CUDA, OpenCL e hStreams. Por exemplo, o desenvolvedor pode inicializar um *array* na memória da CPU com dados lidos do disco e copiar seu conteúdo para um *array* equivalente na memória do dispositivo, e pode copiar um *array* alocado na memória do dispositivo, que é a saída de um *kernel*, para um *array* equivalente na memória da CPU. Esse padrão é bem comum na computação heterogênea.

Em arquiteturas onde o dispositivo é conectado via barramento PCI Express, conforme a Figura 3.1, é possível aumentar o desempenho entre cópias de memória CPU ⇔ dispositivo alocando uma memória *pinned*[65, 66, 67]. Esse tipo de memória é alocada na memória da CPU e não pode ser paginada (*swap*) pelo sistema operacional. Com isso é possível fazer operações do tipo DMA (*direct memory access*), habilitando a sobreposição das cópias de memória com execuções de *kernels* (em CUDA esta sobreposição é feita com as chamadas assíncronas, que só funcionam se a memória alocada na CPU for *pinned*[68]). O uso desse tipo de memória para transferências CPU ⇔ dispositivo é recomendado pois essas transferências ocorrem no barramento PCI Express, que é o gargalo[69] de muitas arquiteturas heterogêneas.

Para alocar a memória *pinned* localizada na memória da CPU, o desenvolvedor invoca a sub-rotina a seguir:

```fortran
SUBROUTINE HLIB_AlocaMemTransf2D(contexto, lb, ub,
                                  array, ierro)
                            ! Dispositivo
  TYPE(HLIB_contexto_t),    INTENT(IN) :: contexto
                            ! Lower bounds
  INTEGER,                  INTENT(IN) :: lb(2)
                            ! Upper bounds
  INTEGER,                  INTENT(IN) :: ub(2)
                            ! Ponteiro CPU
  REAL,                     POINTER :: array2D(:,:)
  INTEGER,                  INTENT(OUT) :: ierro
```

A função anterior aloca um array 2D de números reais em uma memória *pinned*, localizada na memória da CPU. A biblioteca também oferece as interfaces para arrays de números inteiros e de três dimensões. Esse tipo de alocação com *lower* e *upper bounds* é muito útil, aumentando a flexibilidade na indexação de arrays, que não precisam iniciar no índice 1.

PUC-Rio - Certificação Digital Nº 1421599/CA

Essa memória *pinned* é acessada como qualquer outro array Fortran 90, dando flexibilidade ao desenvolvedor. No *backend* regular essa alocação é feita com um simples `ALLOCATE`.

3.2.4 Zerar memória alocada no dispositivo

Para inicializar com zero (`todos bytes` ← `0x00`) uma memória alocada no dispositivo, basta invocar a sub-rotina a seguir:

```
SUBROUTINE HLIB_ZeraMemDisp(contexto, memDisp, ierro)
                          ! Dispositivo
  TYPE(HLIB_contexto_t), INTENT(IN)    :: contexto
                          ! Memoria no dispositivo
  TYPE(HLIB_double_t),   INTENT(INOUT) :: memDisp
                          ! Codigo de erro
  INTEGER,               INTENT(OUT)   :: ierro
```

3.2.5 Cópia de memória CPU⇔dispositivo e dispositivo⇔dispositivo

Na HLIB as transferências de memória são explícitas. É possível fazer cópias entre as memórias da CPU e do dispositivo, e entre *arrays* alocados na memória do dispositivo.

Como a memória alocada no dispositivo é um objeto opaco, para se definir a posição inicial da cópia de memória é necessário passar um *offset* explícito para a sub-rotina de cópia. Em OpenCL o tipo opaco `cl_mem` é utilizado para representar a memória do dispostivio. Não é possível indexar diretamente este objeto como por exemplo `ptr[idx]` ou `ptr+=idx`, o desenvolvedor deve passar como argumento o *offset* `idx`.

Também é preciso passar explicitamente a direção da cópia que pode ser CPU⇒dispositivo ou dispositivo⇒CPU. A sub-rotina de cópia entre as memórias da CPU e do dispositivo é mostrada a seguir:

```
SUBROUTINE HLIB_CopiaDispCPU(contexto, memDisp, offset,
                              arrayCPU, dir, ierro)
                          ! Dispositivo
  TYPE(HLIB_contexto_t), INTENT(IN)    :: contexto
                          ! Memoria dispositivo
  TYPE(HLIB_real_t),     INTENT(INOUT) :: memDisp
                          ! Offset em elementos
  INTEGER(HLIB_KINDMEM), INTENT(IN)    :: offset
                          ! Memoria CPU
  REAL,                  INTENT(INOUT) :: arrayCPU(:,:)
```

PUC-Rio - Certificação Digital Nº 1421599/CA

```
                               ! Direcao da copia
  INTEGER,                     INTENT(IN)     :: dir
                               ! Codigo de erro
  INTEGER,                     INTENT(OUT)    :: ierro
```

A interface acima define a cópia para arrays 2D de números reais. A biblioteca provê diversas interfaces Fortran 90 com verificação de tipo. Como existe um tipo básico associado à memória do dispositivo, a interface só permite a invocação de cópias de memória entre arrays do mesmo tipo, como um array na CPU tipo `DOUBLE PRECISION` e um array tipo `HLIB_double_t` no dispositivo.

O parâmetro `offset` indica a posição inicial da cópia no *array* da memória do dispositivo. Os *arrays* localizados na memória do dispositivo são organizados como um memória 1D linear, e o uso do `offset` tem a mesma semântica de indexação de um ponteiro C/C++, conforme o exemplo a seguir:

```
array_no_dispositivo[offset]
```

O próximo exemplo dispara uma cópia da memória do dispositivo para a memória da CPU, iniciando pelo centésimo elemento na memória do dispositivo (`offset`=99). A direção da cópia é determinada pelas constantes `HLIB_DESTINO_CPU` e `HLIB_ORIGEM_CPU`.

```
offset=99
CALL HLIB_CopiaDispCPU(contexto, memDisp, offset,
                       arrayCPU(10:20,30:50),
                       HLIB_DESTINO_CPU, ierro)
```

A quantidade de elementos da cópia é determinada pelos *lower* e *upper bounds* do `arrayCPU` localizado na memória da CPU. Antes da cópia, a HLIB verifica se haverá uma invasão de memória, pois tanto o tamanho do array do dispositivo, quanto o tamanho do array da CPU são conhecidos.

A sub-rotina para cópia entre memórias alocadas no dispositivo recebe dois arrays, um de origem e outro de destino, que devem ser do mesmo tipo básico. Para cada array um offset em elementos é informado. A seguir uma das interfaces para cópias de memória do dispositivo:

```
SUBROUTINE HLIB_CopiaDispDisp(contexto, origem, offOrig,
                              destino, offDest, tam, ierro)

                               ! Dispositivo
  TYPE(HLIB_contexto_t),       INTENT(IN)     :: contexto
```

PUC-Rio - Certificação Digital Nº 1421599/CA

```
                           ! Mem disp origem
  TYPE(HLIB_double_t),    INTENT(IN)    :: origem
                           ! Offset em elementos
  INTEGER(HLIB_KINDMEM), INTENT(IN)    :: offOrig
                           ! Mem disp destino
  TYPE(HLIB_double_t) ,   INTENT(INOUT) :: destino
                           ! Offset em elementos
  INTEGER(HLIB_KINDMEM), INTENT(IN)    :: offDest
                           ! Tamanho da copia em elementos
  INTEGER(HLIB_KINDMEM), INTENT(IN)    :: tam
                           ! Codigo de erro
  INTEGER,                INTENT(OUT)   :: ierro
```

O argumento `tam` define o número de elementos a serem copiados. A HLIB também verifica se haverá uma invasão de memória.

3.2.6 Filas de execução concorrentes

A biblioteca também provê a capacidade de gerenciar filas de comandos a serem executados no dispositivo. Um comando pode ser uma transferência de memória ou uma execução de *kernel*. Cada fila é uma estrutura FIFO (*first in first out*). Dentro da mesma fila, um comando só pode começar a executar quando o anterior terminar por completo.

A funcionalidade de fila é implementada de forma diferente pelo OpenCL, CUDA e hStreams. Em uma fila `cl_command_queue` (OpenCL), dependendo da configuração, os comandos podem executar fora de ordem, enquanto que em uma fila `cudaStream` (CUDA), um comando só inicia após o encerramento do anterior, e em uma fila `HSTR_LOG_DOM` (hStreams), um comando pode iniciar se não houver dependência de dados com os comandos em execução naquela fila. O funcionamento das filas da HLIB é igual ao `cudaStream` da Nvidia, que é um caso particular do OpenCL e hStreams.

Os comandos em filas distintas podem executar em qualquer ordem e simultaneamente, i.e. filas distintas indicam operações independentes, que podem ser executadas de forma concorrente e fora de ordem. Por exemplo, uma cópia de memória C_1 inserida na $fila_1$ em um instante anterior ao da inserção de uma cópia C_2 na $fila_2$ poderá executar após C_2. Nada pode se afirmar sobre a ordem de execução de C_1 e C_2, pois são operações independentes.

A API da HLIB induz o desenvolvedor a implementar a concorrência de forma incremental, pois suas chamadas não recebem explicitamente a informação de qual fila utilizar. Acreditamos que o desenvolvedor deva primeiro desenvolver um código sequencial, e depois adicionar a concorrência de forma

PUC-Rio - Certificação Digital Nº 1421599/CA

incremental, informando qual fila deverá ser utilizada antes de invocar uma chamada HLIB. Não expomos a primitiva enfileirar. Para enfileirar comandos em uma determinada fila, basta selecioná-la como sendo a fila corrente. Para enfileirar um *kernel*, o desenvolvedor deve utilizar a API específica no código do *driver*. A Listagem 3.1, apresenta um exemplo de um *driver* para a arquitetura CUDA. O driver do exemplo enfileira o *kernel* em uma fila do tipo `cudaStream`.

Implementamos as seguintes primitivas de fila:

– Criar / destruir fila

– Selecionar fila

– Selecionar fila síncrona (padrão)

– Aguardar fila

– Testar se a fila está vazia

Após a seleção de uma fila, todos os comandos posteriores a seleção são enfileirados nessa fila. Os comandos são disparados em segundo plano, o controle volta para a CPU imediatamente. Para garantir que a execução de um comando terminou, é preciso aguardar a fila ou testar se ela está vazia. A seguir a rotina de criação de fila:

```
SUBROUTINE HLIB_CriaFila(contexto, fila, ierro)
                         ! Dispositivo
  TYPE(HLIB_contexto_t), INTENT(IN)  :: contexto
                         ! Fila a ser criada
  TYPE(HLIB_fila_t),     INTENT(OUT) :: fila
                         ! Codigo de erro
  INTEGER,               INTENT(OUT) :: ierro
```

Todo contexto possui uma fila síncrona padrão, que é a fila corrente desde o momento da criação do contexto. Se o desenvolvedor não criar e selecionar fila alguma, a fila síncrona padrão será utilizada em todos os comandos e não é preciso testar se um comando acabou. Ou seja, o comportamento padrão da HLIB é sempre síncrono (fila síncrona padrão): cada comando só começa a executar após o termino do anterior, e toda chamada bloqueia a CPU até o seu término. Uma fila só pode estar associada a um único objeto contexto (um único dispositivo), que deve ser informado no momento da criação da fila.

Porém toda fila criada explicitamente é assíncrona em relação à CPU: se uma tal fila está selecionada, nenhuma chamada à HLIB bloqueia a CPU. Com isso também é possível obter concorrência entre a CPU e o dispositivo. Essa opção de seleção de filas deixa o desenvolvedor livre para começar a

PUC-Rio - Certificação Digital Nº 1421599/CA

desenvolver com uma abordagem síncrona e adicionar a execução concorrente posteriormente, quando o código estiver mais maduro, pronto para receber otimizações. As sub-rotinas de cópia de memória não recebem a fila como argumento. O programador deve selecionar a fila com a sub-rotina de seleção de fila.

Os objetos `HLIB_fila_t` podem ser passados para os *drivers* criados pelo desenvolvedor. Estes objetos contêm a estrutura nativa da arquitetura em uso, como `cudaStream` em máquinas com GPUs Nvidia, conforme a Listagem 3.1.

As cópias do tipo CPU⇔dispositivo só poderão ser enfileiradas em uma fila assíncrona se a memória alocada na CPU for *pinned*. A rotina `HLIB_AlocaMemTransf` descrita anteriormente provê esse tipo de alocação.

As filas são muito utilizadas para sobrepor as cópias de memória com execução de *kernels*. Alguns dispositivos, como as GPUs atuais da Nvidia, suportam sobrepor duas cópias em sentidos diferentes e execução de *kernel*, permitindo esconder a latência da cópia de memória.

Em uma aplicação que implementa concorrência com filas, se o tempo de execução do *kernel* for maior que o tempo das cópias de memória, apenas o tempo do *kernel* irá contabilizar para o tempo total de execução. O tempo gasto com as cópias de memória não afeta o tempo de execução da aplicação. A seguir um exemplo da sobreposição de cópia de memória e execução de *kernel*:

Listagem 3.2: Pseudocódigo da sobreposição de cópia de memória e execução de *kernel*

```
Para i = 1, numero de trabalhos
  indiceFila = i%3   ! resto da divisao por 3

  ! define a fila a ser utilizada pelas sub-rotinas da HLIB
  CALL HLIB_SelecionaFila(contexto, filas(indiceFila), ierro)

  ! Aguarda o encerramento dos comandos na fila atual
  CALL HLIB_Sincroniza(contexto, ierro, filas(indiceFila))

  Dispara a copia da entrada para o dispositivo
  Dispara o kernel na fila atual
  Dispara a copia da saida do kernel para a CPU
Fim do para
! Aguarda o encerramento dos comandos em todas as filas
CALL HLIB_Sincroniza(contexto, ierro)
```

Na Listagem 3.2, enquanto o dispositivo executa o $trabalho_i$ o resultado do $trabalho_{i-1}$ é copiado de volta para a CPU e a entrada do $trabalho_{i+1}$ é copiada para o dispositivo. Como a fila tem um comportamento FIFO a cópia

PUC-Rio - Certificação Digital Nº 1421599/CA

do resultado do $trabalho_i$ só inicia após a execução do *kernel* do $trabalho_i$ que só inicia após a cópia da entrada do $trabalho_i$, pois todas essas operações são enfileiradas na mesma fila.

A sub-rotina `HLIB_Sincroniza` (primitiva aguardar fila) bloqueia a execução da CPU enquanto houver trabalho na fila. Se o desenvolvedor não informar a fila, a CPU fica bloqueada até todos os trabalhos do contexto ecenrrarem, conforme o exemplo da Listagem 3.2.

A Figura 3.6 mostra a execução no dispositivo utilizando as três filas. Note que se o tempo de cópia for menor ou igual ao tempo do *kernel* o dispositivo sempre estará executando um *kernel*. No caso da Figura 3.6b três comandos executam simultaneamente no dispositivo.

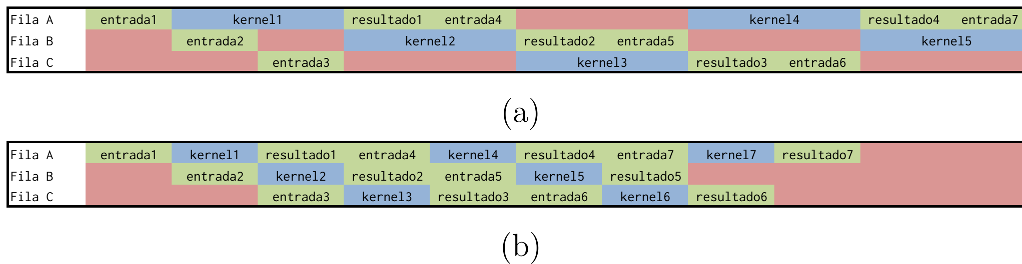

Figura 3.6: Uso do dispositivo ao longo do tempo. Execuções de *kernel* em azul, cópias de memória em verde e nada sendo executado em vermelho. (a) um exemplo de execução onde o tempo de cópia é menor que o tempo do *kernel*. (b) um exemplo de execução onde o tempo de cópia é igual ao tempo do *kernel*.

Essa sobreposição de cópia de memória com execução de *kernel* tem um potencial de ganho de desempenho de até 3 vezes, caso onde o tempo de cópia de ida e o tempo de cópia de volta são iguais ao tempo do *kernel*, pois essas três operações vão ocorrer em paralelo, conforme Figura 3.6b.

A Figura 3.7 utiliza o exemplo da Figura 3.6a mostrando o comportamento das três unidades de execução do dispositivo: execução de *kernel*, cópia para o dispositivo e cópia do dispositivo.

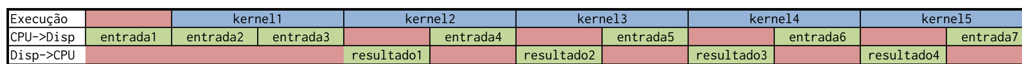

Figura 3.7: Uso das unidades de execução do dispositivo ao longo do tempo. Execuções de *kernel* em azul, cópias de memória em verde e nada sendo executado em vermelho.

Podemos notar na Figura 3.7 que a unidade de execução de *kernel* fica sempre cheia após a cópia da entrada do primeiro *kernel*. É possível afirmar que o tempo de execução amortizado, será o maior valor entre os tempos de cópia e de *kernel*, quando é possível executar as três operações em paralelo.

PUC-Rio - Certificação Digital Nº 1421599/CA

3.2.7
Passagem de Mensagem entre Dispositivos

A primitiva de troca de mensagens entre dispositivos não faz parte de nenhuma API da Figura 3.5. Em CUDA, é possível acessar a memória de outra GPU, mas ela deve estar instalada na mesma máquina. Implementamos chamadas equivalentes às do MPI para troca de mensagens entre dispositivos, utilizando o prefixo `HLIB_MPI`. Essas chamadas aceitam *buffers* alocados nos dispositivos. Nossa implementação, em sua camada mais baixa, invoca o MPI. Somente utilizamos funções do padrão MPI 1.1[70], com isso nossa implementação deverá funcionar com qualquer implementação da biblioteca MPI. Implementamos apenas algumas chamadas `HLIB_MPI`, conforme a demanda das aplicações do Capítulo 4.

Nos backends onde a memória do dispositivo é diferente da memória da CPU, existe um passo interno adicional para copiar a memória alocada no dispositivo para um buffer intermediário na memória da CPU, então o buffer da CPU é passado para a chamada MPI equivalente. Todo esse processo é transparente para o usuário.

A Figura 3.8 mostra a troca de mensagens entre dispositivos em um processo MPI trocando informações com dois vizinhos, um à esquerda e outro à direita. A biblioteca quebra a mensagem enviada em pedaços para sobrepor a comunicação MPI com as cópias de memória e execuções de *kernel*. Podemos notar que o tempo de comunicação não aumenta o tempo total de execução (em azul) e até cinco operações ocorrem simultaneamente na Figura 3.8:

- `Execuções de` *`kernel`*
- `Cópia de memória dispositivo⇒CPU`
- `Cópia de memória CPU⇒dispositivo`
- `Send MPI`
- `Receive MPI`

O primeiro `HLIB_MPI_sendRecv` da Figura 3.8 envia para a esquerda e recebe da direita. O primeiro pedaço da esquerda (e_1) é copiado do dispositivo para CPU, então e_1 é enviado para o vizinho da esquerda e o primeiro pedaço do vizinho da direita (vd_1) é recebido com `MPI_sendRecv(`e_1`, `vd_1`)`, então vd_1 é copiado para o dispositivo, repetindo esses passos até todos os pedaços acabarem. Então o segundo `HLIB_MPI_sendRecv`, que envia para a direita e recebe da esquerda é iniciado, com o primeiro pedaço da direita (d_1) sendo copiado do dispositivo para CPU e assim por diante.

PUC-Rio - Certificação Digital Nº 1421599/CA

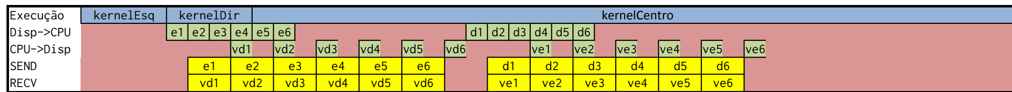

Figura 3.8: Comportamento de dois `HLIB_MPI_sendRecv` para trocar as bordas com dois processos MPI vizinhos. Eixo do tempo na horizontal. Execuções de *kernel* em azul, cópias de memória em verde, comunicação MPI em amarelo e nada sendo executado em vermelho. Mensagem quebrada em 6 pedaços. $e_1 = pedaço_1$ da esquerda do processo corrente, $ev_1 = pedaço_1$ do vizinho da esquerda, $d_1 = pedaço_1$ da direita do processo corrente, $vd_1 = pedaço_1$ do vizinho da direita.

Para melhor visualização, na Figura 3.8 definimos a velocidade das cópias CPU⇔dispositivo como duas vezes a velocidade da rede, ficando próximo do observado na prática com redes Infiniband[71].

Na Figura 3.8 cinco filas são envolvidas, utilizamos uma fila para cada invocação de *kernel* e a sub-rotina `HLIB_MPI_SendRecv` utiliza uma fila interna para cópias de memória dispositivo⇒CPU e outra fila interna para cópias de memória CPU⇒dispositivo.

O *backend* regular (computação homogênea) invoca o MPI diretamente, pois só existe a memória principal, e neste caso a memória do dispositivo é alocada na memória principal.

Até recentemente não existia suporte MPI à comunicação direta entre GPUs CUDA[72]. Nosso *backend* CUDA implementa essa comunicação sem depender de tal suporte, usando as funções CUDA IPC *Inter-Process Communications*. Atualmente existem implementações CUDA-*aware* MPI[73, 74]. Estas implementações aceitam, de forma transparente, *buffers* alocados na GPU. A última versão do nosso *backend* CUDA apenas invoca o CUDA-*aware* MPI[73, 74] se o mesmo estiver disponível. Senão, ele utiliza o mesmo código dos outros *backends* heterogêneos. Definimos esta funcionalidade em tempo de compilação.

3.3 Avaliação

As aplicações do Capítulo 4, RTM e FWI da Petrobras, utilizam a HLIB com as arquiteturas CUDA e convencional (somente CPU). Com a aplicação RTM, realizamos uma prova de conceito utilizando a HLIB (*backend* hStreams) com placas Xeon Phi KNC.

Executamos a aplicação RTM com sucesso nas plataformas - CPU Power8 + GPU Nvidia, CPU Intel x86 + GPU Nvidia e apenas CPU Intel x86 - utilizando os compiladores Intel e IBM XL para a CPU e o nvcc para a GPU.

PUC-Rio - Certificação Digital Nº 1421599/CA

Os testes compreendem as versões do CUDA: 4, 5, 6.5 e 7.5.

Testamos nossas funções de troca de mensagens entre dispositivos com com diversas implementações MPI - OpenMPI, Intel, IBM e Cray - e diversos tipos de rede - Ethernet 10Gb, Infiniband e Cray.

Não tivemos que alterar nenhuma linha de código da HLIB para executar em todas essas plataformas.

Fizemos uma avaliação informal da HLIB em conjunto com um desenvolvedor usuário da HLIB. Constatamos alguns benefícios ao utilizar a HLIB:

- Curva de aprendizado rápida
- Alterar na biblioteca, resolve todos os projetos que a utilizam
- Encapsular boas práticas, como a verificação de invasão de memória da Seção 3.2.5
- Facilitar a portabilidade para outros paradigmas ex:OpenCL, mas é necessário reescrever o *kernel*.
- Suporte à troca de mensagens entre dispositivos (MPI)
- Apesar da Nvidia alegar que CUDA tem retrocompatibilidade, sabemos que não é sempre assim (a HLIB trata internamente diferentes versões de CUDA)

O desenvolvedor citou esse problema da retrocompatibilidade CUDA porque recentemente um outro desenvolvedor, que está codificando diretamente em CUDA, passou pelo mesmo problema que já havíamos solucionado na HLIB. Esse problema se refere a alocação de memória não paginável (Seção 3.2.3), e é difícil de encontrar. O código compila mas a aplicação aborta com um *runtime error*, que não informa a origem do erro.

3.4 Discussão

O uso da HLIB induz o desenvolvedor a dividir a aplicação em duas camadas, basta ele utilizar a mesma interface para definir o código *driver* que enfileira o *kernel* no dispositivo. Com isso, a camada agnóstica de arquitetura enfileira o *kernel* com uma única chamada para todas as arquiteturas.

A HLIB mantém a mesma abordagem das APIs CUDA, OpenCL e hStreams. Com a HLIB, o desenvolvedor utiliza o mesmos paradigmas dessas APIs. Cabe ao desenvolvedor gerenciar a memória do dispositivo e enfileirar execuções de *kernels* e as transferências entre a memória da CPU e a memória do dispositivo.

PUC-Rio - Certificação Digital Nº 1421599/CA

Se uma aplicação tira benefício de uma funcionalidade específica de uma arquitetura, então haverá uma perda de desempenho ao se utilizar a HLIB. Mas caso contrário, a perda de desempenho é apenas referente a uma invocação Fortran. A biblioteca se comporta como um *wrapper* Fortran 90 para as APIs específicas de cada arquitetura.

Poderemos no futuro, adicionar à HLIB extensões com funcionalidades específicas de cada arquitetura. Caso o desenvolvedor opte pelo uso das extensões, haverá uma perda de portabilidade na camada superior. Pois essa camada passa a tratar diferentes arquiteturas. O desenvolvedor também pode estender a HLIB com funcionalidades específicas.

Como cada dispositivo possui uma memória, assim como a CPU, escolhemos utilizar o MPI para agregar o poder computacional de múltiplos dispositivos, tendo em vista que o MPI já é utilizado em aplicações de memória distribuída. Nas aplicações do Capítulo 4, utilizamos um processo MPI por dispositivo. Com uma operação `MPI_Comm_split` encontramos o *rank* MPI dentro do nó, e com este *rank*, definimos o dispositivo a ser associado ao objeto contexto de cada processo MPI.

PUC-Rio - Certificação Digital Nº 1421599/CA

4
Aplicações de Alto Desempenho

Neste capítulo, apresentamos as aplicações de imageamento sísmico Migração Reversa no Tempo RTM (*Reverse Time Migration*) e a Inversão Completa do Campo de Onda FWI (*Full Waveform Inversion*), que implementamos com a estratégia de duas camadas e as ferramentas HLIB e OpenVec.

4.1
Migração Reversa no Tempo RTM

A migração reversa no tempo[5] RTM (*Reverse Time Migration*), tem como finalidade gerar uma imagem de subsuperfície. De forma simplificada, podemos compará-la a um exame de ultrassom. O aparelho de ultrassom emite ondas que são refletidas pelos diferentes tecidos orgânicos, e baseando-se no tempo de retorno de cada onda, posiciona cada reflexão gerando uma imagem. A migração RTM posiciona os eventos geológicos de forma mais correta que outras migrações que demandam menos poder computacional, como a migração Kirchhoff[75, 76]. Uma única execução da migração RTM pode ocupar uma máquina de um petaflop por três meses.

Um levantamento sísmico[77] pode ser visto na Figura 4.1. Este processo compreende milhares de experimentos, chamados de tiros (*seismic shots*), que registram nos sensores (receptores) as transições entre diferentes camadas geológicas. O levantamento sísmico da Figura 4.1 é realizado com o navio em movimento, ele se movimenta com velocidade constante ativando a fonte e registrando nos sensores a energia refletida pelas diferentes camadas geológicas. Cada registro de uma ativação da fonte representa um tiro e um conjunto de tiros compõe um dado sísmico. Quando posicionamos estes sensores em terra, os chamamos de geofones e quando os posicionamos no mar, os chamamos de hidrofones.

Um tiro é o processo onde uma fonte emite energia, e diversos sensores registram por T segundos essa energia que é propagada e refletida pela terra. Quando posicionada em terra, podemos implementar a fonte com uma detonação de uma carga de TNT, normalmente enterrada a alguns metros de profundidade. Nos levantamentos sísmicos marítimos, um canhão de ar comprimido acoplado ao fundo do casco do navio gera a energia da fonte. O contraste entre as diferentes camadas geológicas reflete parte dessa energia, e os sensores registram a porção da energia que retorna a superfície.

Antes de executar a RTM, os profissionais da equipe de processamento sísmico estimam o modelo de velocidades, que de forma aproximada, repre-

PUC-Rio - Certificação Digital Nº 1421599/CA

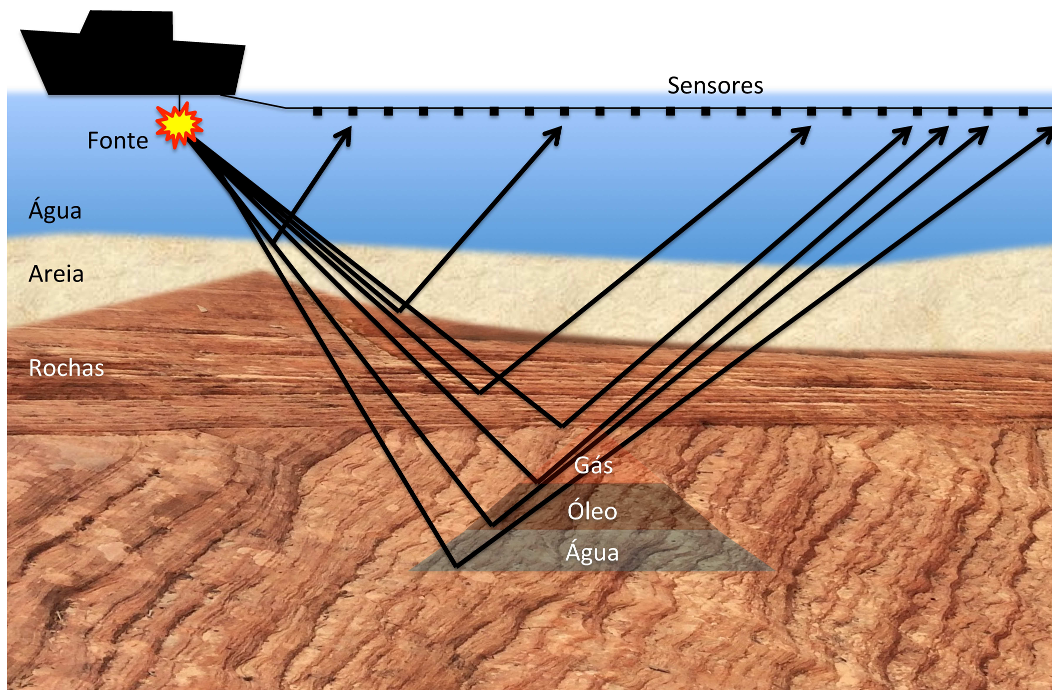

Figura 4.1: Levantamento sísmico marítimo. O navio puxa cabos com os sensores (hidrofones). Abaixo do casco do navio temos a fonte sísmica que é um canhão de ar comprimido. Parte da energia que a fonte produz é refletida pelas camadas geológicas do fundo do mar.

PUC-Rio - Certificação Digital Nº 1421599/CA

senta as velocidades de propagação da onda para as camadas geológicas do levantamento sísmico.

A migração RTM simula o caminho percorrido pela onda sísmica, que se inicia na posição da fonte e propaga em um volume ao redor do tiro. E também simula o caminho reverso da onda, que se inicia nas posições dos receptores (sensores) e utiliza os sinais registrados pelo levantamento sísmico. Ambas as simulações utilizam a Equação 4-1 para calcular a propagação da onda. A propagação leva em conta as velocidades estimadas do modelo de velocidade. A velocidade de propagação é o termo c da Equação 4-1.

$$\frac{1}{c^2}\frac{\partial^2 p}{\partial t^2} = \left(\frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial y^2} + \frac{\partial^2 p}{\partial z^2}\right) \tag{4-1}$$

Obtemos a posição do evento onde as ondas dos receptores e da fonte se encontram no mesmo passo de tempo (*time step*). Ou seja, a correlação das duas simulações (fonte e receptores) gera uma imagem 3D, conforme a Equação 4-2, onde S é o campo da simulação da fonte (*source*), R é o campo da simulação dos receptores e T é o tempo total de registro do tiro. De forma simplificada a migração RTM simula o caminho percorrido pelo sinal para encontrar a posição onde ocorreu a reflexão que originou o sinal. A imagem

final, resultado da migração, é a soma das imagens de todos os tiros, que podem ser processados de forma embaraçosamente paralela.

$$Img(x, y, z) = \sum_{t=0}^{T} S(x, y, z, t) \times R(x, y, z, t) \tag{4-2}$$

O código de propagação de onda que representa mais de 95% do tempo de execução, utiliza o método de diferenças finitas para simular o caminho percorrido pela onda. O cálculo do próximo instante na simulação utiliza o tempo atual e o anterior (derivada temporal de segunda ordem), e vários pontos vizinhos do tempo atual para calcular as derivadas espaciais x,y,z da Equação 4-1. O cálculo das derivadas espaciais utiliza um estêncil como o da Figura 4.2. Escolhemos este tamanho de estêncil para efetuarmos uma comparação direta de desempenho com o estado da arte[22].

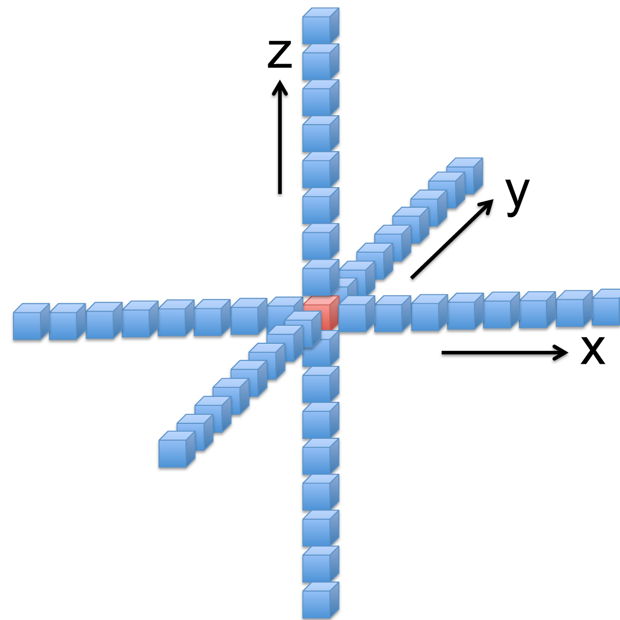

Figura 4.2: Estêncil para calcular derivadas espaciais de 16ª ordem. O cálculo da derivada no ponto central (em vermelho) depende dos pontos vizinhos (em azul).

Nos nossos testes, utilizamos o mesmo tamanho de cela para as três dimensões. Esta decisão deixa os pesos iguais para as três dimensões. Os pesos variam apenas em função da distância ao ponto central. Com isso, o cálculo de cada ponto central da Figura 4.2 requer 62 operações de ponto flutuante (conforme a Listagem 4.2 com `RAIO=8`). Com essa demanda de operações, temos que a migração reversa no tempo em um levantamento sísmico de 400.000 tiros requer 736 exa (10^{18}) operações de ponto flutuante, conforme a Tabela 4.1.

PUC-Rio - Certificação Digital Nº 1421599/CA

Número de Tiros	400000
Simulações por Tiro	2
Comprimento	1500
Largura	1100
Profundidade	750
Iterações (passos de tempo)	12000
Operações por Ponto	62
Total de Operações	736×10^{18}

Tabela 4.1: Estimativa da quantidade de operações de ponto flutuante para uma migração RTM. O total de operações é o produto das demais linhas da tabela. As dimensões comprimento, largura e profundidade compreendem um volume discretizado em uma grade regular 3D.

A estimativa da Tabela 4.1 leva em conta apenas a solução da Equação 4-1. Para cada tiro executamos duas simulações. Uma para o campo de onda da fonte, e outra para o campo de onda dos receptores.

4.2 Inversão Completa do Campo de Onda FWI

A Inversão Completa do Campo de Onda (*Full Waveform Inversion*) FWI[78, 79, 80, 81] gera o modelo de velocidade a partir dos dados coletados. Com esse modelo é possível fazer uma migração RTM mais precisa, pois esse modelo de velocidade é mais realista.

Chamamos de traço sísmico o sinal registrado por um receptor de um tiro, e associamos a um traço um único par $\{tiro, receptor\}$. Por exemplo, se o navio da Figura 4.1 puxa 1000 sensores podemos afirmar que cada tiro possui 1000 traços sísmicos.

O método FWI faz uma modelagem do traço sísmico de cada tiro e o compara com o respectivo traço registrado pelo levantamento sísmico. Essa comparação gera uma função de erro que é minimizada por um processo de otimização iterativo. Quanto mais próximo da realidade o modelo de velocidade estiver, menor é o erro entre o sinal modelado e o registrado.

Cada iteração desse processo faz uma migração RTM com o resíduo da comparação entre os traços modelados e registrados. O resultado é um gradiente que o método FWI utiliza como guia para gerar um novo modelo de velocidade a partir do modelo atual. Então o método aplica esse gradiente com vários pesos e faz uma busca linear para encontrar o melhor peso, gerando um novo modelo de velocidade que será utilizado pela próxima iteração. A Figura 4.3 apresenta os passos do método FWI. Como muitas iterações são necessárias para gerar um modelo de velocidade, o método FWI tem um custo computacional uma ordem de grandeza maior que a RTM.

PUC-Rio - Certificação Digital Nº 1421599/CA

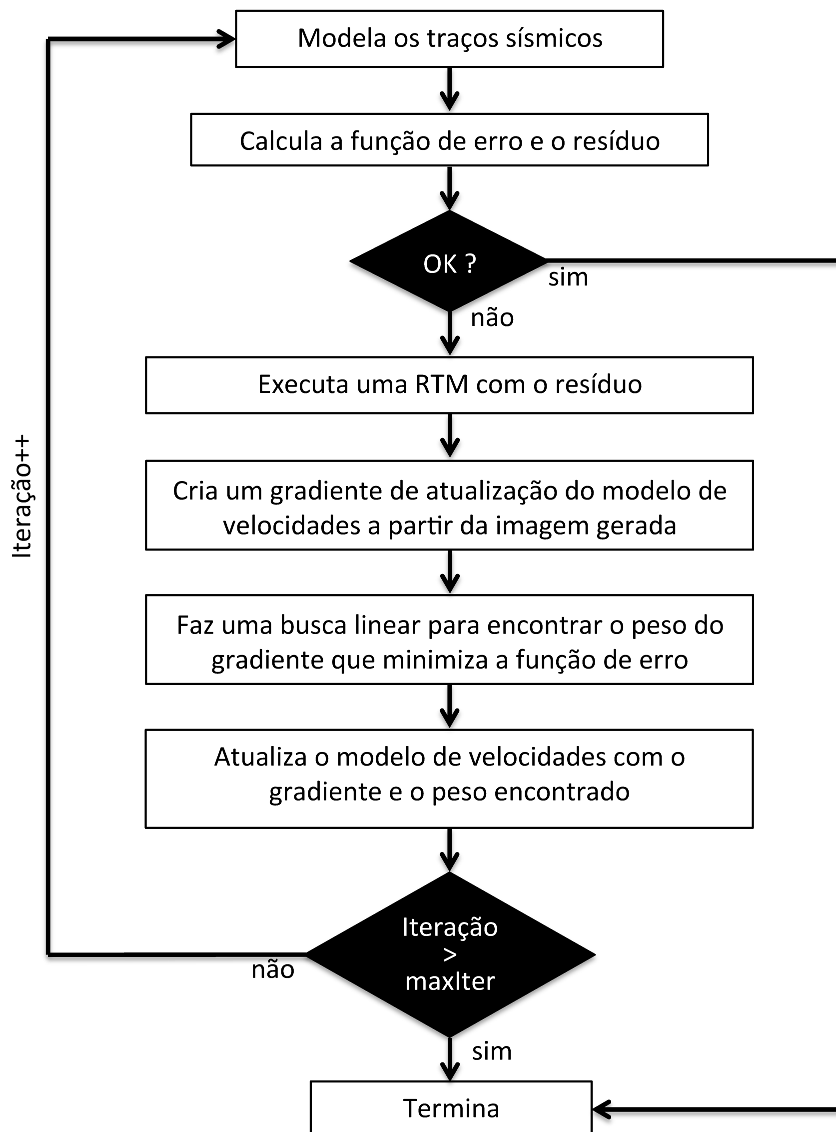

Figura 4.3: Fluxograma do método FWI.

PUC-Rio - Certificação Digital Nº 1421599/CA

4.3 Observações

As aplicações RTM e FWI demandam muito poder computacional. Uma execução da RTM pode ocupar uma máquina[1] petaflop por três meses. Como a FWI executa uma RTM e várias modelagens a cada iteração, sua demanda computacional é ainda maior.

O tempo de execução destas aplicações depende muito da parametrização do usuário e dos dados de entrada. Por exemplo, o tempo de execução é diretamente proporcional ao número de tiros, profundidade do modelo e tempo de registro. Com relação à parametrização do usuário, o parâmetro que mais afeta o tempo de execução é a frequência máxima, que define a maior

[1]Máquina com 2,6 petaflops de pico teórico em precisão simples com 300 GPUs Nvidia K80 e rede Infiniband FDR.

frequência presente na simulação. Quanto maior a frequência, maior o nível de detalhamento na imagem. O tempo de execução é proporcional a $freqmax^4$, pois para contemplarmos uma determinada frequência máxima precisamos respeitar os critérios de estabilidade e dispersão numérica do método de diferenças finitas, que dependem de quatro parâmetros: dx, dy, dz e dt, onde dx, dy e dz se referem ao tamanho da cela da grade 3D discretizada e dt se refere ao passo de tempo.

Para executar a FWI no mesmo parque computacional que executa a RTM podemos reduzir o conteúdo de frequência. Por exemplo, uma redução de 50% na frequência máxima, reduz o tempo de execução da simulação da propagação da onda em 16 vezes $\left(\frac{1}{2}\right)^4$. Normalmente a execução da FWI começa com um conteúdo de frequência baixo e aumenta a frequência máxima gradativamente com as iterações[82] do fluxo da Figura 4.3. Com isso, temos que as primeiras iterações do método FWI são menos custosas computacionalmente em relação a RTM pois utilizam frequências baixas como 3,5Hz e 7,2Hz de acrodo com o trabalho de Virieux e Operto[83].

PUC-Rio - Certificação Digital Nº 1421599/CA

4.4
Exemplo de Uso HLIB

Nesta seção, demonstramos o uso da HLIB em aplicações de imageamento sísmico. Estas aplicações demandam mais de um petaflop de capacidade de processamento. E podem utilizar mais de mil dispositivos em paralelo.

Em um ambiente paralelo, cada dispositivo recebe um trecho do volume sísmico. Nas aplicações RTM e FWI implementamos um *kernel* que simula a propagação do campo de ondas. Para calcular as derivadas espaciais, os pontos vizinhos são necessários, conforme a Figura 4.2.

Utilizamos as filas para sobrepor a computação com o envio (MPI) dos pontos necessários aos outros dispositivos. A Figura 4.4 mostra uma decomposição de domínio em uma dimensão com um processo trocando os pontos com seus vizinhos, satisfazendo as dependências do estêncil da Figura 4.2.

Seguindo a Figura 4.4 e a Listagem 4.1, após o $processo_i$ computar os pontos da borda (em verde claro), o dispositivo inicia a computação dos pontos do centro (em verde escuro), ao mesmo tempo em que a troca as bordas com seus dois vizinhos, utilizando duas chamadas `HLIB_MPI_sendRecv`, conforme a Figura 3.8.

Fizemos a decomposição de domínio com MPI, utilizando um dispositivo por *rank* MPI. A seguir o pseudocódigo do propagador de ondas paralelo com troca de bordas:

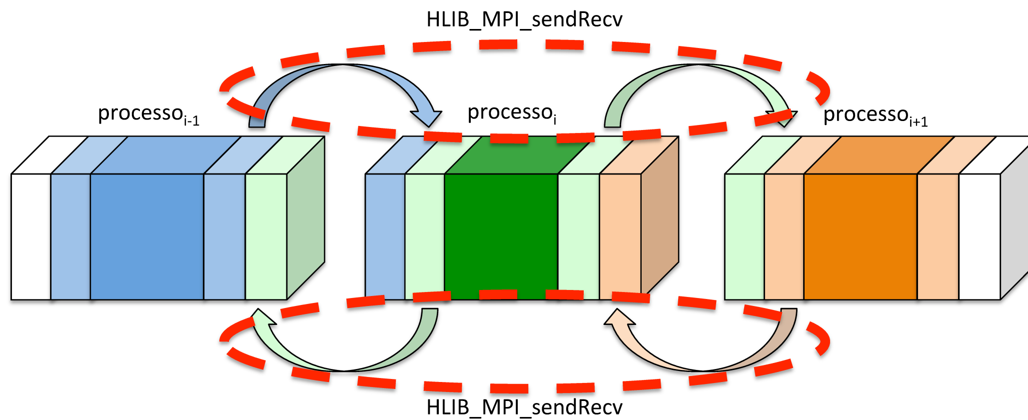

Figura 4.4: Troca de bordas do algoritmo de propagação de onda 3D por diferenças finitas, que é utilizado pelas aplicações RTM e FWI. Em verde claro os pontos das bordas necessários aos vizinhos do $processo_i$. Os pontos do centro que não são dependência para nenhum outro processo, em verde escuro. O $processo_i$ invoca `HLIB_MPI_sendRecv` duas vezes, conforme as setas.

Listagem 4.1: Pseudocódigo RTM/FWI para uma propagação de onda.

```
Para todas as iteracoes de tempo

   Seleciona a fila_esquerda
   Dispara o kernel na minha_borda_esquerda

   Seleciona a fila_direita
   Dispara o kernel na minha_borda_direita

   Seleciona a fila_centro
   Dispara o kernel nos pontos do centro

   Aguarda a fila_esquerda ficar vazia
   Aguarda a fila_direita  ficar vazia

   HLIB_MPI_sendRecv(minha_borda_esquerda,   meu_rank-1,
                     borda_direita_vizinho,  meu_rank+1)

   HLIB_MPI_sendRecv(minha_borda_direita,    meu_rank+1,
                     borda_esquerda_vizinho, meu_rank-1)

   Aguarda a fila_centro ficar vazia

Fim do para
```

Como o tempo de comunicação é menor que o tempo de computação, podemos afirmar que o tempo de comunicação não afeta o tempo total de execução. No *kernel* CUDA de propagação de onda, obtivemos 751 GFlops/s por placa Nvidia K80 (duas GPUs).

PUC-Rio - Certificação Digital Nº 1421599/CA

Utilizamos a o suporte da HLIB para troca de mensagens em um cluster com 2,6 petaflops de pico teórico com GPUs Nvidia K80 e rede Infiniband FDR capaz de transmitir dados a 56 Gb/s.

4.5 Exemplo de Uso OpenVec

As técnicas modernas de imageamento sísmico como a RTM e FWI utilizam o método de diferenças finitas (DF) para simular a propagação de um campo de onda em um volume geológico. Obter um ganho de desempenho nessas aplicações, traz um grande benefício, pois essas etapas podem consumir meses no fluxo de processamento sísmico utilizando clusters que custam dezenas de milhões de reais. Estas aplicações utilizam apenas operações de precisão simples (32 bits).

Os estêncis para o cálculo de DF aparecem em métodos iterativos para resolver equações diferenciais parciais, como a Equação 4-3.

$$\frac{1}{c^2}\frac{\partial^2 p}{\partial t^2} = \left(\frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial y^2} + \frac{\partial^2 p}{\partial z^2}\right) \tag{4-3}$$

Para um ponto qualquer em uma grade regular, a computação estêncil é uma soma ponderada com pesos fixos em um conjunto de pontos. A Figura 4.5a mostra os pontos necessários ao cálculo de uma derivada espacial de 8ª ordem para o ponto central (em vermelho). O cálculo da derivada temporal requer o ponto central do passo de tempo anterior e o ponto central do passo de tempo atual.

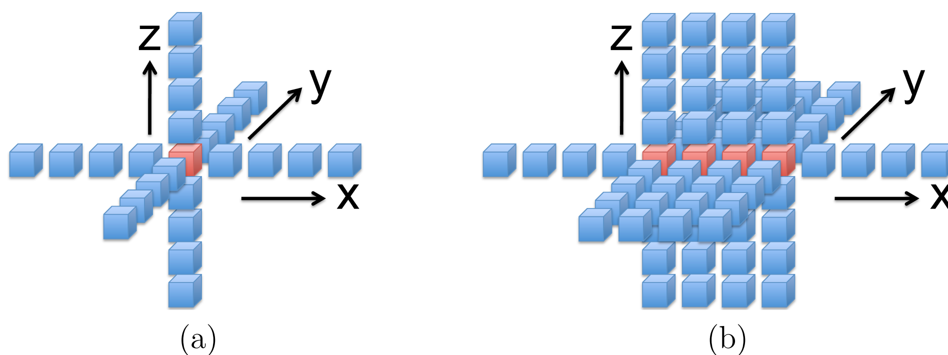

Figura 4.5: (a) um estêncil de 25 pontos (RAIO=4), obtemos o ponto central em vermelho com uma soma ponderada de todos os pontos. (b) 4 estêncis computados simultaneamente em uma arquitetura SIMD com 4 elementos por vetor.

Para realizar a simulação da propagação do campo de onda ao longo do tempo, utilizamos dois *arrays*, um que armazena o instante atual da simulação e outro que armazena o instante anterior. Calculamos o próximo instante de tempo sobrescrevendo o instante anterior e o instante atual passa a ser o

PUC-Rio - Certificação Digital Nº 1421599/CA

instante anterior com uma simples troca de ponteiros. Com isso mantemos o uso de memória em apenas dois *arrays* para armazenar o campo de onda.

Definimos o *array* onde o passo anterior e o próximo são armazenados como $U1(nx, ny, nz)$ e o *array* que armazena o passo de tempo atual como $U0(nx, ny, nz)$. Baseado na Figura 4.5a, no cálculo do ponto central, utilizamos os pontos em azul e o ponto central em vermelho do *array* $U0$ (instante atual) e apenas o ponto central do *array* $U1$ (instante anterior).

Listagem 4.2: Pseudocódigo da propagação de onda.

```
Para todos os passos de tempo
  Para todos os pontos
    novo = W(0) * U0(ix,iy,iz) ! Aplica o peso do ponto central

    ! Derivadas espaciais utilizando os pontos vizinhos
    Para k=1, RAIO;
      novo += W(k) * (
              U0(ix+k, iy,   iz  ) + U0(ix-k, iy,   iz  ) +
              U0(ix,   iy+k, iz  ) + U0(ix,   iy-k, iz  ) +
              U0(ix,   iy,   iz+k) + U0(ix,   iy,   iz-k))
    Fim do para k

    ! Sobrescreve o passo anterior com o novo
    U1(ix,iy,iz) = P(ix,iy,iz)*P(ix,iy,iz)*novo +
                   2*U0(ix,iy,iz) - U1(ix,iy,iz)

  Fim do para todos os pontos

  Troca ponteiros U1 <-> U0

Fim do para todos os passos de tempo
```

Na Listagem 4.2, $W(k)$ é o peso da soma ponderada para um ponto do estêncil com distância k do ponto central. $P(ix, iy, iz)$ representa uma propriedade do modelo geológico. Neste caso a velocidade de propagação da onda. $U0(ix, iy, iz)$ e $U1(ix, iy, iz)$ são os arrays representando dois passos de tempo.

Podemos notar que, para o mesmo passo de tempo, os pontos podem ser calculados em paralelo de forma independente, mas não podemos iniciar o próximo passo em um ponto sem que todos os seus pontos vizinhos estejam calculados.

Utilizamos o OpenVec para para calcular simultaneamente vários pontos na direção x, conforme as Figuras 4.5b e 4.6. A dimensão x está contígua em

PUC-Rio - Certificação Digital Nº 1421599/CA

memória. Modificamos o tamanho nx de todos os *arrays* para ser múltiplo de `OV_FLOAT_WIDTH`, garantindo com isso o alinhamento de memória.

Apenas os acessos com *offset* $ix \pm k$ podem estar desalinhados na memória, como por exemplo o termo `U0(ix+k, iy, iz)`. Os acessos $iy \pm k$ saltam $\pm k \times nx$ elementos e os acessos $iz \pm k$ saltam $\pm k \times nx \times ny$ elementos. Como ambos os saltos são múltiplos de nx, que é múltiplo de `OV_FLOAT_WIDTH`, todos esses acessos são alinhados em memória.

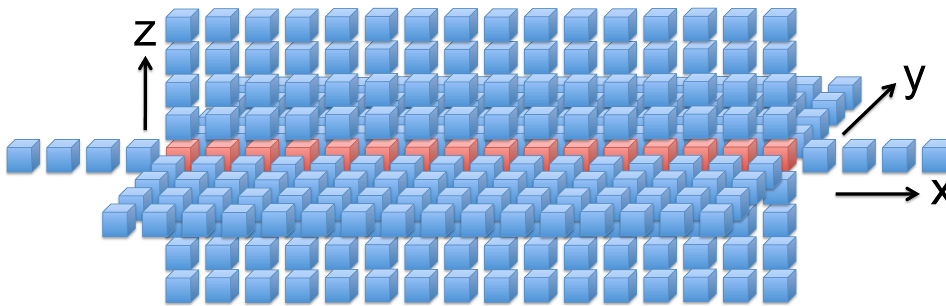

Figura 4.6: Cálculo simultâneo de 16 elementos em uma arquitetura SIMD com 16 elementos por vetor, como o Intel Xeon Phi.

Testamos o OpenVec na aplicação RTM (*reverse time migration*) (que também utiliza a HLIB) em um cluster com 200 nós *dual socket* AVX-2.

4.6 Avaliação de Desempenho OpenVec

Comparamos nossa implementação de um estêncil de 16ª ordem com o trabalho referência[22]. Escolhemos este tamanho de estêncil para poder fazer uma comparação direta com o trabalho de referência. Implementamos o estêncil utilizando o OpenVec em ambas as linguagens C e C++, com e sem *unroll* explícito. Tanto a nossa implementação, quanto a implementação de referência[22], utilizam o OpenMP para utilizar todos os núcleos da CPU. A Figura 4.7 mostra o comparativo de desempenho.

Nosso melhor resultado atingiu 2506 milhões de estêncis por segundo contra 2681 da implementação de referência. Atingimos 93% do desempenho da implementação de referência[22], que utiliza *autotuning* em conjunto com um algoritmo genético. O desempenho dos códigos em C, que somente utilizam intrínsecos, foi melhor que o desempenho dos códigos em C++, que utilizam os operadores matemáticos e intrínsecos.

O *unroll* explícito aumenta o desempenho no compilador gcc mas penaliza o desempenho no compilador da Intel.

Nossa implementação do estêncil faz parte dos exemplos de uso da distribuição do OpenVec.

PUC-Rio - Certificação Digital Nº 1421599/CA

OpenVec	
Compilador	Msamples/s
gcc	2217.1
gcc unroll	2413.3
g++ unroll	2147.7
g++	2081.9
icpc	2389.8
icpc unroll	2321.0
icc	2506.1
icc unroll	2483.7

Código Referência AVX-2	
Compilador	Msamples/s
icc	2681.6

Figura 4.7: Desempenho do estêncil de 16ª ordem com OpenVec+OpenMP e com o código de referência, que utiliza intrínsecos AVX-2 e OpenMP. Ambiente do teste: processador Xeon E5-2697v3 @ 2.60 GHz com 14 núcleos, compilador da Intel versão 15.0.1.133 e gcc 4.4.7.

PUC-Rio - Certificação Digital Nº 1421599/CA

4.7
Uso Combinado HLIB + OpenVec

Neste capítulo exploramos o uso combinado da HLIB com o OpenVec. Utilizamos a aplicação migração reversa no tempo (RTM) para demonstrar o uso combinado dessas duas ferramentas. A Figura 4.8 mostra as camadas da RTM e a integração com a HLIB e o OpenVec.

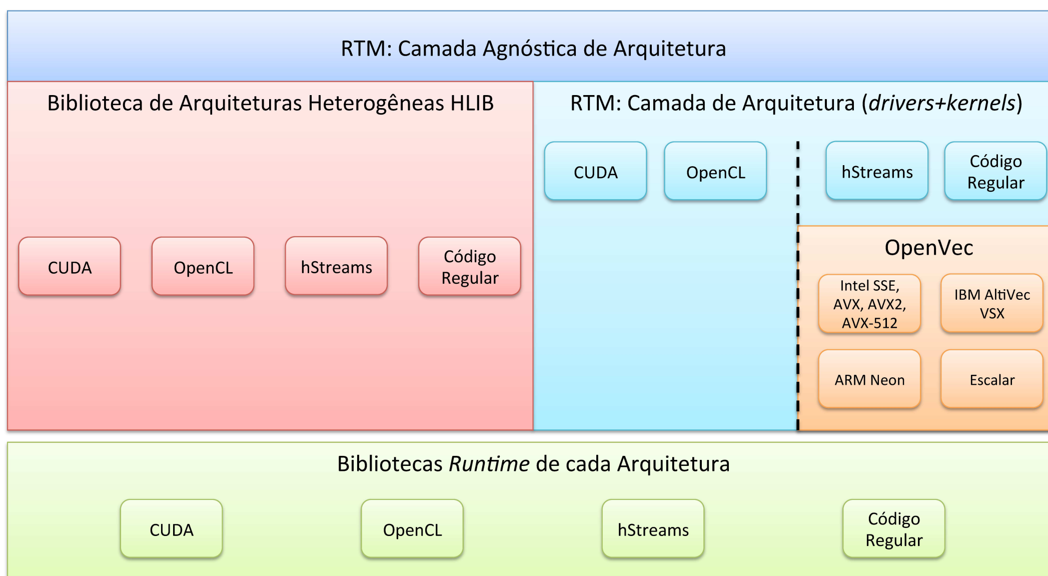

Figura 4.8: Camadas da RTM com HLIB+OpenVec. A RTM utiliza a HLIB para fazer o gerenciamento da computação heterogênea. Codificamos os *kernels* das arquiteturas hStreams e regular com o OpenVec.

A Figura 4.9 mostra em detalhe a camada de arquitetura da RTM. Os *drivers* têm interfaces idênticas e compatíveis com Fortran 90. Cada driver repassa os argumentos e invoca seu respectivo *kernel*. O *kernel* OpenVec é agnóstico de arquitetura e pode executar em um dispositivo secundário como uma placa Xeon Phi ou em um processador tradicional, SIMD ou escalar.

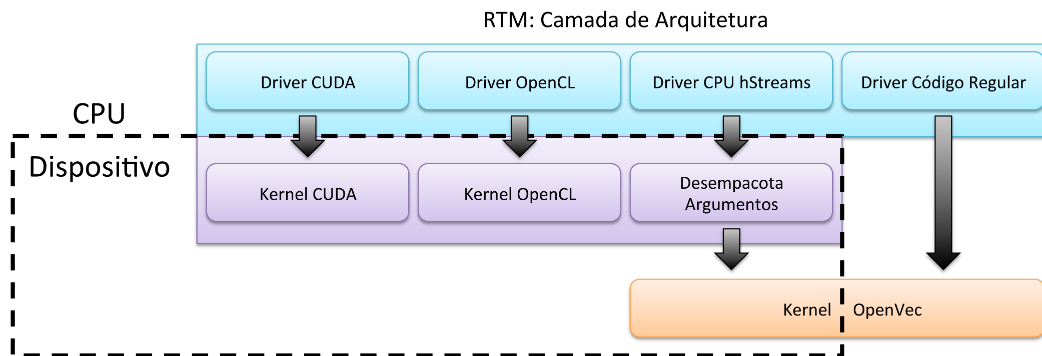

Figura 4.9: Camada de arquitetura da RTM. As setas indicam a direção de invocação. Por exemplo, o *driver* do código regular invoca o *kernel* OpenVec.

Suportamos as arquiteturas hStreams e regular com um único *kernel*, que utiliza o OpenVec. A invocação hStreams é diferente do código regular pois o *kernel* executa em um dispositivo enquanto que no código regular o *kernel* executa na CPU. Com isso, nosso *driver* hStreams, que executa na CPU, empacota os argumentos e dispara o *kernel* no dispositivo com a função `hStreams_EnqueueCompute`.

A maneira como o hStreams prepara os argumentos e dispara o *kernel* é similar ao OpenCL. Como mantivemos o mesmo *kernel* nos *backends* regular e hStreams. No *backend* hStreams foi necessário criar um código, que executa no dispositivo, e desempacota os argumentos antes de invocar o *kernel*, conforme a Figura 4.9. Este código de ligação mantem o *kernel* livre de chamadas proprietárias hStreams. A implementação atual do hStreams só permite argumentos do tipo `uint64_t`, por isso precisamos empacotar e desempacotar os argumentos nas invocações de *kernels*.

4.8 Experimentos

Comparamos o desempenho do estêncil de 16ª ordem da Listagem 4.2 com diversos dispositivos e APIs. A Tabela 4.2 mostra o desempenho em GFlops para cada API e dispositivo.

Executamos o nosso *kernel* CUDA no primeiro experimento. Este *kernel* possui otimizações específicas para arquiteturas de GPU iguais ou posteriores à

PUC-Rio - Certificação Digital Nº 1421599/CA

	API	Dispositivo	GFlops	Ano
1	CUDA	Nvidia K80 (dual)	751	2014
2	OpenCL	Nvidia K80 (dual)	266	2014
3	OpenMP 4.0	Nvidia K80 (dual)	31	2014
4	OpenVec+OpenMP	Intel Xeon E5-2697v3@2.6 Ghz	155	2014
5	OpenVec+OpenMP	Intel Xeon Phi 7120	262	2013
6	OpenCL	AMD Radeon HD 7970	236	2012

Tabela 4.2: Experimentos com diversos dispositivos e APIs. A primeira coluna se refere ao número do experimento, e a última ao ano de lançamento do dispositivo.

Kepler. Como por exemplo, acessar a memória `__shared__` utilizando palavras de 64bits[84] (`cudaSharedMemBankSizeEightByte`).

Os experimentos 2 e 6 tratam da execução de um *kernel* OpenCL inicialmente desenvolvido para suportar as GPUs da AMD. Executamos este *kernel* em uma GPU Nvidia (experimento 2) e em uma GPU AMD (experimento 6).

No experimento 3, utilizamos o OpenMP 4 sem a HLIB. Baixamos o trunk do projeto `clang-omp`. Geramos uma versão do compilador `llvm` com suporte ao OpenMP 4 com *offload* para GPUs Nvidia. O código OpenMP 4 ficou muito mais simples que o código HLIB+CUDA/OpenCL. Mas o desempenho ficou muito menor que as versões CUDA e OpenCL para o mesmo dispositivo (experimentos 1 e 2).

Traçamos um perfil da execução do experimento 3 com a ferramenta `nvprof` da Nvidia. Constatamos que a maior parte do tempo de execução foi gasto em cópias de memória entre a CPU e a GPU. Utilizamos a *keyword* `target data` para mover os dados para a GPU antes do *loop* de passos de tempo, e para mover os dados de volta apenas ao término do *loop*. Mas aparentemente, o compilador não respeitou a região `target data`, e transfere os dados em todas as iterações do *loop*.

No experimento 4, executamos o *kernel* OpenVec+OpenMP em um processador Xeon tradicional (homogêneo). Este processador possui instruções SIMD do tipo AVX-2. Este experimento é o mesmo da Figura 4.7, que atingiu 2506,1 milhões de amostras por segundo.

No experimento 5, executamos um *kernel* OpenVec+OpenMP otimizado para o Xeon Phi KNC. Utilizamos o Xeon Phi no modo *offload* com HLIB+hStreams. Esta versão do *kernel* utiliza o intrínseco `ov_stream_stf`, que equivale ao intrínseco `_mm512_storenr_ps` do Xeon Phi KNC. Como o hStreams ainda está na versão alfa (pré-beta) o desempenho poderá sofrer alterações em futuras versões.

Com exceção dos experimentos 2 e 3, em todos os outros experimentos

PUC-Rio - Certificação Digital Nº 1421599/CA

otimizamos os *kernels* para o dispositivo do experimento.

4.9 Discussão

Com relação ao desempenho, as aplicações RTM e FWI não tiram proveito de nenhuma funcionalidade não implementada pela HLIB e pelo OpenVec. As decisões de projeto foram guiadas para atender estas duas aplicações. O uso das nossas ferramentas nestas aplicações trouxe apenas um ganho de portabilidade, pois iríamos implementá-las de forma nativa se não tivéssemos estas ferramentas.

Estas duas aplicações são muito utilizadas pela indústria de Óleo e Gás. Elas executam em parques computacionais com custos da ordem de dezenas de milhões de reais. Tentar extrair o máximo de desempenho de cada arquitetura traz um diferencial competitivo. Mas outras aplicações podem obter um desempenho maior utilizando funcionalidades presentes em apenas uma arquitetura. Para solucionar este problema, sugerimos o uso de extensões para contemplar estas aplicações. O uso das extensões adiciona chamadas específicas na camada agnóstica de arquitetura, mas uma parte do código ainda fica portável.

PUC-Rio - Certificação Digital Nº 1421599/CA

5
Conclusões

A portabilidade e a manutenção de código são muito importantes no desenvolvimento de aplicações de alto desempenho, que podem ter uma vida útil de décadas[85, 86]. Por exemplo, há mais de 20 anos a Petrobras utiliza a Migração Kirchhoff, que foi otimizada ao longo dos anos para as seguintes arquiteturas: IBM Vector Facility, Silicon Graphics SMP, *clusters* de x86, IBM Cell[21] e *clusters* de GPUs Nvidia[42]

Como vimos, estas aplicações podem ser otimizadas para diferentes gerações de *hardware*. A nossa proposta de dividir a aplicação em uma camada de arquitetura e outra camada agnóstica de arquitetura ajudou a organizar e manter um código de alto desempenho para diversas arquiteturas.

Conseguimos, com a a nossa biblioteca HLIB, expor uma interface portável para as primitivas da computação heterogênea. Nosso foco foi manter a mesma abstração de APIs já existentes, como CUDA e OpenCL. Com isso, o desenvolvedor utiliza as mesmas primitivas que, por exemplo, em CUDA, mas escreve o código de gerenciamento uma única vez para todas as arquiteturas suportadas, incluindo a arquitetura tradicional que possui apenas a CPU.

Como a nossa biblioteca não é a responsável por escalonar os *kernels* e as movimentações entre a memória da CPU e a memória do dispositivo, fica a cargo do desenvolvedor atingir o desempenho potencial de cada arquitetura. Também cabe ao desenvolvedor implementar o *kernel* e o respectivo código *driver* (que enfileira o *kernel*) nas arquiteturas de interesse.

Demonstramos o uso da HLIB com GPUs Nvidia e a aplicação RTM de produção da Petrobras em um *cluster* de 2,6 petaflops de pico teórico. No *kernel* CUDA de propagação de onda, obtivemos 751 GFlops/s por placa Nvidia K80. Nas aplicações do Capítulo 4, não encontramos a necessidade de utilizar nenhuma funcionalidade não suportada pela HLIB. Portanto, não faz sentido avaliar o desempenho da HLIB, pois estaríamos medindo apenas o *overhead* da invocação Fortran, tendo em vista que os *kernels* estão implementados de forma nativa, e que a biblioteca é apenas um *wrapper* Fortran 90 para as APIs CUDA, OpenCL e hStreams. Mas pode haver uma perda de desempenho em uma aplicação que tira proveito de uma funcionalidade específica de uma determinada arquitetura.

A remoção de otimizações específicas para uma determinada arquitetura em uma aplicação de alto desempenho tende a deixar marcas no código, que vão tornando cada vez mais difícil a sua manutenção. Acreditamos que essa divisão em camadas e o uso da HLIB, além de facilitar a adição de novas arquiteturas, também facilita a remoção de arquiteturas.

PUC-Rio - Certificação Digital Nº 1421599/CA

Como é difícil prever a evolução do mercado de aceleradores. Não há garantias de que a HLIB poderá ser portada para futuras gerações de arquiteturas heterogêneas.

O OpenVec atingiu o objetivo de implementar uma forma de vetorização explícita, que é agnóstica em relação a arquitetura e ao tamanho do vetor SIMD.

Se compararmos os códigos do estêncil OpenVec C++ com a versão que utiliza intrínsecos proprietários, podemos notar um aumento na produtividade de codificação. Com o OpenVec C++, o desenvolvedor escreve apenas um código para todas as arquiteturas, e o nosso *overload* de operadores facilita a codificação, a leitura e a manutenção do código. O desenvolvedor codifica de uma maneira próxima da tradicional (escalar), quando utiliza os operadores do OpenVec C++.

Em termos de desempenho, o OpenVec apresentou baixo *overhead* pois utiliza macros para redefinir intrínsecos. Nossa implementação do estêncil de diferenças finitas com OpenMP+OpenVec atingiu 93% (155 GFlops/s) do desempenho do estado da arte[22] para CPUs SIMD. Este desempenho, ainda fica distante dos 751 GFlops/s que obtivemos com uma GPU Nvidia K80 utilizando CUDA.

Testamos com sucesso nossa implementação em diversos sistemas operacionais, utilizando diversos compiladores, e em um conjunto de arquiteturas SIMD que compreendem os últimos 17 anos. Também testamos nossa implementação, através de um simulador, para a arquitetura Intel AVX-512 que ainda não foi lançada no mercado. Inspecionamos o código *assembly* para ter certeza que o compilador está gerando instruções vetoriais a partir de nossos intrínsecos e operadores.

Levamos apenas três dias para criar e adicionar diversos *kernels* OpenVec necessários para a aplicação RTM, que já utiliza a HLIB. Testamos com sucesso o uso combinado da HLIB+OpenVec na RTM, em um *cluster* com 200 nós, cada um com duas CPUs AVX 2.

Mas a vetorização explícita é apenas um dos tópicos necessários na difícil tarefa de otimização de código para as CPUs modernas. Cabe ao desenvolvedor explorar os demais tópicos como *threading, blocking, array padding, loop unrolling, prefetching, aliasing*, e NUMA+*page awareness*, e aplicar tudo isso com uma estratégia de *autotuning*. Podemos notar essa dificuldade analisando o desempenho do estêncil OpenVec para a mesma arquitetura (AVX 2), onde a versão com *unroll* explícito, teve um desempenho melhor em relação ao código sem *unroll* explícito nos compiladores `gcc/g++`, e um desempenho pior com os compiladores da Intel `icc/icpc`.

PUC-Rio - Certificação Digital Nº 1421599/CA

Demonstramos que o uso da estratégia de dividir a aplicação em duas camadas em conjunto com o uso da HLIB (que implementa uma terceira camada) e do OpenVec auxiliaram na portabilidade de código em uma aplicação de alto desempenho.

PUC-Rio - Certificação Digital Nº 1421599/CA

6
Referências Bibliográficas

[1] COURTLAND, R. Intel strikes back [news]. **Spectrum, IEEE**, v. 50, n. 8, p. 14–14, August 2013. ISSN 0018-9235.

[2] SHALF, J.; DOSANJH, S.; MORRISON, J. Exascale computing technology challenges. In: **Proceedings of the 9th International Conference on High Performance Computing for Computational Science**. Berlin, Heidelberg: Springer-Verlag, 2011. (VECPAR'10), p. 1–25. ISBN 978-3-642-19327-9. Disponível em: <http://dl.acm.org/citation.cfm?id=1964238.1964240>.

[3] TOP500. **TOP500 Supercomputer Site**. 2016. Disponível em: <http://www.top500.org>.

[4] HOSHINO, T. et al. CUDA vs OpenACC: Performance case studies with kernel benchmarks and a memory-bound cfd application. In: IEEE. **Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on**. [S.l.], 2013. p. 136–143.

[5] BAYSAL, E.; KOSLOFF, D. D.; SHERWOOD, J. W. Reverse time migration. **Geophysics**, Society of Exploration Geophysicists, v. 48, n. 11, p. 1514–1524, 1983.

[6] KRISHNAMURTHY, R. K. et al. High-performance and low-power challenges for sub-70 nm microprocessor circuits. In: **Proc. IEEE Custom Integrated Circuits Conf**. [S.l.: s.n.], 2002. v. 125, p. 12–15.

[7] HENNESSY, J. L.; PATTERSON, D. A. **Computer Organization and Design (2Nd Ed.): The Hardware/Software Interface**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998. 751 p. ISBN 1-55860-428-6.

[8] FRANCHETTI, F. et al. Efficient utilization of SIMD extensions. **Proceedings of the IEEE**, v. 93, n. 2, p. 409–425, Feb 2005. ISSN 0018-9219.

[9] LARSEN, S.; AMARASINGHE, S. Exploiting superword level parallelism with multimedia instruction sets. In: **Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation**. New York, NY, USA: ACM, 2000. (PLDI '00), p. 145–156. ISBN 1-58113-199-2. Disponível em: <http://doi.acm.org/10.1145/349299.349320>.

PUC-Rio - Certificação Digital Nº 1421599/CA

[10] EICHENBERGER, A. E.; WU, P.; O'BRIEN, K. Vectorization for SIMD architectures with alignment constraints. In: ACM. **ACM SIGPLAN Notices**. [S.l.], 2004. v. 39, n. 6, p. 82–93.

[11] NUZMAN, D.; HENDERSON, R. Multi-platform auto-vectorization. In: IEEE COMPUTER SOCIETY. **Proceedings of the International Symposium on Code Generation and Optimization**. [S.l.], 2006. p. 281–294.

[12] WU, P.; EICHENBERGER, A. E.; WANG, A. Efficient SIMD code generation for runtime alignment and length conversion. In: IEEE. **Code Generation and Optimization, 2005. CGO 2005. International Symposium on**. [S.l.], 2005. p. 153–164.

[13] WU, P. et al. An integrated simdization framework using virtual vectors. In: ACM. **Proceedings of the 19th annual international conference on Supercomputing**. [S.l.], 2005. p. 169–178.

[14] NAISHLOS, D. et al. Vectorizing for a SIMdD DSP architecture. In: ACM. **Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems**. [S.l.], 2003. p. 2–11.

[15] NUZMAN, D.; ROSEN, I.; ZAKS, A. Auto-vectorization of interleaved data for SIMD. In: ACM. **ACM SIGPLAN Notices**. [S.l.], 2006. v. 41, n. 6, p. 132–143.

[16] KARRENBERG, R.; HACK, S. Whole-function vectorization. In: IEEE. **Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on**. [S.l.], 2011. p. 141–150.

[17] NUZMAN, D.; ZAKS, A. Outer-loop vectorization: revisited for short SIMD architectures. In: ACM. **Proceedings of the 17th international conference on Parallel architectures and compilation techniques**. [S.l.], 2008. p. 2–11.

[18] REN, G.; WU, P.; PADUA, D. An empirical study on the vectorization of multimedia applications for multimedia extensions. In: **Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International**. [S.l.: s.n.], 2005. p. 89b–89b.

[19] MALEKI, S. et al. An evaluation of vectorizing compilers. In: **Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on**. [S.l.: s.n.], 2011. p. 372–382. ISSN 1089-795X.

PUC-Rio - Certificação Digital Nº 1421599/CA

[20] FRANCHETTI, F.; PUSCHEL, M. Short vector code generation for the discrete fourier transform. In: **Parallel and Distributed Processing Symposium, 2003. Proceedings. International**. [S.l.: s.n.], 2003. p. 10 pp.–. ISSN 1530-2075.

[21] PANETTA, J. et al. Computational characteristics of production seismic migration and its performance on novel processor architectures. In: **Computer Architecture and High Performance Computing, 2007. SBAC-PAD 2007. 19th International Symposium on**. [S.l.: s.n.], 2007. p. 11–18. ISSN 1550-6533.

[22] ANDREOLLI, C. et al. Chapter 23 - characterization and optimization methodology applied to stencil computations. In: JEFFERS, J. R. (Ed.). **High Performance Parallelism Pearls**. Boston: Morgan Kaufmann, 2015. p. 377 – 396. ISBN 978-0-12-802118-7. Disponível em: <http://www.sciencedirect.com/science/article/pii/B9780128021187000236>.

[23] ARAYA-POLO, M. et al. Assessing accelerator-based HPC reverse time migration. **Parallel and Distributed Systems, IEEE Transactions on**, v. 22, n. 1, p. 147–162, Jan 2011. ISSN 1045-9219.

[24] ZHENG, Y. et al. Accelerating with 512-bit SIMD : A case study for molecular dynamics simulation on Intel's Knights Corner. In: **Communications and Information Technology (ICCIT), 2013 Third International Conference on**. [S.l.: s.n.], 2013. p. 195–200.

[25] WILLIAMS, S. et al. Optimization of geometric multigrid for emerging multi-and manycore processors. In: IEEE COMPUTER SOCIETY PRESS. **Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis**. [S.l.], 2012. p. 96.

[26] WILLIAMS, S. et al. Optimization of sparse matrix–vector multiplication on emerging multicore platforms. **Parallel Computing**, Elsevier, v. 35, n. 3, p. 178–194, 2009.

[27] TALLA, D. et al. Evaluating signal processing and multimedia applications on SIMD, VLIW and superscalar architectures. In: **Computer Design, 2000. Proceedings. 2000 International Conference on**. [S.l.: s.n.], 2000. p. 163–172. ISSN 1063-6404.

[28] KURZAK, J.; BUTTARI, A. Introduction to programming high performance applications on the Cell Broadband Engine. In: **High-Performance Inter-**

PUC-Rio - Certificação Digital Nº 1421599/CA

connects, 2007. HOTI 2007. 15th Annual IEEE Symposium on. [S.l.: s.n.], 2007. p. 11–11. ISSN 1550-4794.

[29] WEST, N.; GEIGER, D.; SCHEETS, G. Accelerating software radio on ARM: Adding NEON support to VOLK. In: **Radio and Wireless Symposium (RWS), 2015 IEEE**. [S.l.: s.n.], 2015. p. 174–176.

[30] MITRA, G. et al. Use of SIMD vector operations to accelerate application code performance on low-powered ARM and intel platforms. In: **Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International**. [S.l.: s.n.], 2013. p. 1107–1116.

[31] WILLIAMS, S. et al. The potential of the Cell processor for scientific computing. In: **Proceedings of the 3rd Conference on Computing Frontiers**. New York, NY, USA: ACM, 2006. (CF '06), p. 9–20. ISBN 1-59593-302-6. Disponível em: <http://doi.acm.org/10.1145/1128022.1128027>.

[32] WELCH, E. et al. A study of the use of SIMD instructions for two image processing algorithms. In: **Image Processing Workshop (WNYIPW), 2012 Western New York**. [S.l.: s.n.], 2012. p. 21–24.

[33] DATTA, K. et al. In: KURZAK, J.; BADER, D.; DONGARRA, J. (Ed.). **Scientific Computing with Multicore and Accelerators**. [S.l.]: Chapman and Hall/CRC, 2010. p. 234–235.

[34] SOUZA, P. et al. Chapter 24 - portable explicit vectorization intrinsics. In: JEFFERS, J. R. (Ed.). **High Performance Parallelism Pearls**. second. Boston: Morgan Kaufmann, 2015. p. 377 – 396. ISBN 978-0-12-802118-7. Disponível em: <http://www.sciencedirect.com/science/article/pii/B9780128021187000236>.

[35] SOUZA, P. et al. OpenVec portable SIMD intrinsics. In: **Second EAGE Workshop on High Performance Computing for Upstream**. [S.l.: s.n.], 2015.

[36] SHAHBAHRAMI, A.; JUURLINK, B.; VASSILIADIS, S. Performance impact of misaligned accesses in SIMD extensions. In: **Proceedings of the 17th Annual Workshop on Circuits, Systems and Signal Processing (ProRISC 2006)**. [S.l.: s.n.], 2006. p. 334–342.

[37] HARRIS, M. **How to Access Global Memory Efficiently in CUDA C/C++ Kernels**. 2013. Disponível em:

PUC-Rio - Certificação Digital Nº 1421599/CA

<http://devblogs.nvidia.com/parallelforall/how-access-global-memory-efficiently-cuda-c-kernels/>.

[38] TILLET, P. et al. Towards performance-portable, scalable, and convenient linear algebra. In: **HotPar**. [S.l.: s.n.], 2013.

[39] BYUN, J.-H. et al. Autotuning sparse matrix-vector multiplication for multicore. **ACM Trans. Math. Software, submitted**, 2012.

[40] CABALLERO, D. et al. Optimizing fully anisotropic elastic propagation on intel xeon phi coprocessors. In: **Second EAGE Workshop on High Performance Computing for Upstream**. [S.l.: s.n.], 2015.

[41] MIT. **The MIT License (MIT)**. Disponível em: <https://opensource.org/licenses/MIT>.

[42] PANETTA, J. et al. Accelerating kirchhoff migration by CPU and GPU cooperation. In: **Proceedings of the 2009 21st International Symposium on Computer Architecture and High Performance Computing**. Washington, DC, USA: IEEE Computer Society, 2009. (SBAC-PAD '09), p. 26–32. ISBN 978-0-7695-3857-0. Disponível em: <http://dx.doi.org/10.1109/SBAC-PAD.2009.29>.

[43] PANETTA, J. et al. Accelerating time and depth seismic migration by CPU and GPU cooperation. **International Journal of Parallel Programming**, Springer US, v. 40, n. 3, p. 290–312, 2012. ISSN 0885-7458. Disponível em: <http://dx.doi.org/10.1007/s10766-011-0185-2>.

[44] MARTINS, L. et al. Accelerating curvature estimate in 3d seismic data using GPGPU. In: **Computer Architecture and High Performance Computing (SBAC-PAD), 2014 IEEE 26th International Symposium on**. [S.l.: s.n.], 2014. p. 105–111. ISSN 1550-6533.

[45] OWENS, J. et al. GPU computing. **Proceedings of the IEEE**, v. 96, n. 5, p. 879–899, May 2008. ISSN 0018-9219.

[46] NVIDIA. **CUDA C Programming Guide**. 2015. Disponível em: <https://docs.nvidia.com/cuda/cuda-c-programming-guide/>.

[47] STEUWER, M.; KEGEL, P.; GORLATCH, S. Skelcl-a portable skeleton library for high-level GPU programming. In: IEEE. **Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on**. [S.l.], 2011. p. 1176–1182.

PUC-Rio - Certificação Digital Nº 1421599/CA

[48] MARTINEZ, G.; GARDNER, M.; FENG, W.-c. Cu2cl: A CUDA-to-OpenCL translator for multi-and many-core architectures. In: IEEE. **Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on**. [S.l.], 2011. p. 300–307.

[49] DEMIDOV, D. et al. Programming CUDA and OpenCL: A case study using modern C++ libraries. **SIAM Journal on Scientific Computing**, SIAM, v. 35, n. 5, p. C453–C472, 2013.

[50] REYES, R. et al. accull: An user-directed approach to heterogeneous programming. In: **ISPA**. IEEE, 2012. p. 654–661. ISBN 978-1-4673-1631-6. Disponível em: <http://dblp.uni-trier.de/db/conf/ispa/ispa2012.html#ReyesLFS12>.

[51] HERNANDEZ, O. et al. Facing the multicore-challenge ii. In: KELLER, R.; KRAMER, D.; WEISS, J.-P. (Ed.). Berlin, Heidelberg: Springer-Verlag, 2012. cap. Experiences with High-level Programming Directives for Porting Applications to GPUs, p. 96–107. ISBN 978-3-642-30396-8. Disponível em: <http://dl.acm.org/citation.cfm?id=2340646.2340659>.

[52] OpenMP Architecture Review Board. **OpenMP Application Program Interface, Version 4.0**. [S.l.], Julho 2013. Disponível em: <http://openmp.org/wp/openmp-specifications/>.

[53] OpenACC Architecture Review Board. **The OpenACC Application Programming Interface, Version 2.5**. [S.l.], Outubro 2015. Disponível em: <http://www.openacc.org/content/specifications-tech-reports>.

[54] MEDINA, D. S.; ST-CYR, A.; WARBURTON, T. OCCA: A unified approach to multi-threading languages. **arXiv preprint arXiv:1403.0968**, 2014.

[55] DOLBEAU, R.; BIHAN, S.; BODIN, F. HMPP: A hybrid multi-core parallel programming environment. In: **Workshop on general purpose processing on graphics processing units (GPGPU 2007)**. [S.l.: s.n.], 2007. v. 28.

[56] DING, W. et al. Using GPU shared memory with a directive-based approach. In: **Parallel Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International**. [S.l.: s.n.], 2014. p. 1021–1028.

[57] GHOSH, S. et al. Experiences with OpenMP, PGI, HMPP and OpenACC directives on ISO/TTI kernels. In: **High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:**. [S.l.: s.n.], 2012. p. 691–700.

PUC-Rio - Certificação Digital Nº 1421599/CA

[58] FEKI, S. et al. Porting an explicit time-domain volume-integral-equation solver on GPUs with OpenACC [open problems in cem]. **Antennas and Propagation Magazine, IEEE**, v. 56, n. 2, p. 265–277, April 2014. ISSN 1045-9243.

[59] WIENKE, S. et al. OpenACC—first experiences with real-world applications. In: **Euro-Par 2012 Parallel Processing**. [S.l.]: Springer, 2012. p. 859–870.

[60] SOUZA, P.; BORGES, L.; NEWBURN, C. J. Heterogeneous architecture library. In: **Second EAGE Workshop on High Performance Computing for Upstream**. [S.l.: s.n.], 2015.

[61] NEWBURN, C. et al. Heterogeneous streaming. In: **Sixth International Workshop on Accelerators and Hybrid Exascale**. [S.l.: s.n.], 2016.

[62] AUGONNET, C. et al. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. **Concurrency and Computation: Practice and Experience**, Wiley Online Library, v. 23, n. 2, p. 187–198, 2011.

[63] LUK, C.-K.; HONG, S.; KIM, H. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In: IEEE. **Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on**. [S.l.], 2009. p. 45–55.

[64] BOSILCA, G. et al. DAGuE: A generic distributed DAG engine for high performance computing. **Parallel Computing**, Elsevier, v. 38, n. 1, p. 37–51, 2012.

[65] CAO, C. et al. clMAGMA: High performance dense linear algebra with OpenCL. In: ACM. **Proceedings of the International Workshop on OpenCL 2013 & 2014**. [S.l.], 2014. p. 1.

[66] WANG, L. et al. Scaling scientific applications on clusters of hybrid multicore/GPU nodes. In: ACM. **Proceedings of the 8th ACM international conference on computing frontiers**. [S.l.], 2011. p. 6.

[67] CHEN, Y.; CUI, X.; MEI, H. Large-scale FFT on GPU clusters. In: ACM. **Proceedings of the 24th ACM International Conference on Supercomputing**. [S.l.], 2010. p. 315–324.

[68] PEÑA, A. J. et al. A complete and efficient CUDA-sharing solution for HPC clusters. **Parallel Computing**, Elsevier, v. 40, n. 10, p. 574–588, 2014.

PUC-Rio - Certificação Digital Nº 1421599/CA

[69] CALANDRA, H. et al. Evaluation of successive CPUs/APUs/GPUs based on an OpenCL finite difference stencil. In: **Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on**. [S.l.: s.n.], 2013. p. 405–409. ISSN 1066-6192.

[70] Message Passing Interface Forum. **MPI: A Message-Passing Interface Standard**. Knoxville, TN, USA, 1998. Disponível em: <http://www.mpi-forum.org/docs/mpi-1.1/mpi-11-html/mpi-report.html>.

[71] BERNASCHI, M.; BISSON, M.; ROSSETTI, D. Benchmarking of communication techniques for GPUs. **Journal of Parallel and Distributed Computing**, v. 73, n. 2, p. 250 – 255, 2013. ISSN 0743-7315. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0743731512002213>.

[72] MICIKEVICIUS, P. Multi-gpu programming. **GPU Computing Webinars, NVIDIA**, 2011.

[73] CUDA-Aware OpenMPI distribution. [S.l.]. Disponível em: <https://www.open-mpi.org/faq/?category=buildcuda>.

[74] CUDA-Aware MVAPICH2 distribution. [S.l.]. Disponível em: <http://mvapich.cse.ohio-state.edu/features/#mv2gdr>.

[75] ZHU, J.; LINES, L. R. Comparison of kirchhoff and reverse-time migration methods with applications to prestack depth imaging of complex structures. **GEOPHYSICS**, v. 63, n. 4, p. 1166–1176, 1998. Disponível em: <http://dx.doi.org/10.1190/1.1444416>.

[76] ZENG, C. et al. Broadband least-squares reverse time migration for complex structure imaging. **SEG 84th Annual International Meeting**, Society of Exploration Geophysicists, p. 3715–3719, 2014.

[77] VERMEER, G. J.; BEASLEY, C. J. **3-D seismic survey design**. [S.l.]: Society of Exploration Geophysicists Tulsa, 2002.

[78] TARANTOLA, A. Inversion of seismic reflection data in the acoustic approximation. **Geophysics**, Society of Exploration Geophysicists, v. 49, n. 8, p. 1259–1266, 1984.

[79] GAUTHIER, O.; VIRIEUX, J.; TARANTOLA, A. Two-dimensional nonlinear inversion of seismic waveforms: Numerical results. **Geophysics**, Society of Exploration Geophysicists, v. 51, n. 7, p. 1387–1403, 1986.

PUC-Rio - Certificação Digital Nº 1421599/CA

[80] MORA, P. Nonlinear two-dimensional elastic inversion of multioffset seismic data. **Geophysics**, Society of Exploration Geophysicists, v. 52, n. 9, p. 1211–1228, 1987.

[81] CRASE, E. et al. Robust elastic nonlinear waveform inversion: Application to real data. **Geophysics**, Society of Exploration Geophysicists, v. 55, n. 5, p. 527–538, 1990.

[82] KAPOOR, S. et al. Full waveform inversion around the world. In: **75th EAGE Conference & Exhibition incorporating SPE EUROPEC 2013**. [S.l.: s.n.], 2013.

[83] VIRIEUX, J.; OPERTO, S. An overview of full-waveform inversion in exploration geophysics. **GEOPHYSICS**, Society of Exploration Geophysicists, v. 74, n. 6, p. WCC1–WCC26, nov. 2009. Disponível em: <http://dx.doi.org/10.1190/1.3238367>.

[84] Nvidia. **Kepler Tuning Guide**. 2015. Disponível em: <http://docs.nvidia.com/cuda/kepler-tuning-guide/#shared-memory-bandwidth>.

[85] KENDALL, R. P. et al. **A proposed taxonomy for software development risks for high-performance computing (HPC) scientific/engineering applications**. [S.l.], 2007.

[86] VANTER, M. L. V. D.; POST, D.; ZOSEL, M. E. HPC needs a tool strategy. In: ACM. **Proceedings of the second international workshop on Software engineering for high performance computing system applications**. [S.l.], 2005. p. 55–59.

PUC-Rio - Certificação Digital Nº 1421599/CA

A
Documentação OpenVec

Para utilizar o OpenVec o desenvolvedor deve adicionar o *header file* do OpenVec conforme o código a seguir:

```
#include"openvec.h"
```

A.1
Como Compilar Código OpenVec

No momento da compilação o *header file* `openvec.h` detecta a arquitetura SIMD baseada nas flags do compilador. Para desligar a detecção automática basta adicionar a flag `-D_OV_NOAUTO`. A Tabela A.1 mostra como desligar a detecção automática e forçar a compilação para uma determinada arquitetura.

Flag OpenVec	Arquitetura	Compilador	Flag Arquitetura
-D_OV_NOAUTO -D_OV_MIC	Intel KNC KNF	Intel C/C++	-mmic
-D_OV_NOAUTO -D_OV_AVX	Intel AVX	Intel C/C++	-xAVX
-D_OV_NOAUTO -D_OV_AVX2	Intel AVX 2	Intel C/C++	-xCORE-AVX2
-D_OV_NOAUTO -D_OV_AVX512	Intel AVX 512	Intel C/C++	-xCORE-AVX512
-D_OV_NOAUTO -D_OV_AVX	Intel AVX	gcc/llvm	-mavx
-D_OV_NOAUTO -D_OV_AVX2	Intel AVX 2	gcc/llvm	-mavx2
-D_OV_NOAUTO -D_OV_SSE	Intel SSE	gcc/llvm	-msse
-D_OV_NOAUTO -D_OV_SSE4	Intel SSE4	gcc/llvm	-msse4
-D_OV_NOAUTO -D_OV_SSE	Intel SSE	Intel C/C++	-xSSE
-D_OV_NOAUTO -D_OV_SSE4	Intel SSE4	Intel C/C++	-xSSE4.1
-D_OV_NOAUTO -D_OV_NEON	ARM Neon	gcc/llvm	-mfpu=neon
-D_OV_NOAUTO -D_OV_NEON	ARM Neon 64 bit	gcc/llvm	
-D_OV_NOAUTO -D_OV_ALTIVEC	IBM Altivec	gcc	-maltivec
-D_OV_NOAUTO	Escalar	qualquer	qualquer

Tabela A.1: Flags de compilação para diferentes compiladores e arquiteturas.

Em todos os nossos testes a detecção automática funcionou, mas disponibilizamos uma maneira de forçar a compilação para uma determinada arquitetura.

O código do OpenVec escrito em C respeita o padrão C89, mas alguns exemplos fornecidos com o o OpenVec utilizam características do padrão C99. Para compilar estes códigos de exemplo é necessário adicionar a flag `-std=c99`.

A.2
Operadores

Em C++ o OpenVec faz um *overload* dos principais operadores. As operações matemáticas básicas com variáveis SIMD estão definidas, conforme o exemplo a seguir:

PUC-Rio - Certificação Digital Nº 1421599/CA

```
ov_float a,b,c,d; // SIMD

float e; // escalar

d = a*c - (a+b)/e + 2.0f*c;
```

No exemplo podemos notar que as operações entre vetores SIMD `"a*c"` e `"a+b"` estão definidas, bem como as operações entre vetores e escalares `"(a+b)/e"` e `"2.0f*c"`.

Em toda operação entre um vetor SIMD e um escalar, o escalar é promovido a vetor, criando um vetor temporário onde todos os elementos são iguais ao escalar, conforme o exemplo a seguir.

`2.0f*c;` // $2*c_0; 2*c_1; 2*c_2; ...; 2*c_n;$

A.3 Constantes

Definimos em tempo de compilação todas as constantes OpenVec para uma única arquitetura SIMD. As *flags* de compilação definem a arquitetura.

OV_COMPILER

Contem uma string com o nome do compilador, por exemplo `"gcc"`.

OV_DOUBLE_WIDTH

Contem a largura SIMD para o tipo `ov_double`, ou seja, variáveis do tipo `ov_double` possuem `OV_DOUBLE_WIDTH doubles`.

OV_DOUBLE_TAIL

Contem a largura SIMD menos um elemento, esta constante facilita o tratamento da cauda de vetores.

OV_FLOAT_WIDTH

Contem a largura SIMD para o tipo `ov_float`, ou seja, variáveis do tipo `ov_float` possuem `OV_FLOAT_WIDTH floats`.

OV_FLOAT_TAIL

Contem a largura SIMD menos um elemento, esta constante facilita o tratamento da cauda de vetores.

PUC-Rio - Certificação Digital Nº 1421599/CA

`OV_PLATFORM`

Contem uma string com o nome da plataforma SIMD, por exemplo `"Intel COMPILER AVX 2"`

`ov_restrict`

O OpenVec define a macro `ov_restrict` que funciona como o `restrict` do padrão C99. Esta macro pode ser utilizada tanto em C como em C++, definimos nossa macro para cada linguagem e compilador. Quando o compilador não suporta a definição *restrict* a macro não tem nenhum efeito.

O exemplo a seguir mostra o uso da macro `ov_restrict`:

```
void vsaxpy(int nv, float a, ov_float* ov_restrict x,
                             ov_float* ov_restrict y);
```

O desenvolvedor deve sempre que possível utilizar esta macro, pois ela habilita o compilador a assumir certas otimizações que podem gerar um código com maior desempenho.

PUC-Rio - Certificação Digital Nº 1421599/CA

A.4 Funções

As funções OpenVec atuam em variáveis SIMD do tipo `ov_float` e `ov_double`, a documentação das funções vetoriais mostra em detalhe a operação executada nos elementos da variável vetorial.

O exemplo a seguir mostra o cálculo simultâneo SIMD da raiz quadrada em todos os elementos da variável vetorial `"a"` utilizando a notação $a_{0..n}$ para representar seus elementos.

$$\sqrt{a_0}; \sqrt{a_1}; \sqrt{a_2}; ...; \sqrt{a_n};$$

Essa notação é utilizada durante toda a documentação das funções. Por exemplo, uma variável `"ov_float a"`, em uma arquitetura ARM Neon, possui os seguintes elementos: $\{a_0, a_1, a_2, a_3\}$, onde o limite $n = 3$ se refere a constante `OV_FLOAT_WIDTH-1`. Analogamente em uma variável `"ov_double a"`, n refere-se a constante `OV_DOUBLE_WIDTH-1`.

Algumas funções possuem várias interfaces fazendo um *overload* de argumentos, essas funções estão disponíveis apenas em C++ e são marcadas na documentação conforme o exemplo a seguir para a função `ceil`:

`ceil`$^{C++}$

ov_calloc

```
void *ov_calloc(size_t nmemb, size_t size)
```

Aloca uma área contígua de memória de tamanho `nmemb`×`size` bytes, zera todos os bytes e retorna o endereço alocado.

ov_malloc

```
void *ov_malloc(size_t size)
```

Aloca uma área contígua de memória de tamanho `size` bytes e retorna o endereço alocado.

ceil^{C++}

```
ov_double ceil(ov_double const a)
```

Retorna um vetor SIMD de `doubles` arredondando para o maior inteiro mais próximo de forma independente para todos os elementos de a.

$\lceil a_0 \rceil; \lceil a_1 \rceil; \lceil a_2 \rceil; ...; \lceil a_n \rceil;$

ceilf^{C++}

```
ov_float ceilf(ov_float const a)
```

Retorna um vetor SIMD de `floats` arredondando para o maior inteiro mais próximo de forma independente para todos os elementos de a.

$\lceil a_0 \rceil; \lceil a_1 \rceil; \lceil a_2 \rceil; ...; \lceil a_n \rceil;$

fabs^{C++}

```
ov_double fabs(ov_double const a)
```

Retorna um vetor SIMD de `doubles` com a operação módulo aplicada de forma independente em todos os elementos de a.

$|a_0|; |a_1|; |a_2|; ...; |a_n|;$

PUC-Rio - Certificação Digital Nº 1421599/CA

fabsf^{C++}

```
ov_float fabsf(ov_float const a)
```

Retorna um vetor SIMD de `floats` com a operação módulo aplicada de forma independente em todos os elementos de a.

$|a_0|; |a_1|; |a_2|; ...; |a_n|;$

floor^{C++}

```
ov_double floor(ov_double const a)
```

Retorna um vetor SIMD de `doubles` arredondando para o menor inteiro mais próximo de forma independente para todos os elementos de a.

$\lfloor a_0 \rfloor; \lfloor a_1 \rfloor; \lfloor a_2 \rfloor; ...; \lfloor a_n \rfloor;$

floorf^{C++}

```
ov_float floorf(ov_float const a)
```

Retorna um vetor SIMD de `floats` arredondando para o menor inteiro mais próximo de forma independente para todos os elementos de a.

$\lfloor a_0 \rfloor; \lfloor a_1 \rfloor; \lfloor a_2 \rfloor; ...; \lfloor a_n \rfloor;$

ov_absd

```
ov_double ov_absd(ov_double const a)
```

Retorna um vetor SIMD de `doubles` com a operação módulo aplicada de forma independente em todos os elementos de a.

$|a_0|; |a_1|; |a_2|; ...; |a_n|;$

ov_absf

```
ov_float ov_absf(ov_float const a)
```

Retorna um vetor SIMD de `floats` com a operação módulo aplicada de forma independente em todos os elementos de a.

PUC-Rio - Certificação Digital Nº 1421599/CA

$|a_0|; |a_1|; |a_2|; ...; |a_n|;$

ov_addd

```
ov_double ov_addd(ov_double const a, ov_double const b)
```

Retorna um vetor SIMD de `doubles` com a operação adição aplicada de forma independente elemento a elemento de a e b.

$a_0 + b_0; a_1 + b_1; a_2 + b_2; ...; a_n + b_n;$

ov_addf

```
ov_float ov_addf(ov_float const a, ov_float const b)
```

Retorna um vetor SIMD de `floats` com a operação adição aplicada de forma independente elemento a elemento de a e b.

$a_0 + b_0; a_1 + b_1; a_2 + b_2; ...; a_n + b_n;$

ov_all_ge_0d

```
ov_bool ov_all_ge_0d(ov_double a)
```

Retorna verdadeiro se todos os elementos de a são maiores ou iguais a zero.

$all(a_0 \geq 0, a_1 \geq 0, a_2 \geq 0, ..., a_n \geq 0)$

ov_all_ge_0f

```
ov_bool ov_all_ge_0f(ov_float a)
```

Retorna verdadeiro se todos os elementos de a são maiores ou iguais a zero.

$all(a_0 \geq 0, a_1 \geq 0, a_2 \geq 0, ..., a_n \geq 0)$

PUC-Rio - Certificação Digital Nº 1421599/CA

ov_all_lt_0d

```
ov_bool ov_all_lt_0d(ov_double a)
```

Retorna verdadeiro se todos os elementos de a são menores que zero.

$all(a_0 < 0, a_1 < 0, a_2 < 0, ..., a_n < 0)$

ov_all_lt_0f

```
ov_bool ov_all_lt_0f(ov_float a)
```

Retorna verdadeiro se todos os elementos de a são menores que zero.

$all(a_0 < 0, a_1 < 0, a_2 < 0, ..., a_n < 0)$

ov_all_maxd

```
double ov_all_maxd(ov_double const a)
```

Retorna um escalar `double` com o maior elemento de a.

$max(a_0, a_1, a_2, ..., a_n)$

ov_all_maxf

```
float ov_all_maxf(ov_float const a)
```

Retorna um escalar `float` com o maior elemento de a.

$max(a_0, a_1, a_2, ..., a_n)$

ov_all_mind

```
double ov_all_mind(ov_double const a)
```

Retorna um escalar `double` com o menor elemento de a.

$min(a_0, a_1, a_2, ..., a_n)$

PUC-Rio - Certificação Digital Nº 1421599/CA

ov_all_minf

```
float ov_all_minf(ov_float const a)
```

Retorna um escalar `float` com o menor elemento de a.

$min(a_0, a_1, a_2, ..., a_n)$

ov_all_prodd

```
double ov_all_prodd(ov_double const a)
```

Retorna um escalar `double` com o produto de todos os elementos de a.

$a_0 \times a_1 \times a_2 \times ... \times a_n$

ov_all_prodf

```
float ov_all_prodf(ov_float const a)
```

Retorna um escalar `float` com o produto de todos os elementos de a.

$a_0 \times a_1 \times a_2 \times ... \times a_n$

ov_all_sumd

```
double ov_all_sumd(ov_double const a)
```

Retorna um escalar `double` com a soma de todos os elementos de a.

$a_0 + a_1 + a_2 + ... + a_n$

ov_all_sumf

```
float ov_all_sumf(ov_float const a)
```

Retorna um escalar `float` com a soma de todos os elementos de a.

$a_0 + a_1 + a_2 + ... + a_n$

PUC-Rio - Certificação Digital Nº 1421599/CA

ov_alld

```
ov_bool ov_alld(ov_maskd const mask)
```

Retorna verdadeiro se todos os bits de `mask` estiverem ligados.

ov_allf

```
ov_bool ov_allf(ov_maskf const mask)
```

Retorna verdadeiro se todos os bits de `mask` estiverem ligados.

ov_any_ge_0d

```
ov_bool ov_any_ge_0d(a)
```

Retorna verdadeiro se ao menos um elemento de a é maior ou igual a zero.

$any(a_0 \geq 0; a_1 \geq 0; a_2 \geq 0; ...; a_n \geq 0)$

ov_any_ge_0f

```
ov_bool ov_any_ge_0f(a)
```

Retorna verdadeiro se ao menos um elemento de a é maior ou igual a zero.

$any(a_0 \geq 0; a_1 \geq 0; a_2 \geq 0; ...; a_n \geq 0)$

ov_any_lt_0d

```
ov_bool ov_any_lt_0d(a)
```

Retorna verdadeiro se ao menos um elemento de a é menor que zero.

$any(a_0 < 0; a_1 < 0; a_2 < 0; ...; a_n < 0)$

ov_any_lt_0f

```
ov_bool ov_any_lt_0f(a)
```

Retorna verdadeiro se ao menos um elemento de a é menor que zero.

$any(a_0 < 0; a_1 < 0; a_2 < 0; ...; a_n < 0)$

PUC-Rio - Certificação Digital Nº 1421599/CA

ov_anyd

```
ov_bool ov_anyd(ov_maskd const mask)
```

Retorna verdadeiro se pelo menos um bit de `mask` estiver ligado.

ov_anyf

```
ov_bool ov_anyf(ov_maskf const mask)
```

Retorna verdadeiro se pelo menos um bit de `mask` estiver ligado.

ov_ceild

```
ov_double ov_ceild(ov_double const a)
```

Retorna um vetor SIMD de `doubles` arredondando para o maior inteiro mais próximo de forma independente para todos os elementos de a.

$\lceil a_0 \rceil; \lceil a_1 \rceil; \lceil a_2 \rceil; ...; \lceil a_n \rceil;$

ov_ceilf

```
ov_float ov_ceilf(ov_float const a)
```

Retorna um vetor SIMD de `floats` arredondando para o maior inteiro mais próximo de forma independente para todos os elementos de a.

$\lceil a_0 \rceil; \lceil a_1 \rceil; \lceil a_2 \rceil; ...; \lceil a_n \rceil;$

ov_conditionald

```
ov_double ov_conditionald(ov_maskd const mask, ov_double
const a, ov_double const b)
```

Retorna um vetor SIMD de `doubles` mantendo os elementos de a onde a máscara é verdadeira e os elementos de b onde a máscara é falsa.

PUC-Rio - Certificação Digital Nº 1421599/CA

ov_conditionalf

```
ov_float ov_conditionalf(ov_maskf const mask, ov_float
const a, ov_float const b)
```

Retorna um vetor SIMD de `floats` mantendo os elementos de a onde a máscara é verdadeira e os elementos de b onde a máscara é falsa.

ov_divd

```
ov_double ov_divd(ov_double const a, ov_double const b)
```

Retorna um vetor SIMD de `doubles` com a operação divisão aplicada de forma independente elemento a elemento de a e b.

$\frac{a_0}{b_0}; \frac{a_1}{b_1}; \frac{a_2}{b_2}; ...; \frac{a_n}{b_n};$

ov_divf

```
ov_float ov_divf(ov_float const a, ov_float const b)
```

Retorna um vetor SIMD de `floats` com a operação divisão aplicada de forma independente elemento a elemento de a e b.

$\frac{a_0}{b_0}; \frac{a_1}{b_1}; \frac{a_2}{b_2}; ...; \frac{a_n}{b_n};$

ov_eqd

```
ov_maskd ov_eqd(ov_double const a, ov_double const b)
```

Retorna uma máscara de bits com a operação `==` aplicada de forma independente elemento a elemento de a e b.

$a_0 == b_0; a_1 == b_1; a_2 == b_2; ...; a_n == b_n;$

ov_eqf

```
ov_maskf ov_eqf(ov_float const a, ov_float const b)
```

Retorna uma máscara de bits com a operação `==` aplicada de forma independente elemento a elemento de a e b.

PUC-Rio - Certificação Digital Nº 1421599/CA

$a_0 == b_0; a_1 == b_1; a_2 == b_2; ...; a_n == b_n;$

ov_floord

```
ov_double ov_floord(ov_double const a)
```

Retorna um vetor SIMD de `doubles` arredondando para o menor inteiro mais próximo de forma independente para todos os elementos de a.

$\lfloor a_0 \rfloor; \lfloor a_1 \rfloor; \lfloor a_2 \rfloor; ...; \lfloor a_n \rfloor;$

ov_floorf

```
ov_float ov_floorf(ov_float const a)
```

Retorna um vetor SIMD de `floats` arredondando para o menor inteiro mais próximo de forma independente para todos os elementos de a.

$\lfloor a_0 \rfloor; \lfloor a_1 \rfloor; \lfloor a_2 \rfloor; ...; \lfloor a_n \rfloor;$

ov_free

```
void ov_free(void *ptr)
```

Desaloca uma memória alocada previamente com a função `ov_malloc` ou `ov_calloc`.

ov_ged

```
ov_maskd ov_ged(ov_double const a, ov_double const b)
```

Retorna uma máscara de bits com a operação $\geq$ aplicada de forma independente elemento a elemento de a e b.

$a_0 \geq b_0; a_1 \geq b_1; a_2 \geq b_2; ...; a_n \geq b_n;$

ov_gef

```
ov_maskf ov_gef(ov_float const a, ov_float const b)
```

PUC-Rio - Certificação Digital Nº 1421599/CA

Retorna uma máscara de bits com a operação $\geq$ aplicada de forma independente elemento a elemento de a e b.

$a_0 \geq b_0; a_1 \geq b_1; a_2 \geq b_2; ...; a_n \geq b_n;$

`ov_getzerod`

```
ov_double ov_getzerod()
```

Retorna um vetor SIMD de `doubles` com todos os valores zerados.

`ov_getzerof`

```
ov_float ov_getzerof()
```

Retorna um vetor SIMD de `floats` com todos os valores zerados.

PUC-Rio - Certificação Digital Nº 1421599/CA

`ov_gtd`

```
ov_maskd ov_gtd(ov_double const a, ov_double const b)
```

Retorna uma máscara de bits com a operação `>` aplicada de forma independente elemento a elemento de a e b.

$a_0 > b_0; a_1 > b_1; a_2 > b_2; ...; a_n > b_n;$

`ov_gtf`

```
ov_maskf ov_gtf(ov_float const a, ov_float const b)
```

Retorna uma máscara de bits com a operação `>` aplicada de forma independente elemento a elemento de a e b.

$a_0 > b_0; a_1 > b_1; a_2 > b_2; ...; a_n > b_n;$

`ov_ldd`

```
ov_double ov_ldd(double const *addr)
```

Retorna um vetor SIMD de `doubles` carregado a partir de um endereço **alinhado** em memória.

ov_ldf

```
ov_float ov_ldf(float const *addr)
```

Retorna um vetor SIMD de `floats` carregado a partir de um endereço **alinhado** em memória.

ov_led

```
ov_maskd ov_led(ov_double const a, ov_double const b)
```

Retorna uma máscara de bits com a operação $\leq$ aplicada de forma independente elemento a elemento de a e b.

$a_0 \leq b_0; a_1 \leq b_1; a_2 \leq b_2; ...; a_n \leq b_n;$

ov_lef

```
ov_maskf ov_lef(ov_float const a, ov_float const b)
```

Retorna uma máscara de bits com a operação $\leq$ aplicada de forma independente elemento a elemento de a e b.

$a_0 \leq b_0; a_1 \leq b_1; a_2 \leq b_2; ...; a_n \leq b_n;$

ov_loadd

```
ov_double ov_loadd(double const *addr)
```

Retorna um vetor SIMD de `doubles` carregado a partir de um endereço **alinhado** em memória.

ov_loadf

```
ov_float ov_loadf(float const *addr)
```

Retorna um vetor SIMD de `floats` carregado a partir de um endereço **alinhado** em memória.

PUC-Rio - Certificação Digital Nº 1421599/CA

ov_ltd

```
ov_maskd ov_ltd(ov_double const a, ov_double const b)
```

Retorna uma máscara de bits com a operação `<` aplicada de forma independente elemento a elemento de a e b.

$a_0 < b_0; a_1 < b_1; a_2 < b_2; ...; a_n < b_n;$

ov_ltf

```
ov_maskf ov_ltf(ov_float const a, ov_float const b)
```

Retorna uma máscara de bits com a operação `<` aplicada de forma independente elemento a elemento de a e b.

$a_0 < b_0; a_1 < b_1; a_2 < b_2; ...; a_n < b_n;$

PUC-Rio - Certificação Digital Nº 1421599/CA

ov_maddd

```
ov_double ov_maddd(ov_double const a, ov_double const b,
ov_double const c)
```

Retorna um vetor SIMD de `doubles` com a operação *multiply and add* aplicada de forma independente elemento a elemento de a, b e c.

$(a_0 * b_0) + c_0; (a_1 * b_1) + c_1; (a_2 * b_2) + c_2; ...; (a_n * b_n) + c_n;$

ov_maddf

```
ov_float ov_maddf(ov_float const a, ov_float const b, ov_-
float const c)
```

Retorna um vetor SIMD de `floats` com a operação *multiply and add* aplicada de forma independente elemento a elemento de a, b e c.

$(a_0 * b_0) + c_0; (a_1 * b_1) + c_1; (a_2 * b_2) + c_2; ...; (a_n * b_n) + c_n;$

ov_maxd^{C++}

```
ov_double ov_maxd(double const a, ov_double const b)
```

Retorna um vetor SIMD de `doubles` com a operação maior valor aplicada de forma independente elemento a elemento de b com o escalar a.

$max(a, b_0); max(a, b_1); max(a, b_2); ...; max(a, b_n);$

ov_maxd^{C++}

```
ov_double ov_maxd(ov_double const a, double const b)
```

Retorna um vetor SIMD de `doubles` com a operação maior valor aplicada de forma independente elemento a elemento de a com o escalar b.

$max(a_0, b); max(a_1, b); max(a_2, b); ...; max(a_n, b);$

ov_maxd

```
ov_double ov_maxd(ov_double const a, ov_double const b)
```

Retorna um vetor SIMD de `doubles` com a operação maior valor aplicada de forma independente elemento a elemento de a e b.

$max(a_0, b_0); max(a_1, b_1); max(a_2, b_2); ...; max(a_n, b_n);$

ov_maxf^{C++}

```
ov_float ov_maxf(float const a, ov_float const b)
```

Retorna um vetor SIMD de `floats` com a operação maior valor aplicada de forma independente elemento a elemento de b com o escalar a.

$max(a, b_0); max(a, b_1); max(a, b_2); ...; max(a, b_n);$

ov_maxf^{C++}

```
ov_float ov_maxf(ov_float const a, float const b)
```

Retorna um vetor SIMD de `floats` com a operação maior valor aplicada de forma independente elemento a elemento de a com o escalar b.

PUC-Rio - Certificação Digital Nº 1421599/CA

$max(a_0, b); max(a_1, b); max(a_2, b); ...; max(a_n, b);$

ov_maxf

```
ov_float ov_maxf(ov_float const a, ov_float const b)
```

Retorna um vetor SIMD de `floats` com a operação maior valor aplicada de forma independente elemento a elemento de a e b.

$max(a_0, b_0); max(a_1, b_1); max(a_2, b_2); ...; max(a_n, b_n);$

ov_mind^{C++}

```
ov_double ov_mind(double const a, ov_double const b)
```

Retorna um vetor SIMD de `doubles` com a operação menor valor aplicada de forma independente elemento a elemento de b com o escalar a.

$min(a, b_0); min(a, b_1); min(a, b_2); ...; min(a, b_n);$

ov_mind^{C++}

```
ov_double ov_mind(ov_double const a, double const b)
```

Retorna um vetor SIMD de `doubles` com a operação menor valor aplicada de forma independente elemento a elemento de a com o escalar b.

$min(a_0, b); min(a_1, b); min(a_2, b); ...; min(a_n, b);$

ov_mind

```
ov_double ov_mind(ov_double const a, ov_double const b)
```

Retorna um vetor SIMD de `doubles` com a operação menor valor aplicada de forma independente elemento a elemento de a e b.

$min(a_0, b_0); min(a_1, b_1); min(a_2, b_2); ...; min(a_n, b_n);$

PUC-Rio - Certificação Digital Nº 1421599/CA

ov_minf^{C++}

```
ov_float ov_minf(float const a, ov_float const b)
```

Retorna um vetor SIMD de `floats` com a operação menor valor aplicada de forma independente elemento a elemento de b com o escalar a.

$min(a, b_0); min(a, b_1); min(a, b_2); ...; min(a, b_n);$

ov_minf^{C++}

```
ov_float ov_minf(ov_float const a, float const b)
```

Retorna um vetor SIMD de `floats` com a operação menor valor aplicada de forma independente elemento a elemento de a com o escalar b.

$min(a_0, b); min(a_1, b); min(a_2, b); ...; min(a_n, b);$

ov_minf

```
ov_float ov_minf(ov_float const a, ov_float const b)
```

Retorna um vetor SIMD de `floats` com a operação menor valor aplicada de forma independente elemento a elemento de a e b.

$min(a_0, b_0); min(a_1, b_1); min(a_2, b_2); ...; min(a_n, b_n);$

ov_msubd

```
ov_double ov_msubd(ov_double const a, ov_double const b,
ov_double const c)
```

Retorna um vetor SIMD de `doubles` com a operação *multiply and subtract* aplicada de forma independente elemento a elemento de a, b e c.

$(a_0 * b_0) - c_0; (a_1 * b_1) - c_1; (a_2 * b_2) - c_2; ...; (a_n * b_n) - c_n;$

ov_msubf

```
ov_float ov_msubf(ov_float const a, ov_float const b, ov_-
float const c)
```

PUC-Rio - Certificação Digital Nº 1421599/CA

Retorna um vetor SIMD de `floats` com a operação *multiply and subtract* aplicada de forma independente elemento a elemento de a, b e c.

$(a_0 * b_0) - c_0; (a_1 * b_1) - c_1; (a_2 * b_2) - c_2; ...; (a_n * b_n) - c_n;$

ov_muld

```
ov_double ov_muld(ov_double const a, ov_double const b)
```

Retorna um vetor SIMD de `doubles` com a operação multiplicação aplicada de forma independente elemento a elemento de a e b.

$a_0 \times b_0; a_1 \times b_1; a_2 \times b_2; ...; a_n \times b_n;$

ov_mulf

```
ov_float ov_mulf(ov_float const a, ov_float const b)
```

Retorna um vetor SIMD de `floats` com a operação multiplicação aplicada de forma independente elemento a elemento de a e b.

$a_0 \times b_0; a_1 \times b_1; a_2 \times b_2; ...; a_n \times b_n;$

ov_ned

```
ov_maskd ov_ned(ov_double const a, ov_double const b)
```

Retorna uma máscara de bits com a operação `!=` aplicada de forma independente elemento a elemento de a e b.

$a_0 \neq b_0; a_1 \neq b_1; a_2 \neq b_2; ...; a_n \neq b_n;$

ov_nef

```
ov_maskf ov_nef(ov_float const a, ov_float const b)
```

Retorna uma máscara de bits com a operação `!=` aplicada de forma independente elemento a elemento de a e b.

$a_0 \neq b_0; a_1 \neq b_1; a_2 \neq b_2; ...; a_n \neq b_n;$

PUC-Rio - Certificação Digital Nº 1421599/CA

ov_not_alignedd

```
unsigned long ov_not_alignedd(double const *addr)
```

Retorna zero se `addr` está alinhado, caso contrário retorna o número de elementos (`double`) adicionais ao maior endereço alinhado que é menor ou igual à `addr`. Por exemplo, em uma máquina SIMD, se o array `double *x` está alinhado o retorno para `addr=&x[1]` é 1, e o retorno para `addr=&x[OV_DOUBLE_WIDTH]` é zero.

Em uma CPU escalar esta função sempre retorna 0.

Também fornecemos a função `ov_is_alignedd`, que implementamos da seguinte forma:

```
#define ov_is_alignedd(addr) (!ov_not_alignedd(addr))
```

ov_not_alignedf

```
unsigned long ov_not_alignedf(float const *addr)
```

Retorna zero se `addr` está alinhado, caso contrário retorna o número de elementos (`float`) adicionais ao maior endereço alinhado que é menor ou igual à `addr`. Por exemplo, em uma máquina SIMD, se o array `float *x` está alinhado o retorno para `addr=&x[1]` é 1, e o retorno para `addr=&x[OV_FLOAT_WIDTH]` é zero.

Em uma CPU escalar esta função sempre retorna 0.

Também fornecemos a função `ov_is_alignedf`, que implementamos da seguinte forma:

```
#define ov_is_alignedf(addr) (!ov_not_alignedf(addr))
```

ov_rcpd

```
ov_double ov_rcpd(ov_double const a)
```

Retorna um vetor SIMD de `doubles` com a operação *reciprocal* aplicada de forma independente em todos os elementos de a.

$\frac{1}{a_0}; \frac{1}{a_1}; \frac{1}{a_2}; \ldots; \frac{1}{a_n};$

PUC-Rio - Certificação Digital Nº 1421599/CA

ov_rcpf

```
ov_float ov_rcpf(ov_float const a)
```

Retorna um vetor SIMD de `floats` com a operação *reciprocal* aplicada de forma independente em todos os elementos de a.

$\frac{1}{a_0}; \frac{1}{a_1}; \frac{1}{a_2}; ...; \frac{1}{a_n};$

ov_rsqrtd

```
ov_double ov_rsqrtd(ov_double const a)
```

Retorna um vetor SIMD de `doubles` com a operação raiz quadrada inversa aplicada de forma independente em todos os elementos de a.

$\frac{1}{\sqrt{a_0}}; \frac{1}{\sqrt{a_1}}; \frac{1}{\sqrt{a_2}}; ...; \frac{1}{\sqrt{a_n}};$

ov_rsqrtf

```
ov_float ov_rsqrtf(ov_float const a)
```

Retorna um vetor SIMD de `floats` com a operação raiz quadrada inversa aplicada de forma independente em todos os elementos de a.

$\frac{1}{\sqrt{a_0}}; \frac{1}{\sqrt{a_1}}; \frac{1}{\sqrt{a_2}}; ...; \frac{1}{\sqrt{a_n}};$

ov_setd

```
ov_double ov_setd(double const a)
```

Retorna um vetor SIMD de `doubles` com o escalar a repetido em todas as posições do vetor.

$a; a; a; ...; a;$

ov_setf

```
ov_float ov_setf(float const a)
```

PUC-Rio - Certificação Digital Nº 1421599/CA

Retorna um vetor SIMD de `floats` com o escalar a repetido em todas as posições do vetor.

$a; a; a; ...; a;$

ov_sqrtd

```
ov_double ov_sqrtd(ov_double const a)
```

Retorna um vetor SIMD de `doubles` com a operação raiz quadrada aplicada de forma independente em todos os elementos de a.

$\sqrt{a_0}; \sqrt{a_1}; \sqrt{a_2}; ...; \sqrt{a_n};$

ov_sqrtf

```
ov_float ov_sqrtf(ov_float const a)
```

Retorna um vetor SIMD de `floats` com a operação raiz quadrada aplicada de forma independente em todos os elementos de a.

$\sqrt{a_0}; \sqrt{a_1}; \sqrt{a_2}; ...; \sqrt{a_n};$

ov_std

```
void ov_std(double *addr, ov_double const a)
```

Armazena um vetor SIMD de `doubles` em um endereço **alinhado** em memória.

ov_stf

```
void ov_stf(float *addr, ov_float const a)
```

Armazena um vetor SIMD de `floats` em um endereço **alinhado** em memória.

ov_stream_std

```
void ov_stream_std(double *addr, ov_double const a)
```

PUC-Rio - Certificação Digital Nº 1421599/CA

Armazena um vetor SIMD de `doubles` em um endereço **alinhado** em memória.

Em algumas CPUs esta instrução move uma linha de cache completa para a memória, e pode trazer mais desempenho evitando um `load/update/store`.

Em outras arquiteturas, que não possuem este tipo de *store*, esta função se comporta igual a função `ov_std`.

ov_stream_stf

```
void ov_stream_stf(float *addr, ov_float const a)
```

Armazena um vetor SIMD de `floats` em um endereço **alinhado** em memória.

Em algumas CPUs esta instrução move uma linha de cache completa para a memória, e pode trazer mais desempenho evitando um `load/update/store`.

Em outras arquiteturas, que não possuem este tipo de *store*, esta função se comporta igual a função `ov_stf`.

PUC-Rio - Certificação Digital Nº 1421599/CA

ov_subd

```
ov_double ov_subd(ov_double const a, ov_double const b)
```

Retorna um vetor SIMD de `doubles` com a operação subtração aplicada de forma independente elemento a elemento de a e b.

$a_0 - b_0; a_1 - b_1; a_2 - b_2; ...; a_n - b_n;$

ov_subf

```
ov_float ov_subf(ov_float const a, ov_float const b)
```

Retorna um vetor SIMD de `floats` com a operação subtração aplicada de forma independente elemento a elemento de a e b.

$a_0 - b_0; a_1 - b_1; a_2 - b_2; ...; a_n - b_n;$

ov_uldd

```
ov_double ov_uldd(double const *addr)
```

Retorna um vetor SIMD de `doubles` carregado a partir de um endereço **desalinhado** em memória.

ov_uldf

```
ov_float ov_uldf(float const *addr)
```

Retorna um vetor SIMD de `floats` carregado a partir de um endereço **desalinhado** em memória.

sqrtd^{C++}

```
ov_double sqrtd(ov_double const a)
```

Retorna um vetor SIMD de `doubles` com a operação raiz quadrada aplicada de forma independente em todos os elementos de a.

$\sqrt{a_0}; \sqrt{a_1}; \sqrt{a_2}; ...; \sqrt{a_n};$

sqrtf^{C++}

```
ov_float sqrtf(ov_float const a)
```

Retorna um vetor SIMD de `floats` com a operação raiz quadrada aplicada de forma independente em todos os elementos de a.

$\sqrt{a_0}; \sqrt{a_1}; \sqrt{a_2}; ...; \sqrt{a_n};$

PUC-Rio - Certificação Digital Nº 1421599/CA

B
Documentação HLIB

Para utilizar a HLIB basta fazer um `USE` do módulo Fortran 90 da HLIB, conforme o código a seguir:

```
USE HLIB
```

Como todas as sub-rotinas estão definidas no módulo HLIB, o compilador Fortran 90 será capaz de detectar se as invocações respeitam a interface durante a compilação.

B.1
Sub-rotinas

Todas as sub-rotinas da HLIB retornam um código de erro. Definimos o valor zero para sucesso e um valor diferente de zero para erro.

HLIB_AlocaMemDisp

```
SUBROUTINE HLIB_AlocaMemDisp(contexto, tam, zerar, memDisp,
   ierro)
                              ! Dispositivo
  TYPE(HLIB_contexto_t),      INTENT(IN)  :: contexto
                              ! Numero de elementos
  INTEGER(HLIB_KINDMEM),      INTENT(IN)  :: tam
                              ! .TRUE. para inicializar com zero
  LOGICAL,                    INTENT(IN)  :: zerar
                              ! Area alocada de memoria
  TYPE(HLIB_real_t),          INTENT(OUT) :: memDisp
                              ! Codigo de erro
  INTEGER,                    INTENT(OUT) :: ierro
```

Aloca uma memória com tamanho `tam` bytes no dispositivo associado ao `contexto`.

Se o argumento `zerar` é `.TRUE.` a memória é inicializada com zero.

Suportamos interfaces para os seguintes tipos do argumento `memDisp`: `HLIB_real_t`, `HLIB_double_t`, `HLIB_int_t` e `HLIB_long_t`.

PUC-Rio - Certificação Digital Nº 1421599/CA

HLIB_AlocaMemTransf1D

```
SUBROUTINE HLIB_AlocaMemTransf1D(contexto, lb, ub,
                                 array, ierro)
                         ! Dispositivo
  TYPE(HLIB_contexto_t), INTENT(IN) :: contexto
                         ! Lower bounds
  INTEGER,               INTENT(IN) :: lb(1)
                         ! Upper bounds
  INTEGER,               INTENT(IN) :: ub(1)
                         ! Ponteiro CPU
  REAL,                  POINTER :: array(:)
                         ! Codigo de erro
  INTEGER,               INTENT(OUT) :: ierro
```

Aloca uma memória da CPU otimizada para ser utilizada em transferências do tipo CPU ⇔ dispositivo. Em alguns *backends* esta memória não pode ser paginada (*pinned*), com isso, nestes *backends*, temos um ganho de desempenho nas transferências CPU ⇔ dispositivo.

O *lower bound* do ponteiro é definido pelo argumento `lb(1)` e o *upper bound* é definido pelo argumento `ub(1)`. Ou seja, a área é alocada com as seguintes dimensões `array(lb(1):ub(1))`.

O desenvolvedor deve somente desalocar esta memória com a sub-rotina `HLIB_DesalocaMemTransf`.

Suportamos interfaces para os seguintes tipos do argumento `array`: `REAL`, `DOUBLE PRECISION`, `INTEGER(KIND=HLIB_I32)` e `INTEGER(KIND=HLIB_I64)`.

HLIB_AlocaMemTransf2D

```
SUBROUTINE HLIB_AlocaMemTransf2D(contexto, lb, ub,
                                 array, ierro)
                         ! Dispositivo
  TYPE(HLIB_contexto_t), INTENT(IN) :: contexto
                         ! Lower bounds
  INTEGER,               INTENT(IN) :: lb(2)
                         ! Upper bounds
  INTEGER,               INTENT(IN) :: ub(2)
                         ! Ponteiro CPU
  REAL,                  POINTER :: array(:,:)
                         ! Codigo de erro
  INTEGER,               INTENT(OUT) :: ierro
```

Aloca uma memória da CPU otimizada para ser utilizada em transferências do tipo CPU ⇔ dispositivo. Em alguns *backends* esta memória não

PUC-Rio - Certificação Digital Nº 1421599/CA

pode ser paginada (*pinned*), com isso, nestes *backends*, temos um ganho de desempenho nas transferências CPU ⇔ dispositivo.

O *lower bound* do ponteiro é definido pelo argumento `lb(1:2)` e o *upper bound* é definido pelo argumento `ub(1:2)`. Ou seja, a área é alocada com as seguintes dimensões `array(lb(1):ub(1), lb(2):ub(2))`.

O desenvolvedor deve somente desalocar esta memória com a sub-rotina `HLIB_DesalocaMemTransf`.

Suportamos interfaces para os seguintes tipos do argumento `array`: `REAL`, `DOUBLE PRECISION`, `INTEGER(KIND=HLIB_I32)` e `INTEGER(KIND=HLIB_I64)`.

HLIB_AlocaMemTransf3D

```
SUBROUTINE HLIB_AlocaMemTransf3D(contexto, lb, ub,
                                  array, ierro)
                          ! Dispositivo
  TYPE(HLIB_contexto_t),  INTENT(IN) :: contexto
                          ! Lower bounds
  INTEGER,                INTENT(IN) :: lb(3)
                          ! Upper bounds
  INTEGER,                INTENT(IN) :: ub(3)
                          ! Ponteiro CPU
  REAL,                   POINTER :: array(:,:,:)
                          ! Codigo de erro
  INTEGER,                INTENT(OUT) :: ierro
```

Aloca uma memória da CPU otimizada para ser utilizada em transferências do tipo CPU ⇔ dispositivo. Em alguns *backends* esta memória não pode ser paginada (*pinned*), com isso, nestes *backends*, temos um ganho de desempenho nas transferências CPU ⇔ dispositivo.

O *lower bound* do ponteiro é definido pelo argumento `lb(1:3)` e o *upper bound* é definido pelo argumento `ub(1:3)`. Ou seja, a área é alocada com as seguintes dimensões `array(lb(1):ub(1), lb(2):ub(2), lb(3):ub(3))`.

O desenvolvedor deve somente desalocar esta memória com a sub-rotina `HLIB_DesalocaMemTransf`.

Suportamos interfaces para os seguintes tipos do argumento `array`: `REAL`, `DOUBLE PRECISION`, `INTEGER(KIND=HLIB_I32)` e `INTEGER(KIND=HLIB_I64)`.

HLIB_CopiaDispCPU

```
SUBROUTINE HLIB_CopiaDispCPU(contexto, memDisp, offset,
   arrayCPU, dir, ierro)
                          ! Dispositivo
```

PUC-Rio - Certificação Digital Nº 1421599/CA

```
TYPE(HLIB_contexto_t), INTENT(IN)    :: contexto
                       ! Memoria dispositivo
TYPE(HLIB_real_t),     INTENT(INOUT) :: memDisp
                       ! Offset em elementos
INTEGER(HLIB_KINDMEM), INTENT(IN)    :: offset
                       ! Memoria CPU
REAL,                  INTENT(INOUT) :: arrayCPU(:,:)
                       ! Direcao da copia
INTEGER,               INTENT(IN)    :: dir
                       ! Codigo de erro
INTEGER,               INTENT(OUT)   :: ierro
```

Faz a cópia entre uma memória alocada no dispositivo associado ao `contexto` e uma memória alocada na CPU. A memória da CPU deve ser alocada previamente com uma das rotinas `HLIB_AlocaMemTransf*`.

O desenvolvedor deve informar o `offset` (em elementos) para a memória do dispositivo, que é mapeada pelo argumento `memDisp`.

O tamanho da cópia é determinado pelos *lower* e *upper bounds* de `arrayCPU`. O desenvolvedor deve passar *lower* e *upper bounds* que definam uma área contígua de memória.

O argumento `dir` define a direção da cópia, que pode ser `HLIB_DESTINO_CPU` ou `HLIB_ORIGEM_CPU`.

Como armazenamos o tamanho da área alocada no dispositivo, antes de efetuar a cópia verificamos se ocorrerá uma invasão de memória no dispositivo.

Suportamos interfaces para os seguintes pares de argumentos `arrayCPU` e `memDisp` conforme a tabela a seguir.

arrayCPU	memDisp
REAL, DIMENSION(:)	HLIB_real_t
REAL, DIMENSION(:,:)	HLIB_real_t
REAL, DIMENSION(:,:,:)	HLIB_real_t
DOUBLE PRECISION, DIMENSION(:)	HLIB_double_t
DOUBLE PRECISION, DIMENSION(:,:)	HLIB_double_t
DOUBLE PRECISION, DIMENSION(:,:,:)	HLIB_double_t
INTEGER(KIND=HLIB_I32), DIMENSION(:)	HLIB_int_t
INTEGER(KIND=HLIB_I32), DIMENSION(:,:)	HLIB_int_t
INTEGER(KIND=HLIB_I32), DIMENSION(:,:,:)	HLIB_int_t
INTEGER(KIND=HLIB_I64), DIMENSION(:)	HLIB_long_t
INTEGER(KIND=HLIB_I64), DIMENSION(:,:)	HLIB_long_t
INTEGER(KIND=HLIB_I64), DIMENSION(:,:,:)	HLIB_long_t

PUC-Rio - Certificação Digital Nº 1421599/CA

HLIB_CopiaDispDisp

```
SUBROUTINE HLIB_CopiaDispDisp(contexto, origem, offOrig,
   destino, offDest, tam, ierro)
  TYPE(HLIB_contexto_t), INTENT(IN)    :: contexto
  TYPE(HLIB_double_t),   INTENT(IN)    :: origem
  INTEGER(HLIB_KINDMEM), INTENT(IN)    :: offOrig
  TYPE(HLIB_double_t) ,  INTENT(INOUT) :: destino
  INTEGER(HLIB_KINDMEM), INTENT(IN)    :: offDest
  INTEGER(HLIB_KINDMEM), INTENT(IN)    :: tam
  INTEGER,               INTENT(OUT)   :: ierro
```

Faz a cópia entre memórias alocadas no dispositivo associado ao `contexto`.

O desenvolvedor deve informar os *offsets* (em elementos) para a origem e destino. O argumento `tam` define a quantidade de elementos a serem copiados.

Como armazenamos o tamanho da área alocada no dispositivo, antes de efetuar a cópia verificamos se ocorrerá uma invasão de memória.

Suportamos interfaces para os seguintes tipos dos argumentos `origem` e `destino`: `HLIB_real_t`, `HLIB_double_t`, `HLIB_int_t` e `HLIB_long_t`. O argumento `origem` deve ser do mesmo tipo do argumento `destino`.

PUC-Rio - Certificação Digital Nº 1421599/CA

HLIB_CriaContexto

```
SUBROUTINE HLIB_CriaContexto(dispNum, contexto, ierro)
                         ! Numero do dispositivo
  INTEGER,               INTENT(IN)  :: dispNum
                         ! Contexto retornado
  TYPE(HLIB_contexto_t), INTENT(OUT) :: contexto
                         ! Codigo de erro
  INTEGER,               INTENT(OUT) :: ierro
```

Cria um contexto com o dispositivo `dispNum` mod `ndisp`. Onde `ndisp` é o total de dispositivos. O desenvolvedor deve informar no argumento `dispNum` o número do dispositivo a ser utilizado.

O processo de inicialização compreende em verificar quantos dispositivos (`ndisp`) existem na máquina e associar ao contexto o dispositivo `dispNun` mod `ndisp`, onde `dispNum` é o número do dispositivo passado para a sub-rotina `HLIB_CriaContexto`.

Em alguns backends como o OpenCL e hStreams, também é necessário criar um fila de execução para as operações internas da biblioteca. Em CUDA não é necessário a criação desta fila, pois existem chamadas síncronas que não necessitam de uma fila, um objeto `cudaStream` neste caso.

HLIB_CriaFila

```
SUBROUTINE HLIB_CriaFila(contexto, fila, ierro)
  TYPE(HLIB_contexto_t), INTENT(IN)  :: contexto
  TYPE(HLIB_fila_t),     INTENT(OUT) :: fila
  INTEGER,               INTENT(OUT) :: ierro
```

Cria uma fila de execução para enfileirar comandos para execução no dispositivo associado ao `contexto`.

HLIB_DesalocaMemDisp

```
SUBROUTINE HLIB_DesalocaMemDisp(contexto, memDisp, ierro)
                         ! Dispositivo associado
  TYPE(HLIB_contexto_t), INTENT(IN)    :: contexto
                         ! Memoria a ser desalocada
  TYPE(HLIB_real_t),     INTENT(INOUT) :: memDisp
                         ! Codigo de erro
  INTEGER,               INTENT(OUT)   :: ierro
```

Desaloca uma memória previamente alocada com a sub-rotina `HLIB_AlocaMemDisp` no dispositivo associado ao `contexto`.

Suportamos interfaces para os seguintes tipos do argumento `memDisp`: `HLIB_real_t`, `HLIB_double_t`, `HLIB_int_t` e `HLIB_long_t`.

HLIB_DesalocaMemTransf

```
SUBROUTINE HLIB_DesalocaMemTransf(contexto, array, ierro)
                         ! Dispositivo associado
  TYPE(HLIB_contexto_t), INTENT(IN)    :: contexto
                         ! Memoria a ser desalocada
  REAL, POINTER :: array(:,:)
                         ! Codigo de erro
  INTEGER,               INTENT(OUT)   :: ierro
```

Desaloca uma memória previamente alocada com uma das sub-rotinas `HLIB_AlocaMemTransf*` no dispositivo associado ao `contexto`.

Suportamos interfaces para os seguintes tipos do argumento `array`: `REAL`, `DOUBLE PRECISION`, `INTEGER(KIND=HLIB_I32)` e `INTEGER(KIND=HLIB_I64)`.

PUC-Rio - Certificação Digital Nº 1421599/CA

HLIB_DestroiContexto

```
SUBROUTINE HLIB_DestroiContexto(contexto, ierro)
  TYPE(HLIB_contexto_t), INTENT(INOUT) :: contexto
  INTEGER,               INTENT(OUT)   :: ierro
```

Destrói o contexto com o dispositivo. Após a destruição o objeto `contexto` não pode ser utilizado por outras sub-rotinas.

HLIB_DestroiFila

```
SUBROUTINE HLIB_DestroiFila(contexto, fila, ierro)
  TYPE(HLIB_contexto_t), INTENT(IN)  :: contexto
  TYPE(HLIB_fila_t),     INTENT(INOUT) :: fila
  INTEGER,               INTENT(OUT) :: ierro
```

Destrói a fila no dispositivo associado ao `contexto`.

HLIB_ObtemNumDisp

```
SUBROUTINE HLIB_ObtemNumDisp(numDisp, ierro)
                         ! Total de dispositivos nesta maquina
  INTEGER,               INTENT(OUT)  :: numDisp
                         ! Codigo de erro
  INTEGER,               INTENT(OUT) :: ierro
```

Retorna no argumento `numDisp` o número de dispositivos disponíveis na máquina.

HLIB_ObtemTamMemDisp

```
SUBROUTINE HLIB_ObtemTamMemDisp(contexto, tam, ierro)
                         ! Contexto com um dispositivo
  TYPE(HLIB_contexto_t), INTENT(IN)  :: contexto
                         ! Tamanho total da memoria em bytes
  INTEGER(HLIB_KINDMEM), INTENT(OUT) :: tam
                         ! Codigo de erro
  INTEGER,               INTENT(OUT) :: ierro
```

Retorna no argumento `tam` o total de bytes da memória do dispositivo associado ao `contexto`.

PUC-Rio - Certificação Digital Nº 1421599/CA

HLIB_SelecionaFila

```
SUBROUTINE HLIB_SelecionaFila(contexto, fila, ierro)
  TYPE(HLIB_contexto_t), INTENT(IN)  :: contexto
  TYPE(HLIB_fila_t),     INTENT(IN) :: fila
  INTEGER,               INTENT(OUT) :: ierro
```

Seleciona o objeto `fila` como sendo a fila a ser utilizada pelas sub-rotinas da HLIB. Todas chamadas HLIB subsequentes serão enfileiradas nessa fila.

HLIB_SelecionaFilaPadrao

```
SUBROUTINE HLIB_SelecionaFilaPadrao(contexto, ierro)
  TYPE(HLIB_contexto_t), INTENT(IN)  :: contexto
  INTEGER,               INTENT(OUT) :: ierro
```

Seleciona a fila síncrona padrão, com isso, todas as invocações de sub-rotinas da HLIB neste contexto passam a bloquear em relação a CPU.

Esse é o comportamento padrão inicial da HLIB.

HLIB_MPI_SendRecv

```
SUBROUTINE HLIB_MPI_SendRecv(contexto, sendBuf, sendCount,
                             sendOff, dest, sendTag, recvBuf,
                             recvCount, recvOff, source,
                             recvTag, comm, status, ierro)
                         ! Dispositivo
  TYPE(HLIB_contexto_t), INTENT(IN) :: contexto
                         ! Memoria de envio (dispositivo)
  TYPE(HLIB_real_t),     INTENT(IN) :: sendBuf
                         ! Numero de elementos
  INTEGER,               INTENT(IN) :: sendCount
                         ! Offset em elementos no sendBuf
  INTEGER(HLIB_KINDMEM), INTENT(IN) :: sendOff
                         ! Rank do destino
  INTEGER,               INTENT(IN) :: dest
                         ! Etiqueta send
  INTEGER,               INTENT(IN) :: sendTag
                         ! Memoria de recebimento (dispositivo)
  TYPE(HLIB_real_t),     INTENT(INOUT) :: recvBuf
                         ! Numero de elementos
  INTEGER,               INTENT(IN)  :: recvCount
                         ! Offset em elementos no recvBuf
  INTEGER(HLIB_KINDMEM), INTENT(IN)  :: recvOff
                         ! Rank de origem
```

PUC-Rio - Certificação Digital Nº 1421599/CA

```
  INTEGER,                INTENT(IN)  :: source
                          ! Etiqueta recv
  INTEGER,                INTENT(IN)  :: recvTag
                          ! Comunicador MPI
  INTEGER,                INTENT(IN)  :: comm
                          ! Estado MPI retornado
  INTEGER,                INTENT(OUT) :: status(MPI_STATUS_SIZE)
                          ! Codigo de erro
  INTEGER,                INTENT(OUT) :: ierro
```

Faz um `MPI_SendRecv` utilizando `buffers` alocados no dispositivo associado ao `contexto`.

HLIB_Sincroniza

```
SUBROUTINE HLIB_Sincroniza(contexto, ierro, fila)
  TYPE(HLIB_contexto_t),        INTENT(IN)    :: contexto
  INTEGER,                      INTENT(OUT)   :: ierro
  TYPE(HLIB_fila_t), OPTIONAL, INTENT(INOUT) :: fila
```

Se o argumento opcional `fila` estiver presente, esta função aguarda o término de todos os comandos enfileirados no objeto fila. Senão aguarda o término de todos os comandos deste processo enfileirados no dispositivo associado ao `contexto`.

Esta sub-rotina bloqueia a execução na CPU até o término dos comandos.

HLIB_TestaFila

```
SUBROUTINE HLIB_TestaFila(contexto, fila, vazia, ierro)
  TYPE(HLIB_contexto_t), INTENT(IN)    :: contexto
  TYPE(HLIB_fila_t),     INTENT(INOUT) :: fila
                         ! Indica fila vazia
  LOGICAL,               INTENT(OUT)   :: vazia
                         ! Codigo de erro
  INTEGER,               INTENT(OUT)   :: ierro
```

Testa se todos os comandos da fila terminaram, ou seja, testa se a fila está vazia. Esta rotina retorna o estado da fila no argumento `vazia`.

Esta sub-rotina não bloqueia a execução até o término dos comandos, ela apenas retorna o estado da fila.

PUC-Rio - Certificação Digital Nº 1421599/CA

HLIB_ZeraMemDisp

```
SUBROUTINE HLIB_ZeraMemDisp(contexto, memDisp, ierro)
                            ! Dispositivo
  TYPE(HLIB_contexto_t), INTENT(IN)    :: contexto
                            ! Memoria no dispositivo
  TYPE(HLIB_double_t),   INTENT(INOUT) :: memDisp
                            ! Codigo de erro
  INTEGER,               INTENT(OUT)   :: ierro
```

Preenche com zero (0x00) todos os bytes da memória `memDisp` alocada no dispositivo associado ao `contexto`.

Suportamos interfaces para os seguintes tipos do argumento `memDisp`: `HLIB_real_t`, `HLIB_double_t`, `HLIB_int_t` e `HLIB_long_t`.

PUC-Rio - Certificação Digital Nº 1421599/CA