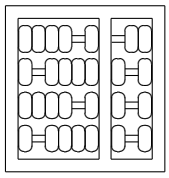


INSTITUTO DE COMPUTAÇÃO
UNIVERSIDADE ESTADUAL DE CAMPINAS

Um estudo do desempenho da multiplicação de matrizes em arquiteturas modernas

C. Benedicto *E. Borin* *P. R. B. Devloo*

Technical Report - IC-13-11 - Relatório Técnico

April - 2013 - Abril

Um estudo do desempenho da multiplicação de matrizes em arquituras modernas

Caian Benedicto* Edson Borin† Philippe Remy Bernard Devloo‡

Resumo

Nesse trabalho nós analisamos o impacto no desempenho causado por variações no algoritmo de multiplicação de matrizes densas e como esse impacto varia ao utilizarmos um compilador com estratégias diferentes de otimização. Comparamos o resultado obtido com as bibliotecas fornecidas por fabricantes de *hardware* para 2 sistemas distintos e também comparamos com com os resultados de desempenho obtidos utilizando-se a plataforma de programação paralela *OpenCL*.

*Instituto de Computação da Universidade Estadual de Campinas. Pesquisa desenvolvida com suporte financeiro do CNPq, processo 126785/2012-0

†Instituto de Computação da Universidade Estadual de Campinas

‡Faculdade de Engenharia Civil da Universidade Estadual de Campinas

1 Introdução

Com a constante evolução dos sistemas computacionais é importante avaliar o desempenho de rotinas comumente utilizadas na computação científica, traçando assim a evolução do impacto de diferentes otimizações e implementações no desempenho obtido. Nesse relatório nós avaliamos o desempenho de variações do algoritmo de multiplicação de matrizes densas em 2 sistemas de computação científica atuais bem como o desempenho de bibliotecas fornecidas pelos fabricantes dos processadores desses 2 sistemas. Avaliamos também o impacto de um compilador agressivo no desempenho das implementações e o desempenho obtido ao se introduzir uma nova plataforma de desenvolvimento, o *OpenCL*, focada em processamento paralelo para sistemas híbridos e que conta com gerenciamento de recursos implementado pelo fabricante de *hardware*.

2 Algoritmos de multiplicação

Esta seção descreve os algoritmos de multiplicação de matrizes densas utilizados durante o nosso estudo, esses algoritmos são uma simplificação do padrão estabelecido pela rotina GEMM ($C \leftarrow \alpha AB + \beta C$), assumindo escalares α e β iguais a 1 e dimensões quadradas. No pseudocódigo apresentado consideremos matrizes na representação *row-major*, mas a representação das matrizes pode variar durante a implementação por diversos motivos.

2.1 Ingênuo

O algoritmo 1 é a implementação ingênua da definição matemática de multiplicação de matrizes, onde cada elemento da matriz resultado é o produto escalar de uma linha do primeiro operando com uma coluna do segundo. Ao iterar sobre os elementos de colunas esse algoritmo deixa de explorar recursos presentes em arquiteturas modernas como localidade espacial por parte do subsistema de memoria e vetorização.

Algoritmo 1 Algoritmo ingênuo

```
para i = 1 → N faça
    para j = 1 → N faça
        para k = 1 → N faça
            C[i,j] = A[i,k] * B[k,j] + C[i,j]
            k = k + 1
        fim para
        j = j + 1
    fim para
    i = i + 1
fim para
```

2.2 Transposição

Para permitir uma melhor utilização da hierarquia de memória, as colunas da matriz B podem ser copiadas para um vetor de elementos consecutivos na memória antes da multiplicação da coluna pelas linhas. Esta otimização é implementada pelo algoritmo 2, de transposição, invertendo a ordem dos laços externos e copiando a coluna da matriz B em um vetor auxiliar antes da multiplicação pelas colunas. Essas 2 modificações também viabilizam a vetorização do código e permite que o programa faça um uso mais eficiente da *cache* do processador, já que os elementos do vetor temporário estão próximos na memória.

Algoritmo 2 Algoritmo de transposição

```
para j = 1 → N faça
    para k = 1 → N faça
        T[k] = B[k,j]
        k = k + 1
    fim para
    para i = 1 → N faça
        para k = 1 → N faça
            C[i,j] = A[i,k] * T[k] + C[i,j]
            k = k + 1
        fim para
        i = i + 1
    fim para
    j = j + 1
fim para
```

2.3 PZ

Outra forma de abordar o problema da localidade é a explorada pela biblioteca *PZ*, descrita no algoritmo 3 ela consiste em alterar a posição do laço mais interno ao invés dos externos, essa troca também proporciona uma iteração sobre elementos consecutivos do segundo operando e do resultado no laço mais interno, sob o ponto de vista desse laço, o acesso ao primeiro operando se dá em apenas um elemento.

2.4 *OpenCL*

O *OpenCL* é um padrão aberto mantido pelo *Khronos group* e voltado para o desenvolvimento paralelo em sistemas heterogêneos contendo *CPUs*, *GPUs* e outros aceleradores [3]. O modelo de execução proposto pelo *OpenCL* se baseia em dispositivos contendo diversas unidades computacionais (*compute unit*) capazes de gerenciar grupos de *threads* chamados *work-groups*. Esses grupos de *threads* executam uma mesma rotina, chamada de *Kernel*, sobre dados diferentes. Esse modelo de programação é compatível com implementações paralelas do algoritmo de blocagem, que melhora a localidade de dados do algoritmo ao dividir

Algoritmo 3 Algoritmo usado na PZ

```
para j = 1 → N faça
    para k = 1 → N faça
        para i = 1 → N faça
            C[i,j] += A[i,k] * B[k,j] + C[i,j]
            i = i + 1
        fim para
        k = k + 1
    fim para
    j = j + 1
fim para
```

a matriz principal em sub-matrizes [2]. Ao contrário de *CPUs* que contam com uma hierarquia multinível de *cache*, algumas *GPUs* possuem apenas uma *cache* de instruções e uma memória local de baixa latência compartilhada entre elementos do *work-group*, que deve ser explicitamente manipulada pelo algoritmo. O algoritmo 4 é uma implementação possível do algoritmo de blocagem em *OpenCL*, ele define que cada *thread* de um *work-group* atuará sobre um elemento de um bloco da matriz resultado [6].

Algoritmo 4 Algoritmo de blocagem em *OpenCL*

```
(i, j) = coordenadas da thread em relação ao bloco
(ig, jg) = coordenadas da thread em relação às matrizes principais
BA = matrix local b × b
BB = matrix local b × b
acc = 0
para k = 1 → N faça
    AA[i, j] = A[ig, j ∗ b]
    BA[i, j] = B[i ∗ b, jg]
    garantir: Sincronização do work-group
    para l = 1 → B faça
        acc = acc + BA[i, l] ∗ BB[l, j]
    fim para
    k = k + b
    garantir: Sincronização do work-group
fim para
C[i, j] = acc
```

3 Materiais e Métodos

Esta seção descreve os materiais e métodos utilizados neste estudo.

3.1 Materiais

Os seguintes sistemas foram utilizados para execução dos programas em nossos experimentos:

- MacPro/12: Sistema operacional Ubuntu 12.04.1 LTS (GNU/Linux 3.2.0-32-generic x86_x64), 2 processadores *Intel Xeon X5670* 2.93 *GHz* com 6 núcleos cada e tecnologia *Intel Hyper Threading* e 62.9*GB* de memória *RAM*.

- SGI/64: Sistema operacional Ubuntu 12.04.1 LTS (GNU/Linux 3.2.0-23-generic x86_x64), 4 processadores *AMD Opteron 6282SE* 2.6 *GHz* com 16 núcleos cada e 125.9*GB* de memória *RAM*.

- GPU/10: O sistema *MacPro-12* possui ainda duas *GPUs Radeon* modelo *HD* 5770 com 10 *compute units* e 1 *GB* de memória principal. Para os testes utilizamos apenas uma delas.

As seguintes ferramentas foram utilizadas para compilação dos programas:

- ICC: Compilador *Intel C++ Compiler*, versão 12.1.3. Os programas foram compilados com a opção de otimização `-O3`.

- GCC: Compilador *GNU Compiler Collection*, versão 4.6.3. Os programas foram compilados com a opção de otimização `-O3`.

- OpenCL-CPU: Pacote *Intel SDK for OpenCL Applications - 2012* no sistema *MacPro-12* e *AMD APP SDK* versão 2.8 no sistema *SGI-64*.

- OpenCL-GPU: Pacote *AMD APP SDK* versão 2.8 e driver de vídeo *AMD* versão 8.96.7 no sistema *MacPro-12*.

Além dos algoritmos implementados, descritos na Seção 2, utilizamos as rotinas de multiplicação de matrizes das seguintes bibliotecas:

- Intel MKL: versão 10.3.11 utilizando rotinas de multiplicação densa de matrizes `DGEMM` e `SGEMM`.

- AMD ACML: versão 5.1.0 utilizando rotinas de multiplicação densa de matrizes `dgemm` e `sgemm`.

- AMD APPML: versão 1.8.291 utilizando rotina de multiplicação densa de matrizes `clAmdBlasSgemm`.

- Biblioteca PZ: revisão 4190 da classe `TPZFMatrix` utilizando rotina de multiplicação `MultAdd`.

3.2 Métodos

A principal métrica para a avaliação do desempenho foi o número de operações de ponto flutuante realizadas por segundo (*Flops*) durante a multiplicação de matrizes quadradas de dimensão $N \times N$. O cálculo dos *GFlops** é baseado na definição matemática da multiplicação de matrizes. Como o cálculo de cada um dos N^2 elementos da matriz é dado por:

$$C_{ij} = \sum_{p=1}^{N} A_{ip} \times B_{pj},$$

cada elemento requer $2 \times N$ operações de ponto flutuante (N multiplicações e N adições). Assim sendo, o número total de operações de ponto flutuante é $2 \times N^3$. Supondo que t seja o tempo em segundos para a execução da multiplicação das matrizes, o desempenho é medido da seguinte forma:

$$GFlops(N, t) = \frac{2 \times N^3}{t} \times 10^{-9}$$

Para o teste de desempenho dos algoritmos, o procedimento consistiu na multiplicação sequencial das matrizes, da dimensão 64 até a dimensão 4928, com um incremento de 256. Esse procedimento foi repetido de 10 a 1000 vezes, sem o término do programa, a fim de permitir o cálculo da média e desvio estatístico do resultado e, consequentemente, de intervalos de confiança. A coleta de tempo dos algoritmos executados na *CPU* foi realizada pela rotina *clock_gettime* da biblioteca *time*, utilizando o contador `CLOCK_MONOTONIC`. Já na plataforma *OpenCL*, seja a execução na *CPU* ou *GPU*, foi possível utilizar contadores de *hardware* disponibilizados pela *API* do *OpenCL*.

4 Resultados experimentais para o sistema *MacPro-12*

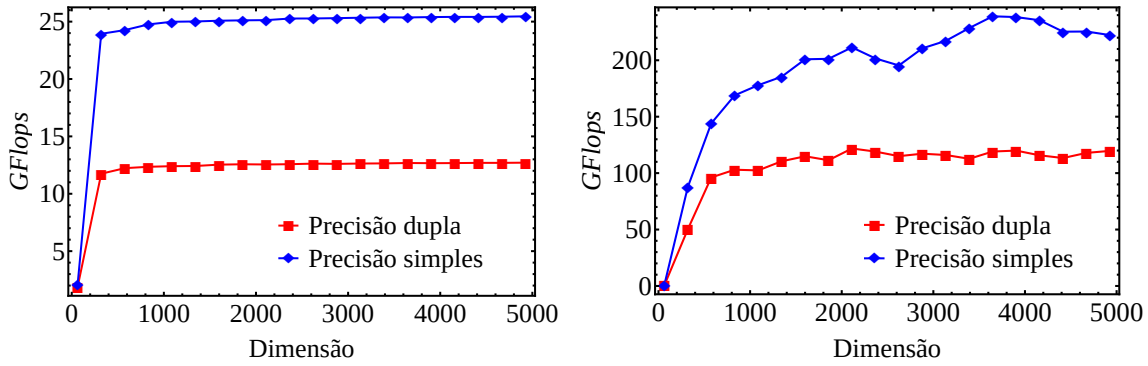
4.1 MKL

A princípio foram feitos testes de desempenho no sistema *MacPro-12* utilizando a biblioteca *MKL* e a partir dela pudemos estabelecer um patamar de desempenho para o sistema. A biblioteca *MKL* é uma biblioteca proprietária, comercializada pela *Intel*, e desenvolvida por especialistas capazes de gerar código altamente eficiente para os processadores da *Intel*. Os testes foram executados tanto em precisão simples (*float*) como precisão dupla (*double*) e o número de *threads* utilizado foi limitado com a variável de ambiente `MKL_NUM_THREADS`. Contudo, o uso efetivo das *threads* é determinado pela biblioteca em si, é possível modificar esse comportamento com a variável `MKL_DYNAMIC`, porém foi observada uma perda de desempenho de até 40% com essa modificação. Para os procedimentos de teste adotados, a biblioteca utilizou no máximo 12 *threads*, que é, por padrão, o número de núcleos físicos [5].

Como visto na Figura 1, executando com apenas uma *thread* a rotina de multiplicação da biblioteca *MKL* atingiu, para precisão simples e precisão dupla, respectivamente, um máximo de 25.4 e 12.6 *GFlops* a partir da dimensão 1088, se mantendo estáveis a partir

*1 *GFlop* equivale a 1 bilhão de *Flops*.

desse ponto. Os intervalos de confiança também se mantiveram estáveis, em ambas as precisões, variando entorno de 0.1 a 0.25 *GFlops*. Utilizando o máximo de 12 *threads*, em precisão dupla, a *MKL* apresentou um desempenho estável em torno de 119 *GFlops* a partir da dimensão 2112. Contudo, em precisão simples, ela não pareceu chegar a um ponto estável, ela apresentou um desempenho crescente até 225 *GFlops*, que ocorreu na dimensão 3648 e, a partir desse ponto, apresentou uma pequena queda no desempenho até os últimos pontos coletados. Os intervalos de confiança para as 2 precisões foram relativamente maiores, ficando em torno de 35 *GFlops* para precisão simples e 22 *GFlops* para precisão dupla. Em termos de escalabilidade a *MKL* teve um desempenho médio de 19 *GFlops* por *thread* em precisão simples e 9.8 *GFlops* por *thread* em precisão dupla, o que equivale, respectivamente, a 75% e 78% de eficiência obtida com a execução com uma única *thread*.



(a) 1 *thread* (b) 12 *threads*

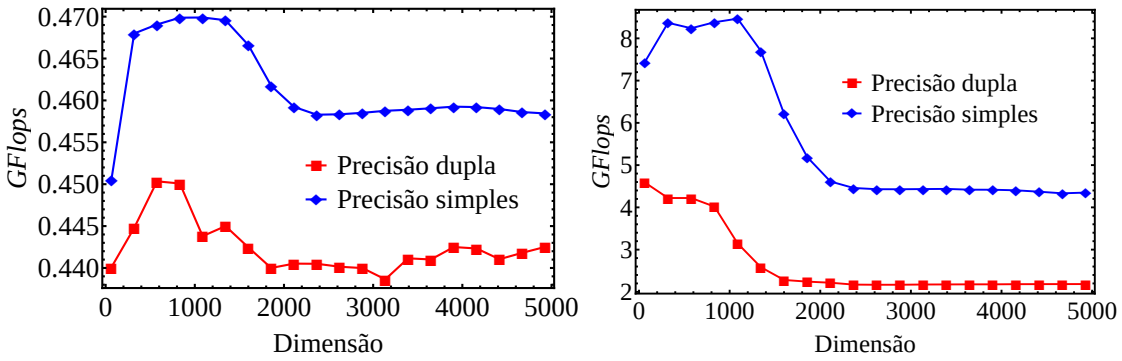
Figura 1: Desempenho da biblioteca *MKL* no sistema *MacPro-12* com 1 e 12 *threads*.

4.2 Transposição

Tendo estabelecido um patamar, decidimos verificar o algoritmo de transposição, baseado na transposição das colunas da segunda matriz. A implementação do algoritmo 2, de transposição, foi primeiramente compilado com o compilador *GCC* e, como podemos observar na Figura 2(a), o desempenho de pico da precisão simples foi de apenas 0.47 *GFlops* na dimensão 1088, diminuindo até chegar ao valor estável de 0.46 *GFlops* a partir da dimensão 2112, com o intervalo de confiança permanecendo na média de 0.006 *GFlops*. O desempenho com elementos de precisão dupla variou entre 0.44 e 0.45 *GFlops*, com um intervalo de confiança na ordem de 0.02 *GFlops*.

Notando a discrepância entre o desempenho dessa implementação contra a *MKL* com apenas uma *thread* realizamos uma inspeção manual do código de máquina gerado para o algoritmo de transposição e observamos que o código não utilizava as instruções vetoriais presentes no processador da *Intel* [4]. De fato, por padrão, mesmo com as *flags* de otimização de nível 3 (-O3) habilitadas, o *GCC* não realiza a otimização de vetorização do código.

Decidimos utilizar então o compilador desenvolvido pela *Intel*, o *ICC*, como compilador agressivo na tentativa de melhorar o desempenho da implementação. Como podemos observar na Figura 2(b), foi possível obter até 17.9 vezes mais desempenho de pico em precisão



(a) Compilador *GCC* (b) Compilador *ICC*

Figura 2: Desempenho do algoritmo de transposição no sistema *MacPro-12* compilado com o compilador *GCC* e com o compilador *ICC*.

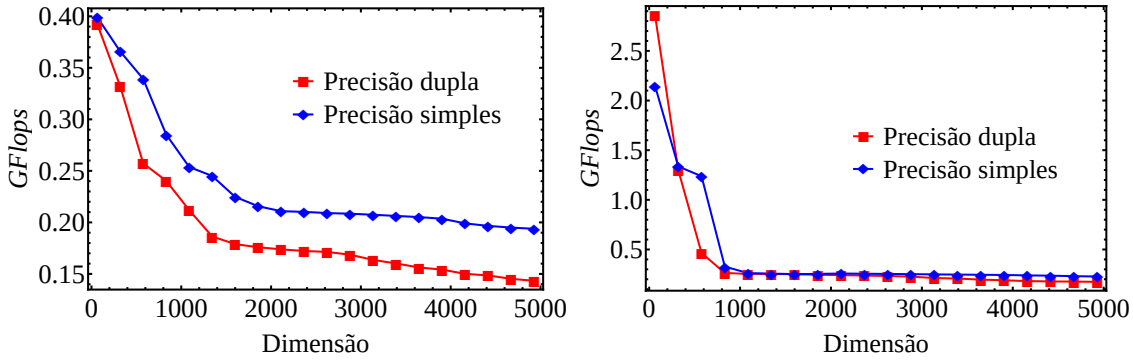
simples e até 9.3 vezes em precisão dupla. A curva de desempenho seguiu o mesmo padrão do teste anterior, com um pico de desempenho, seguido de uma região estável, em precisão simples o pico se deu na dimensão 1088 com 8.4 *GFlops*, o desempenho se torna estável na dimensão 2368 com 4.4 *GFlops* e em precisão dupla o pico se deu na dimensão 576, com 4.2 *GFlops* e com uma região estável a partir da dimensão 2112, com 2.1 *GFlops*. Em ambos os casos o intervalo de confiança se manteve entre os 0.03 *GFlops*. Analisando o ganho de desempenho obtido para os valores estáveis, temos um ganho de desempenho de 9.6 vezes para precisão simples e 4.8 vezes para precisão dupla.

Analisando novamente o código gerado, dessa vez pelo *ICC*, encontramos várias instruções *SSE* para multiplicação paralela de elementos, porém o desempenho, além de muito inferior ao da *MKL*, não se mantém constante no pico, esse último fato provavelmente seria explicado analisando-se o subsistema de memória. Verificamos então, com o auxílio da ferramenta *perf*, que a taxa de falta na *cache* de último nível foi próximo a 3% em torno da dimensão de pico e próximo de 90% nas dimensões de desempenho estável.

4.3 Implementação ingênua

Ao contrário da rotina da *MKL* e do algoritmo de transposição, a implementação ingênua da multiplicação de matriz, diretamente a partir do modelo matemático, não possui qualquer preocupação com a vetorização nem com a hierarquia de *cache*. Como consequência o desempenho do algoritmo decresce com o aumento da dimensão, variando de 0.33 até 0.23 *GFlops* em precisão simples e de 0.26 até 0.17 *GFlops* em precisão dupla, com picos iniciais de 0.4 *GFlops*, como indicado na Figura 3, e com intervalos de confiança de, respectivamente, 0.005 e 0.001 *GFlops* para o *GCC*. Embora a compilação utilizando o *ICC* tenha mostrado uma melhora significativa no desempenho de pico, em 4.9 vezes para precisão simples e 7.2 vezes para precisão dupla, para valores estáveis não houve nenhum ganho de desempenho.

Uma análise dos códigos de máquina revelou que mesmo o *ICC* não foi capaz de vetorizar o código, embora o tenha deixado mais compacto. O padrão de acesso não consecutivo dos elementos da segunda matriz é um dos principais problemas que inibem a vetorização



(a) Compilador *GCC* (b) Compilador *ICC*

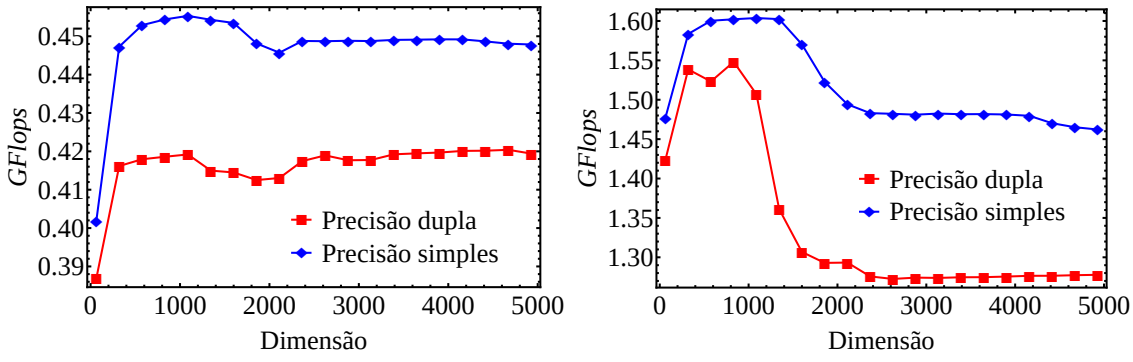
Figura 3: Desempenho do algoritmo ingênuo no sistema *MacPro-12* compilado com o compilador *GCC* e com o compilador *ICC*.

automática de código por parte dos compiladores. A taxa de falta na *cache* também se manteve alta para as regiões estáveis, tanto para o último nível de *cache* (L3) quanto para o nível L2.

4.4 PZ

Com as margens de desempenho para as implementações em *CPU* estabelecidas, testamos o desempenho da rotina de multiplicação de matrizes da biblioteca *PZ*.

Os resultados da Figura 4(a) indicam que o desempenho do código gerado pelo *GCC* se estabilizou em 0.45 *GFlops* em precisão simples e 0.42 *GFlops* em precisão dupla. Os desempenhos apresentaram ainda uma pequena queda de aproximadamente 0.01 *GFlops* em torno da dimensão 2112, sendo possível notar, em precisão dupla, que a queda se inicia a partir de 1344. Os intervalos de confiança foram, respectivamente, 0.003 e 0.03 *GFlops*.



(a) Compilador *GCC* (b) Compilador *ICC*

Figura 4: Desempenho do algoritmo da biblioteca *PZ* no sistema *MacPro-12* compilado com o compilador *GCC* e com o compilador *ICC*.

Supondo que o algoritmo estivesse sofrendo do mesmo problema de vetorização do algoritmo de referencia, o código também foi compilado com o compilador *ICC*. Como indicado na Figura 4(b), obtivemos, para precisão simples um pico de desempenho na dimensão 1088 de 1.60 *GFlops* que decresceu até a dimensão 2368, estabilizando em 1.48 *GFlops*. Notamos também que o desempenho voltou a cair a partir da dimensão 4160, chegando a 1.462 *GFlops*, correspondente a ultima dimensão testada. Para precisão dupla tivemos um pico de 1.55 *GFlops* na dimensão 832, estabilizando na dimensão 2368 com 1.27 *GFlops*. Foi obtido um intervalo de confiança de 0.004 *GFlops* para precisão simples e 0.01 *GFlops* para precisão dupla. Isso equivale a um ganho de desempenho de apenas 3 vezes para a região estável.

Uma inspeção do código fonte da rotina de multiplicação revelou a presença de declarações condicionais dentro dos laços da multiplicação, o que impossibilita uma boa otimização por parte do *ICC*, foi criada então uma versão simplificada da rotina, preservando sua estrutura fundamental, mas removendo-se as declarações condicionais. Essa nova rotina foi mais bem otimizada pelo *ICC*, como indicado na Figura 5, atingindo um desempenho de pico, em precisão simples, de 6.85 *GFlops* na dimensão 1088 e 4.14 *GFlops* na dimensão 2624, e gradualmente perdendo desempenho até atingir 3.92 *GFlops* para a última dimensão medida. Em precisão dupla o algoritmo apresentou um pico de 3.39 *GFlops* na dimensão 832 e estabilizou em 1.97 *GFlops* a partir da dimensão 2368. Em ambos os casos o intervalo de confiança esteve na ordem de 0.05 *GFlops*, com alguns picos esporádicos em algumas dimensões entre 1088 e 2112. Dessa vez o ganho de desempenho para a região estável, em relação ao código original compilado com o *GCC*, foi de 8.7 vezes para precisão simples e 4.7 vezes para precisão dupla.

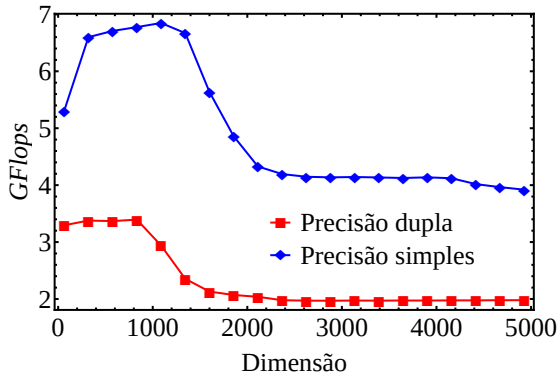


Figura 5: Desempenho da biblioteca *PZ* modificada no sistema *MacPro-12* compilada com o *ICC*.

A Tabela 1 resume os resultados de desempenho dos algoritmos de multiplicação de matrizes no sistema *MacPro-12*. A tabela apresenta para cada algoritmo o desempenho em *GFlops* da multiplicação de elementos de precisão simples e de elementos de precisão dupla. A coluna "Pico" indica o melhor resultado obtido com os diversos tamanhos de matrizes e a coluna "Moda" indica o resultado mais frequente nos experimentos. Além disso, os algoritmos estão ordenados de acordo com a moda do desempenho.

Algoritmo/Biblioteca	Desempenho em *GFlops*			
	Precisão Simples		Precisão Dupla	
	Pico	Moda	Pico	Moda
MKL (12 *threads*)	225.00	225.00 [a]	119.00	119.00
MKL (1 *thread*)	25.40	25.40	12.60	12.60
Transposição (*ICC*)	8.40	4.40	4.20	2.10
PZ Simples (*ICC*)	6.85	3.92 [b]	3.39	1.97
PZ (*ICC*)	1.60	1.48	1.55	1.27
Transposição (*GCC*)	0.47	0.46	0.45	0.44
PZ (*GCC*)	0.46	0.45	0.42	0.44
Ingênuo (*ICC*)	2.15	0.23 [b]	2.86	0.17
Ingênuo (*GCC*)	0.40	0.23 [b]	0.39	0.17

[a] A execução não atingiu um máximo estável dentro do intervalo testado
[b] A execução não atingiu um mínimo estável dentro do intervalo testado

Tabela 1: Tabela comparativa de desempenho dos algoritmos no sistema *MacPro-12*

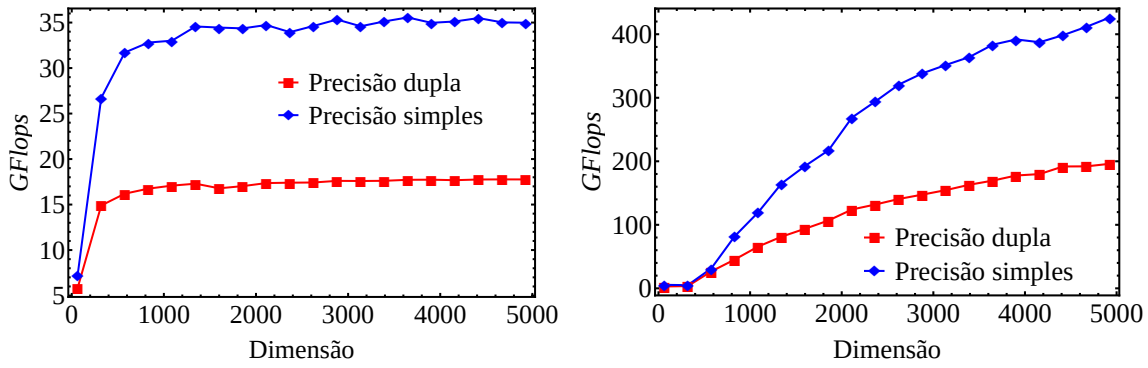
Os resultados obtidos com o sistema *MacPro-12* indicam que o desempenho de um algoritmo de multiplicação de matrizes ingênuo (0.23 *GFlops*), que não leva em consideração a microarquitetura do computador, pode ser mais do que 100 vezes pior do que o desempenho de um algoritmo desenvolvido por especialistas para a mesma máquina (25 *GFlops*). Também observamos que, otimizações simples, como a cópia da coluna a ser multiplicada, aliadas a um compilador otimizante capaz de vetorizar automaticamente o código, podem melhorar significativamente o desempenho do algoritmo. Mais especificamente, a multiplicação de números com precisão simples fica quase 20 vezes mais rápida (0.23 *GFlops* vs. 4.4 *GFlops*) e a multiplicação de números com precisão dupla fica mais de 12 vezes mais rápida (0.17 *GFlops* vs. 2.1 *GFlops*). Durante nossos experimentos identificamos estruturas condicionais na rotina de multiplicação da biblioteca PZ que inibia a vetorização automática com o compilador *ICC*. A remoção desta estrutura permitiu a vetorização do código e o desempenho aumentou em aproximadamente 3 vezes, se aproximando do desempenho do algoritmo de transposição vetorizado. Por fim, observamos que, mesmo após a otimização de cópia da coluna e da vetorização automática com o compilador *ICC*, o desempenho da multiplicação de matrizes dos códigos compilados ainda ficou em torno de 5 e 6 vezes aquém do desempenho da rotina de multiplicação de matrizes da *MKL*.

5 Resultados experimentais para o sistema *SGI-64*

Tendo disponível também um sistema com processadores *multi-core AMD*, foi possível testar o desempenho das bibliotecas e algoritmos em outra configuração de sistema.

5.1 ACML

Assim como a *MKL*, a biblioteca *ACML* é o equivalente da *AMD* para computação de alto desempenho, otimizada para seus processadores. Também é possível controlar o número de *threads* a serem executadas pela biblioteca, agora com a variável de ambiente `OMP_NUM_THREADS`. Os procedimentos de teste novamente foram executados para uma *thread* e todas as *threads* disponíveis, em precisão simples e precisão dupla. Como ilustrado na Figura 6, a *ACML* apresentou, para uma *thread*, um desempenho constante de 35.03 e 17.75 *GFlops* para precisão simples e precisão dupla, respectivamente, com intervalos de confiança de 0.04 *GFlops* para ambos os casos. Contudo, para 64 *threads*, não é possível determinar um ponto máximo para o desempenho da biblioteca, sendo que o maior desempenho notado foi de 426 *GFlops* em precisão simples e 196 *GFlops* em precisão dupla, o intervalo de confiança decresceu em ambos os casos com o aumento da dimensão, chegando a 20 *GFlops* em precisão simples e 11 *GFlops* em precisão dupla para as últimas dimensões medidas. Com isso podemos estabelecer um desempenho aproximado de 6.67 *GFlops* por *thread* para precisão simples e 3.06 *GFlops* por *thread* para precisão dupla. Comparando com o desempenho da biblioteca com apenas uma *thread*, o que equivale a aproximadamente 17% de eficiência de cada *thread*.



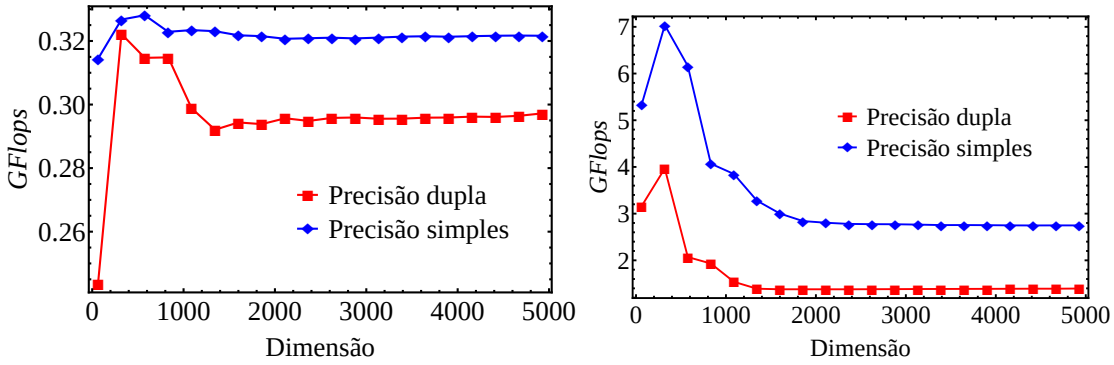
(a) 1 *thread* (b) 64 *threads*

Figura 6: Desempenho da biblioteca *ACML* no sistema *SGI-64* com 1 *thread* e com 64 *threads*.

5.2 Transposição

A curva de desempenho do algoritmo de transposição compilado com o *GCC* no sistema *SGI-64* manteve o mesmo padrão observado no *MacPro-12*, com um pico seguido de uma região estável. O pico observado foi de 0.328 *GFlops* em precisão simples na dimensão 576 e 0.322 *GFlops* em precisão dupla na dimensão 320, o desempenho nos pontos estáveis foi de, respectivamente, 0.321 e 0.116 *GFlops*, como indicado na Figura 7(a). Os intervalos de confiança foram bem pequenos, em torno de 2×10^{-5} *GFlops*.

Utilizando o compilador *ICC* obtivemos picos de desempenho de 7.036 e 3.967 *GFlops* em precisão simples e precisão dupla, respectivamente, na dimensão 320, como indicado na



(a) Compilador *GCC* (b) Compilador *ICC*

Figura 7: Desempenho do algoritmo de transposição no sistema *SGI-64* compilado com o compilador *GCC* e com o compilador *ICC*.

figura Figura 7(b). O desempenho estável para essa implementação foi de, respectivamente, 2.752 e 1.387 *GFlops*, o que equivale a um ganho de desempenho de mais de 9 vezes para a precisão simples e mais de 4 vezes para a precisão dupla. Os intervalos de confiança obtidos foram de 0.005 *GFlops* em precisão simples e 0.0003 *GFlops* em precisão dupla. Foi possível observar, dentro dos intervalos de confiança, a tendência do decrescimento do desempenho da implementação em precisão simples.

5.3 Implementação ingênua

A implementação ingênua compilada com o *GCC* no sistema *SGI-64* também seguiu o padrão de curva de desempenho apresentado no *MacPro-12*, com um pico inicial de 0.256 *GFlops* em precisão simples e 0.226 *GFlops* em precisão dupla e com uma queda abrupta até 0.102 *GFlops* em ambos os casos. Esse valor também decresceu com o aumento da dimensão até atingir, para precisão simples, 0.098 e 0.082 *GFlops*. O intervalo de confiança foi bem estreito, da ordem de 10^{-6} *GFlops*.

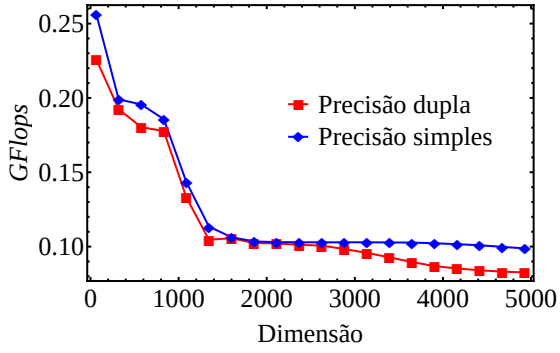


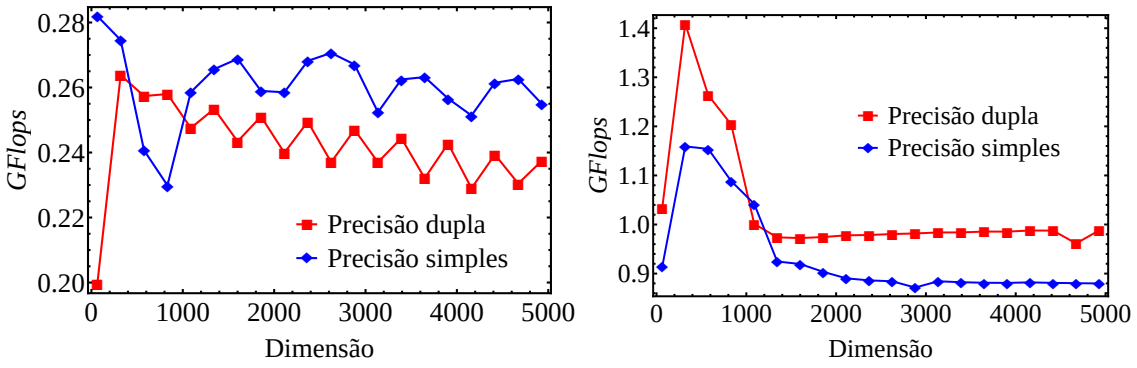
Figura 8: Desempenho do algoritmo ingênuo no sistema *SGI-64*.

Para esse caso em especial não achamos necessário o teste de desempenho com o com-

pilador *ICC* por não se tratar de uma implementação que faça uso de algum recurso da microarquitetura que possa ser analisado com um compilador agressivo.

5.4 PZ

Ao contrário das demais implementações, o desempenho da *PZ* compilada com o *GCC* no sistema *SGI-64* não se assimilou ao de nenhuma outra execução, a Figura 9(a) mostra uma oscilação no desempenho com o aumento da dimensão da matriz. Em precisão simples notou-se um desempenho que oscila em torno de 0.26 *GFlops* com um intervalo de 0.01 *GFlops*. Para precisão dupla esse intervalo também se manteve próximo aos 0.01 *GFlops*, mas com uma média claramente decrescente, de 0.25 a 0.24 *GFlops*.



(a) Compilador *GCC* (b) Compilador *ICC*

Figura 9: Desempenho do algoritmo da biblioteca *PZ* no sistema *SGI-64* compilado com o compilador *GCC* e com o compilador *ICC*.

Utilizando novamente o compilador *ICC*, indicado na figura Figura 9(b), os picos de desempenho obtidos em precisão simples e precisão dupla foram de, respectivamente, 1.159 e 1.409 *GFlops* também na dimensão 320, com desempenhos estáveis de 0.882 *GFlops* em precisão simples e 0.980 *GFlops* em precisão dupla, o que equivale a um ganho de desempenho de mais de 4 vezes em precisão simples e mais de 3 vezes em precisão dupla sobre a versão original compilada com o *GCC*. Assim como no algoritmo de transposição em precisão simples, foi possível notar uma queda progressiva no desempenho com o aumento da dimensão das matrizes. O intervalo de confiança para essa versão foi de aproximadamente 0.005 *GFlops* para os 2 casos. Também podemos notar, na dimensão 2880 para precisão simples e 4672 para precisão dupla, um desvio para baixo na média de desempenho, durante esse desvio o intervalo de confiança cresce para, respectivamente, 0.078 e 0.2 *GFlops*.

Assim como no sistema *MacPro-12*, foi testada uma versão simplificada do algoritmo utilizado pela *PZ*, compilado com o *ICC*, como indicado na figura Figura 10, o desempenho de pico obtido em precisão simples e precisão dupla foram de, respectivamente, 5.219 e 2.848 *GFlops*. Para o desempenho estável, foram atingido um valores de 2.431 *GFlops* em precisão simples e 1.208 *GFlops* em precisão dupla, o que equivale a um ganho de desempenho de mais de 9 vezes em precisão simples e mais de 4 vezes em precisão dupla sobre a versão

original compilada com o *GCC*. Novamente, a diminuição do desempenho com o aumento da dimensão das matrizes estava presente, bem como o desvio da média de desempenho, agora na dimensão 3392 para precisão simples e 4672 para precisão dupla, onde o intervalo de confiança cresce para 0.6 e 0.47 *GFlops*, respectivamente. Nos demais pontos o intervalo de confiança para essa versão foi de aproximadamente 0.14 *GFlops* em precisão simples e 0.22 *GFlops* em precisão dupla.

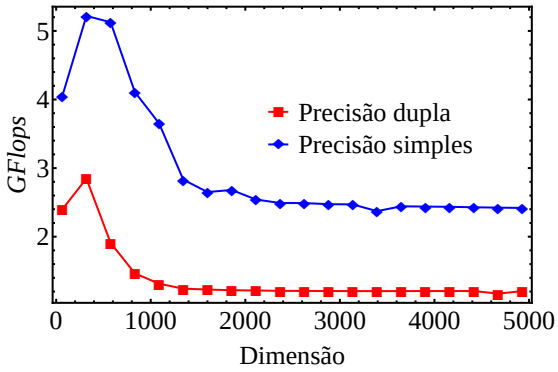


Figura 10: Desempenho da biblioteca *PZ* modificada no sistema *SGI-64* compilada com o *ICC*.

A Tabela 2 resume os resultados de desempenho dos algoritmos de multiplicação de matrizes no sistema *SGI-64*. A tabela apresenta para cada algoritmo o desempenho em *GFlops* da multiplicação de elementos de precisão simples e de elementos de precisão dupla. A coluna "Pico" indica o melhor resultado obtido com os diversos tamanhos de matrizes e a coluna "Moda" indica o resultado mais frequente nos experimentos. Além disso, os algoritmos estão ordenados de acordo com a moda do desempenho.

Algoritmo/Biblioteca	Desempenho em *GFlops*			
	Precisão Simples		Precisão Dupla	
	Pico	Moda	Pico	Moda
ACML (64 *threads*)	426.000	426.000 [a]	196.000	196.000
ACML (1 *thread*)	35.560	35.030	17.770	17.750
Transposição (*ICC*)	7.036	2.752	3.967	1.387
PZ Simples (*ICC*)	5.219	2.431	2.848	1.208
PZ (*ICC*)	1.159	0.882	1.409	0.980
Transposição (*GCC*)	0.330	0.320	0.330	0.296
PZ (*GCC*)	0.280	0.260	0.260	0.250
Ingênuo (*GCC*)	0.256	0.102 [b]	0.226	0.102

[a] A execução não atingiu um máximo estável dentro do intervalo testado
[b] A execução não atingiu um mínimo estável dentro do intervalo testado

Tabela 2: Tabela comparativa de desempenho dos algoritmos no sistema *SGI-64*

No sistema *SGI-64* podemos observar que os resultados se assemelham aos encontrados

no sistema *MacPro-12*, o algoritmo de multiplicação de matrizes ingênuo (0.102 *GFlops*) ainda possui o pior desempenho dentre os algoritmos testados, tendo um desempenho 300 vezes pior do que o desempenho obtido pela biblioteca especializada da *AMD*, a *ACML* (35.03 *GFlops*). Também pudemos observar que o desempenho da biblioteca *ACML* no sistema *SGI-64* com apenas 1 *thread* foi aproximadamente 40% superior ao desempenho obtido com a biblioteca *MKL* no sistema *MacPro-12* muito embora a frequência do processador utilizado no sistema *SGI-64* (2.6 *GHz*) seja inferior à frequência do processador utilizado no sistema *MacPro-12* (2.93 *GHz*) e que o desempenho dos demais algoritmos no sistema *SGI-64* também tenha sido inferior ao desempenho no sistema *MacPro-12*. Considerando execuções em mais de um núcleo a biblioteca *ACML* não apresentou uma moda de desempenho dentro do intervalo considerado, isso resulta em uma utilização efetiva de apenas 17% da capacidade total de cada *thread* contra 75% de utilização obtida pela biblioteca *MKL* no sistema *MacPro-12*. Novamente foi possível utilizar o compilador *ICC* para realizar otimizações agressivas que renderam um ganho de desempenho de pelo menos 4 vezes no algoritmo de transposição (1.387 *GFlops* vs. 0.296 *GFlops*) e pelo menos 3.9 vezes no algoritmo utilizado na *PZ* (0.980 *GFlops* vs. 0.250 *GFlops*). A simplificação da rotina de multiplicação da biblioteca *PZ* para remover estruturas condicionais de dentro de laços, juntamente com a otimização de vetorização do compilador *ICC*, novamente se mostrou uma otimização importante, podendo melhorar o desempenho em até 9 vezes em relação à implementação original (2.431 *GFlops* vs. 0.260 *GFlops*) compilada com o *GCC*. No sistema *SGI-64* também é possível notar que o melhor desempenho obtido com os algoritmos e compiladores testados é aproximadamente 12.7 vezes inferior ao desempenho obtido pela biblioteca do fabricante para o sistema, uma diferença muito superior à observada no sistema *MacPro-12* de, aproximadamente, 5.7 vezes.

6 Resultados experimentais para a plataforma *OpenCL*

A flexibilidade do *OpenCL* permitiu que ele pudesse ser executado nas *CPUs* do *MacPro-12* e *SGI-64* e na *GPU* do *MacPro-12*, dessa forma, os testes com a multiplicação de matrizes em *OpenCL*, foram realizados em todos os sistemas. O algoritmo de blocagem foi implementado com um tamanho padrão de bloco de 16×16.

6.1 *MacPro-12 CPU*

A execução do *OpenCL* na *CPU* do sistema *MacPro-12* obteve um desempenho de pico na dimensão 832 em precisão simples e 576 em precisão dupla de, respectivamente, 36.9 e 30.3 *GFlops*. Embora tenha sido relativamente estável, com intervalos de confiança de 0.5 *GFlops*, o desempenho ainda decaiu até atingir 33.8 *GFlops* em precisão simples e 25.2 *GFlops* em precisão dupla, como indicado na Figura 11.

6.2 *MacPro-12 Radeon*

A Figura 11 indica o desempenho do algoritmo executado na *GPU* do sistema *MacPro-12*. O algoritmo não apresentou nenhuma perda de desempenho com o aumento da dimensão como

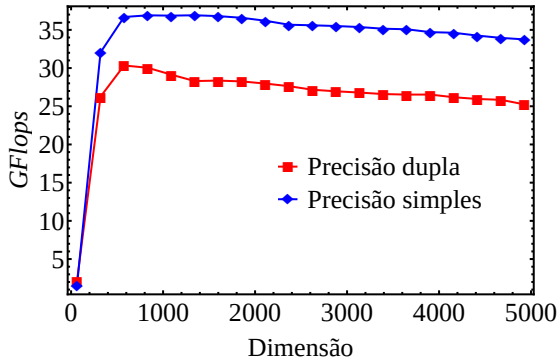


Figura 11: Desempenho do *OpenCL* na *CPU* do sistema *MacPro-12*.

ocorreu na *CPU*. O desempenho escalou até 110.9 *GFlops* na dimensão 1088 e, novamente até 111.4 *GFlops*. Nesse teste o intervalo de confiança foi de 0.7 *GFlops*.

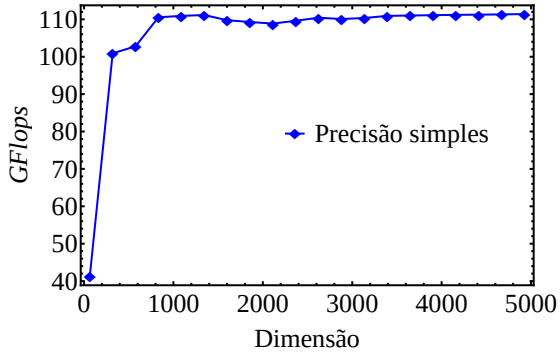


Figura 12: Desempenho do *OpenCL* na *GPU* do sistema *MacPro-12*.

6.3 *SGI-64 CPU*

O sistema *SGI-64* é desprovido de *GPU*, mas foi possível executar o algoritmo na *CPU*, como indicado na Figura 13. Ele obteve um desempenho de 9.45 *GFlops* em precisão simples e 9.98 *GFlops* em precisão dupla, atingidos e mantidos a partir da dimensão 1088. Os intervalos de confiança foram pequenos, se mantendo dentro de 0.03 para ambas as precisões.

6.4 *SGI-64 APPML*

Na tentativa de estabelecermos um patamar de desempenho para o *OpenCL*, utilizamos a biblioteca *APPML*, desenvolvida pela *AMD* especificamente para suas *GPUs*. Infelizmente não foi possível obter nenhum resultado conclusivo da *APPML* uma vez que o programa fornecido para a auto otimização da biblioteca apresentava erro antes do término da otimização.

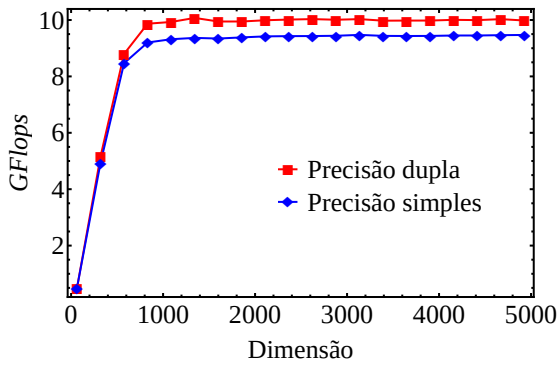


Figura 13: Desempenho do *OpenCL* na *CPU* do sistema *SGI-64*.

A Tabela 3 resume os resultados de desempenho do algoritmo de multiplicação de matrizes na plataforma *OpenCL*. A tabela apresenta para cada sistema o desempenho em *GFlops* da multiplicação de elementos de precisão simples e de elementos de precisão dupla. A coluna "Pico" indica o melhor resultado obtido com os diversos tamanhos de matrizes e a coluna "Moda" indica o resultado mais frequente nos experimentos. Além disso, os algoritmos estão ordenados de acordo com a moda do desempenho.

Algoritmo/Biblioteca	Desempenho em *GFlops*			
	Precisão Simples		Precisão Dupla	
	Pico	Moda	Pico	Moda
MacPro-12 Radeon	111.400	110.900	— [a]	— [a]
MacPro-12 CPU	36.900	33.800 [b]	30.300	25.200 [b]
SGI-64 CPU	9.450	9.450	9.980	9.980

[a] O *hardware* não oferece suporte a operações de precisão dupla
[b] A execução não atingiu um mínimo estável dentro do intervalo testado

Tabela 3: Tabela comparativa de desempenho dos algoritmos no sistema *SGI-64*

Observamos que o desempenho obtido com o algoritmo implementado em *OpenCL*, utilizando todas as *threads* disponíveis, foi aquém às expectativas, a execução na *GPU* do sistema *MacPro-12* atingiu um desempenho inferior a 10% do potencial teórico fornecido pela *GPU* (110.9 *GFlops* vs. 1360 *GFlops*) [1], contudo, a estabilidade do algoritmo sugere que há problemas com o subsistema de memória. O mesmo algoritmo, quando executado na *CPU* do sistema *MacPro-12*, apresentou um desempenho até 6 vezes inferior ao obtido com a *MKL* (33.8 *GFlops* vs. 225.0 *GFlops*), além de apresentar uma queda evidente no desempenho ao longo da execução. O desempenho do algoritmo no sistema *SGI-64* também se apresentou estável, mas foi mais do que 45 vezes pior que a biblioteca *ACML* no mesmo sistema (426.00 *GFlops* vs. 9.45 *GFlops*). Infelizmente, devido a um problema com a biblioteca *APPML*, não foi possível descobrir o desempenho máximo possível utilizando-se uma biblioteca do fabricante.

A Tabela 4 resume os resultados de desempenho de todos os algoritmos utilizados du-

rante esse estudo. A tabela apresenta, para cada algoritmo, o desempenho em *GFlops* da multiplicação de elementos de precisão simples e de elementos de precisão dupla. A coluna "Pico" indica o melhor resultado obtido com os diversos tamanhos de matrizes e a coluna "Moda" indica o resultado mais frequente nos experimentos. Além disso, os algoritmos estão ordenados de acordo com a moda do desempenho.

Sistema	Algoritmo/Biblioteca	Desempenho em *GFlops*			
		Precisão Simples		Precisão Dupla	
		Pico	Moda	Pico	Moda
MacPro-12	*MKL* (12 *threads*)	225.000	225.000 [a]	119.000	119.000
	MKL (1 *thread*)	25.400	25.400	12.600	12.600
	Transposição (*ICC*)	8.400	4.400	4.200	2.100
	PZ Simples (*ICC*)	6.850	3.920 [b]	3.390	1.970
	PZ (*ICC*)	1.600	1.480	1.550	1.270
	Transposição (*GCC*)	0.470	0.460	0.450	0.440
	PZ (*GCC*)	0.460	0.450	0.420	0.440
	Ingênuo (*ICC*)	2.150	0.230 [b]	2.860	0.170
	Ingênuo (*GCC*)	0.400	0.230 [b]	0.390	0.170
SGI-64	*ACML* (64 *threads*)	426.000	426.000 [a]	196.000	196.000 [a]
	ACML (1 *thread*)	35.560	35.030	17.770	17.750
	Transposição (*ICC*)	7.036	2.752	3.967	1.387
	PZ Simples (*ICC*)	5.219	2.431	2.848	1.208
	PZ (*ICC*)	1.159	0.882	1.409	0.980
	Transposição (*GCC*)	0.330	0.320	0.330	0.296
	PZ (*GCC*)	0.280	0.260	0.260	0.250
	Ingênuo (*GCC*)	0.256	0.102 [b]	0.226	0.102
OpenCL	*MacPro-12 Radeon*	111.400	110.900	— [c]	— [c]
	MacPro-12 CPU	36.900	33.800 [b]	30.300	25.200 [b]
	SGI-64 CPU	9.450	9.450	9.980	9.980

[a] A execução não atingiu um máximo estável dentro do intervalo testado
[b] A execução não atingiu um mínimo estável dentro do intervalo testado
[c] O *hardware* não oferece suporte a operações de precisão dupla

Tabela 4: Tabela comparativa de desempenho dos algoritmos nos sistemas testados.

7 Conclusão

Neste relatório nós avaliamos o desempenho de alguns algoritmos de multiplicação de matrizes densas em arquiteturas atuais da *Intel* e *AMD*, bem como as bibliotecas altamente otimizadas de álgebra linear fornecidas por essas empresas. Estudamos também o desempenho de algoritmos incluídos na biblioteca de elementos finitos *PZ* e algoritmos desenvolvidos para a plataforma *OpenCL*, um padrão emergente de programação paralela para sistemas híbridos com *CPUs* e *GPUs*. A partir dos resultados é possível observarmos que as bibliotecas

fornecidas pelos fabricantes foram as que mais utilizaram a capacidade computacional dos sistemas testados, tanto em uma como em várias *threads*. Embora a nossa implementação que obteve melhor desempenho ainda seja pelo menos 13 vezes mais lenta do que a biblioteca do fabricante (transposição vs. *ACML*), essas implementações serviram para mostrar como pequenas otimizações realizadas tendo-se em mente o subsistema de memória permitiram um ganho de desempenho de até 3 vezes sobre a implementação ingênua (transposição *SGI-64*). Ainda mais importante foi a utilização de um compilador otimizante, o *ICC* que, aproveitando-se das otimizações realizadas pensando-se no subsistema de memória, foi capaz de vetorizar o código e permitir um desempenhos pelo menos 12 vezes maiores que a implementação ingênua. Com relação ao *OpenCL*, os resultados obtidos foram muito aquém dos esperados, podendo ser até 40 vezes inferiores aos obtidos utilizando-se a biblioteca fornecida pelo fabricante no mesmo sistema(*OpenCL* vs. *SGI-64*), a perda de desempenho com o aumento da dimensão também foi observada com a implementação em *CPU* do *OpenCL* no sistema *MacPro-12*, sugerindo uma deficiência no uso do subsistema de memória, muito embora o algoritmo de blocagem tenha sido desenvolvido especialmente para contornar esse problema. Com relação à *GPU* foi possível atingir um desempenho em torno de 10% do pico teórico especificado pelo fabricante do *hardware*. Otimizações no algoritmo de multiplicação densa de matrizes em *GPUs* existem na plataforma *CUDA* da *nVIDIA* para aumentar essa porcentagem para até 60% do pico teórico do *hardware* [7] e foram, inclusive, licenciadas e integradas à distribuição do fabricante.

Referências

[1] ATI Radeon HD 5770 GPU Feature Summary. `http://www.amd.com/uk/products/desktop/graphics/ati-radeon-hd-5000/hd-5770/Pages/ati-radeon-hd-5770-specifications.aspx`. [Online; accessado 11-Abril-2013].

[2] A.V. Aho, R. Sethi, and Ullman J.D. *Compilers: Principles, Techniques, and Tools*. Pearson/Addison Wesley, 2 edition, sep. 2006.

[3] Khronos group. The OpenCL specification, version 1.1.

[4] Intel. Intel 64 and ia-32 architectures software developer's manual.

[5] Intel. Intel math kernel library for linux os user's guide.

[6] NVIDIA Corporation. Opencl programming guide for the cuda architecture, 2009.

[7] V. Volkov and J.W. Demmel. Benchmarking gpus to tune dense linear algebra. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1 –11, nov. 2008.