UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA

BRUNO EDUARDO FIRNKES

Análise do desempenho e consumo de energia de multiplicações matriciais através de bibliotecas de álgebra linear em processador de arquitetura Broadwell

Orientador: Prof. Dr. Arthur Francisco Lorenzon

Porto Alegre
2023

“Não é a força
mas a constância dos bons resultados
que conduz os homens à felicidade.”
— FRIEDRICH NIETZSCHE

AGRADECIMENTOS

Primeiramente, e mais importante, à família. Obrigado mãe, pai e irmã. Sem eles eu nada seria. Desde pequeno me ensinam que o trabalho duro e a dedicação trazem frutos. Há dez anos, quando teve início o sonho de me mudar para Porto Alegre e ingressar em uma das melhores universidades dos país, eles estão incondicionalmente do meu lado, apesar da distância. A minha vida acadêmica foi feita de altos e baixos. Diversas greves, troca de curso, aulas interrompidas, até por fim, uma pandemia global de Covid-19. Foram incontáveis os momentos em que eu pensei em desistir, porém, a minha família sempre esteve ao meu lado me incentivando a continuar.

Aos meus amigos, peço desculpas por não me fazer presente ultimamente. A tarefa de concluir um curso de Engenharia na UFRGS, somado ao desafio profissional de assumir a liderança de uma equipe de desenvolvimento, em uma empresa com uma alta expectativa de desempenho, ocuparam muito do tempo, e, principalmente, do meu mental. Sem dúvidas passo pela fase mais desafiadora da minha vida até o momento, porém sei que estou crescendo como nunca cresci. Acredito no meu potencial e resiliência.

Ao Prof. Dr. Arthur Francisco Lorenzon, meu orientador. Foi através dele que tive meu primeiro contato prático com programação. Em meados de 2018 foi meu professor na parte prática da disciplina de Introdução a Programação, enquanto eu ainda cursava Engenharia de Minas. Ao final daquele ano, eu tomei a decisão de trocar de curso e me aventurar no mundo da TI. Para a minha surpresa, no semestre em que eu estava começando o meu Trabalho de Graduação 1 em Engenharia de Computação, cinco anos depois, eis que surge novamente o Arthur como meu professor. O convidei para o desafio de ser meu orientador e ele prontamente aceitou.

Aos meus colegas de trabalho, agradeço a paciência que tiveram comigo nos últimos tempos, há três anos eu falo que falta pouco para a minha conclusão do curso.

RESUMO

Para suprir a demanda por computadores mais rápidos e eficientes, processadores com múltiplos núcleos de processamento se tornou o caminho na busca por computação de alto desempenho. Essa arquitetura permite aproveitar o paralelismo dos núcleos para executar várias tarefas simultaneamente e aumentar significativamente o desempenho de um sistema. Essa arquitetura é adequada para a exploração de técnicas de processamento paralelo, como a programação paralela por *threads*. Esse modelo de programação é adequado para resolver problemas de álgebra linear, pois muitos dos algoritmos de álgebra linear, como multiplicação de matrizes, podem ser paralelizados. Para otimizar esses cálculos, bibliotecas de álgebra linear vem sendo desenvolvidas para aumentar o desempenho de aplicações para uma determinada arquitetura de computador. Comparar essas biblioteca é um trabalho que surge como uma ferramenta para auxiliar projetistas a escolherem qual biblioteca de álgebra linear satisfaz a necessidade do projeto, comparando o consumo e desempenho dessas bibliotecas, visando a exploração do paralelismo por *threads*. Nesse trabalho analisamos e comparamos o desempenho e consumo de energia das bibliotecas IntelMKL, BLIS e OpenBLAS, através da execução das rotinas `GEMM`, em um processador de arquitetura Broadwell.

Palavras-chave: Núcleo. Multithread. Interface de programação paralela. OpenMP. BLAS. BLIS. OpenBLAS. IntelMKL. GEMM. Core.

ABSTRACT

To meet the demand for faster and more efficient computers, processors with multiple processing cores have become the path in the pursuit of high-performance computing. This architecture allows harnessing the parallelism of the cores to execute multiple tasks simultaneously and significantly boost system performance. This architecture is suitable for exploring parallel processing techniques, such as multi-threaded programming. This programming model is well-suited for solving linear algebra problems, as many linear algebra algorithms, such as matrix multiplication, can be parallelized. To optimize these calculations, linear algebra libraries have been developed to enhance application performance for a specific computer architecture. Comparing these libraries emerges as a tool to assist designers in choosing which linear algebra library satisfies the project's needs, by comparing the consumption and performance of these libraries, aiming at exploiting thread-level parallelism. In this work, we analyze and compare the performance and energy consumption of the IntelMKL, BLIS, and OpenBLAS libraries by executing the GEMM routines on a Broadwell architecture processor.

LISTA DE ABREVIATURAS E SIGLAS

BLAS	Basic Linear Algebra Subprograms
GEMM	General Matrix-Matrix multiply
IPP	Interface de programação paralela
LLC	Last level cache
PThreads	Posix Threads

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

BLAS (Basic Linear Algebra Subprograms) surgiu como uma especificação para padronizar as rotinas de operações de álgebra linear, como por exemplo, a rotina GEMM (*General Matrix Multiplication*) que implementa a multiplicação entre duas matrizes. Assim, diversas bibliotecas funcionam como interfaces para as rotinas BLAS, com a Intel MKL (*Math Kernel Library*)[1], que possui otimizações para processadores Intel; Arm Performance Libraries[2] com otimizações para processadores ARM; além de bibliotecas genéricas com otimizações para diversas arquiteturas como a BLIS (*BLAS-like Library Instantiation Software*)[3] que possui uma interface amigável, ATLAS (*Automatically Tuned Linear Algebra Software*)[4], OpenBLAS[5], Eigen[6] e outras.

Essas bibliotecas fornecem uma variedade de funções e rotinas para cálculos matemáticos complexos, sendo essenciais em diversas áreas da computação, tais como processamento de imagens, aprendizado de máquina, simulações científicas e engenharia. Deste modo, devido ao rápido e constante desenvolvimento de novas arquiteturas, é importante que essas bibliotecas sejam otimizadas para tirar o máximo proveito dos recursos de cada arquitetura. A otimização de bibliotecas de álgebra linear pode ter um impacto significativo em outras aplicações que dependem dessas operações, resultando em pesquisas que visam otimizar o desempenho dessas bibliotecas (LORENZON et al., 2022) (TAN et al., 2011).

A programação paralela tem sido amplamente utilizada em operações de álgebra linear para otimizar o seu desempenho. Nela, múltiplos fluxos de instruções são executados de maneira concorrente sobre unidades de processamento independentes (RAUBER; RüNGER, 2013). Assim, para tornar a exploração do paralelismo mais simples e menos suscetível à erros, interfaces de programação paralela são utilizadas, a exemplo de OpenMP (*Open Multi-Processing*) e PThreads(*Posix Threads*). Essas interfaces oferecem abstrações para lidar com a complexidade de programação paralela, permitindo que desenvolvedores possam escrever código para executar em arquiteturas *multicore*.

Dentre todas as rotinas fornecidas pela BLAS, a `GEMM` fornece maior possibilidade de otimização paralela. Isto se dá devido a sua natureza de ser computacionalmente

[1] https://www.intel.com/content/www/us/en/docs/onemkl/get-started-guide/2023-0/overview.html
[2] https://developer.arm.com/Tools%20and%20Software/Arm%20Performance%20Libraries
[3] https://github.com/flame/blis
[4] https://math-atlas.sourceforge.net
[5] https://www.openblas.net
[6] https://eigen.tuxfamily.org

intensiva e exigir um grande conjunto de instruções para serem realizadas, e, muitas vezes, envolvendo um grande número de cálculos que podem ser executados simultaneamente por diferentes núcleos. Neste sentido, uma vez que as bibliotecas de álgebra linear oferecem otimizações de acordo com a arquitetura do computador, algumas bibliotecas podem ser mais otimizadas do que outras para uma dada arquitetura e conjunto de entrada (e.g., tamanho da matriz e tipo dos dados). Portanto, saber qual biblioteca tem melhor desempenho e um menor consumo de energia, ou um balanço entre esses fatores, é de extrema importância para o desenvolvimento de um projeto.

1.1 Objetivos

Considerando que as bibliotecas BLAS implementam otimizações para uma dada arquitetura, esse trabalho surge com o objetivo de avaliar e comparar as bibliotecas BLAS BLIS, OpenBLAS e IntelMKL, através da execução das rotinas `GEMM` em um processador de arquitetura Broadwell. Dito isso, o objetivo principal desse trabalho é:

- Analisar o desempenho das bibliotecas BLAS.
- Analisar o impacto de acessos à memoria no desempenho das bibliotecas BLAS.
- Analisar o o consumo de energia das bibliotecas BLAS.

Para tanto, diversas execuções foram realizadas, alterando parâmetros de execução e explorando as funcionalidades de programação sequencial e paralela por *threads*. Diversos volumes e tipos de dados foram utilizados para abranger um maior domínio de execuções. Através dos dados levantados, como operações em ponto flutuante, acessos à memória *cache*,energia consumida, foi possível comparar as bibliotecas BLAS e analisar comportamentos de desempenho e consumo de energia para uma arquitetura alvo.

1.2 Organização do Texto

O presente trabalho está organizado da seguinte forma: no Capítulo 2 abordamos os conceitos teóricos, em que na Seção 2.1 abordamos as arquiteturas *multicore*; na Seção 2.2 abordamos sobre computação paralela, tratando sobre programação paralela, padrões de programação paralela e interfaces de programação paralela; na Seção 2.3 abordamos sobre as bibliotecas BLAS, introduzindo as que foram utilizadas nesse trabalho; na Seção

2.4 abordamos os trabalhos relacionados. No Capítulo 3 tratamos sobre a metodologia abordada, apontando as aplicações alvo e o ambiente de execução utilizado. No Capítulo 4 há os resultados das execuções, em que foi realizado análises de desempenho e consumo de energia para uma variedade de dados. Por fim, no Capítulo 5 há a conclusão e discussão final sobre esse trabalho.

2 CONCEITOS TEÓRICOS

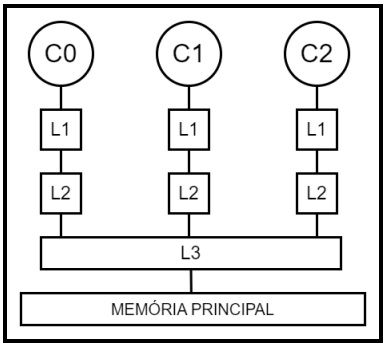
Neste capítulo apresentamos os conceitos fundamentais utilizados no trabalho. Inicialmente abordamos arquiteturas *multicore*, explicando como essa arquitetura está estruturada, bem como é realiza a comunicação pelos múltiplos *cores*. Após, tratamos sobre computação paralela, abordando programação paralela, seus modelos e padrões de programação e interfaces de programação paralela, como a OpenMP e PThreads.

2.1 Arquiteturas Multicore

Arquiteturas multiprocessadas caracterizam-se por possuírem múltiplas unidades de processamento, que podem compartilhar acesso a um mesmo espaço de endereçamento (ROSE, 2002). Nesse tipo de arquitetura, é possível delegar diferentes partes de uma aplicação (*e.g.,* tarefas) para cada núcleo de processamento de modo que a execução dessas ocorra de forma concorrente, melhorando o desempenho da aplicação. Cada núcleo dessa arquitetura é independente com os seus próprios recursos, como registradores, *pipeline* de execução, unidades de memória *cache*, unidade lógica e aritmética, além de possuírem níveis de memória compartilhada, facilitando a comunicação entre os demais. A Figura 2.1 representa a estrutura básica de processador *multicore* com três núcleos de processamento (C0, C1 e C2), cada um com dois níveis de memória privada (L1 e L2) e uma memória compartilhada (L3) além da memória principal.

Um processador *multicore* pode ser dividido em dois grupos de acordo com a organização (VAJDA, 2011): homogêneos e heterogêneos. Um processador com organização homogênea é aquele onde os núcleos são idênticos em termos de arquitetura, frequência base de operação, memória *cache* e outras características. Isso significa que

Figura 2.1 – Estrutura básica de um processador *multicore* com três núcleos



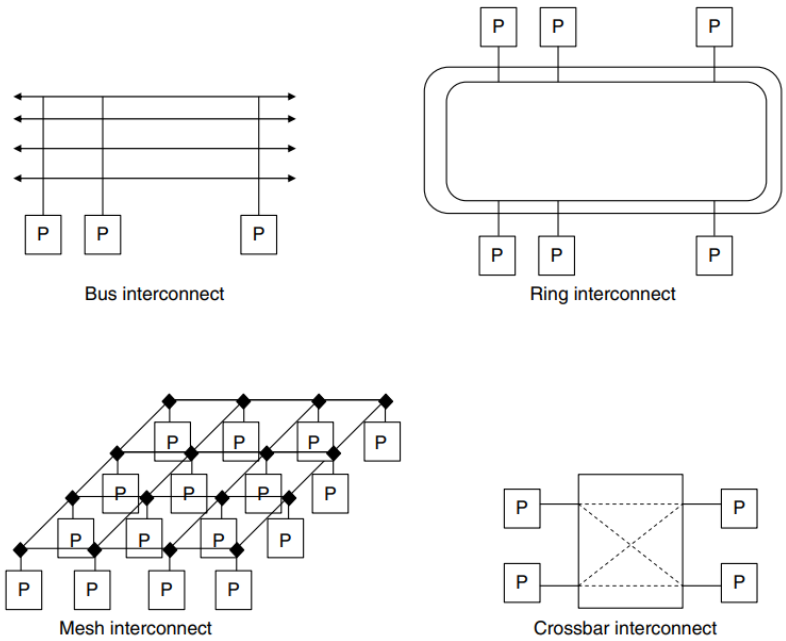
Fonte: o autor

todos os núcleos podem executar as mesmas tarefas e entregarão resultados de desempenho e consumo de energia similares. Esse grupo apresenta baixo grau de dificuldade de programação, uma vez que não há necessidade de lidar com componentes de diferentes arquiteturas. Por outro lado, um processador com organização heterogênea possui núcleos distintos, cada qual com suas características únicas. Alguns núcleos podem ser projetados para realizar tarefas específicas, como processamento gráfico ou inteligência artificial, enquanto outros podem ser projetados para tarefas mais gerais. Há exemplo da arquitetura híbrida Alder Lake, produzida pela Intel (ROTEM et al., 2022), onde há núcleos específicos para desempenho, conhecidos como "P *Cores*", e núcleos para melhor eficiência energética, "E *Cores*".

A comunicação entre os núcleos pode ser realizada através de uma rede de interconexão, que permite a coordenação, a sincronização e a troca de dados entre os diferentes núcleos. A Figura 2.2 representa os diferentes tipos de organização de uma rede de interconexão. A organização da rede é importante pois descreve a topologia do processador, definindo a organização dos *links*, processadores, unidades de memória e tem um impacto importante no roteamento das mensagens e dados entre os processadores visando uma menor latência e um menor consumo da rede de comunicação.

Devido a essa comunicação entre os diferentes núcleos, e o acesso a memoria destes, através das redes de comunicação, é comum a dependência e concorrência de dados.

Figura 2.2 – Redes de intercomunicação



Fonte: (VAJDA, 2011)

A dependência ocorre quando um núcleo depende de um dado que está sendo utilizado por outro(s). A concorrência de dados ocorre quando diferentes núcleos tentam ler ou escrever um dado na memoria de forma simultânea. Para isso, os processadores implementam um sistema de coerência de memória cache que permite o acesso à dados compartilhados em sua unidade de memória privada enquanto mantém a consistência quando outro processador deseja atualizar dados compartilhados (VAJDA, 2011). Para isso, são empregados protocolos de coerência, a exemplo do protocolo MESI (PAPAMARCOS; PATEL, 1984).

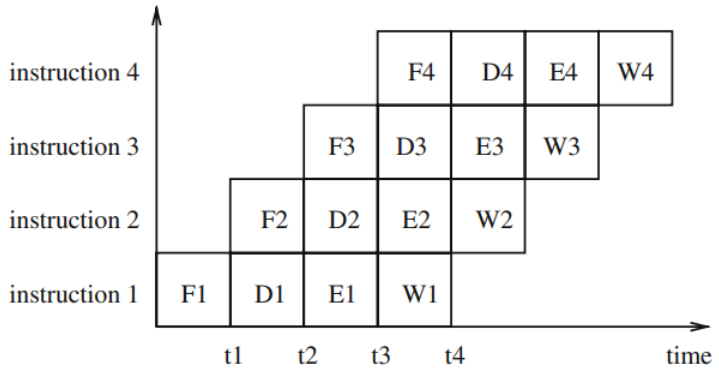
O protocolo MESI é nomeado após os quatro estados de uma linha de *cache* no protocolo: *Modified* (M), *Exclusive* (E), *Shared* (S), *Invalid* (I). A máquina de estados do protocolo pode ser descrita seguindo (CULLER; SINGH; GUPTA, 1998). Quando um núcleo lê ou grava pela primeira vez em uma linha de *cache*, ele entra no estado (E). Nesse estado, o núcleo tem acesso exclusivo à linha de *cache* e pode ler ou gravar nela sem interferência de outros núcleos. Se outro núcleo tentar ler a mesma linha, ele entrará no estado (S). Se um núcleo quiser gravar em uma linha que está no estado (S), ele deve primeiro solicitar a propriedade da linha de *cache* enviando uma mensagem para os outros núcleo para invalidar suas cópias. Uma vez concedida a propriedade, a linha de *cache* entra no estado (M) e o processador pode gravar nela. Se uma linha contém dados que não são mais válidos, ela estará no estado (I).

Processadores *multicore* dão a funcionalidade de programação paralela com uma sustentável redução no tempo de computação (JADON; YADAV, 2016). Neste sentido, para explorar o paralelismo que arquiteturas *multicores* permitem, visando ganhos de desempenho e consumo de energia, interfaces de programação paralela (IPPs) são utilizadas e serão abordadas na seção 2.2.

2.2 Computação Paralela

Computação paralela é um tipo de computação em que muitos cálculos ou processos são realizados simultaneamente (ALMASI; GOTTLIEB, 1989). Há diferentes níveis de computação paralela: (1) Paralelismo a nível de *bit*: baseado no aumento de número de *bits* para representar uma *word*. Aumentar o tamanho da *word* reduz o número de instruções necessária para executar uma operação em uma variável cujo tamanho é maior que o tamanho de uma *word*. A computação até 1986 foi dominada por avanços no paralelismo a nível de *bit*, com processadores de 4-*bits* avançaram para 8-*bits*, posteriormente para 16-

Figura 2.3 – Paralelismo a nível de instrução



Fonte: (RAUBER; RüNGER, 2013)

bits e assim por diante. (CULLER; SINGH; GUPTA, 1998). (2) Paralelismo a nível de instrução: baseado na execução concorrente de diferentes instruções. Paralelismo a nível de instrução é como uma fábrica automobilística (RAUBER; RüNGER, 2013), diferentes etapas (*fetch*, *decode*, *execute*, *write-back*) do *pipeline* podem executar de forma paralela. Exemplo Figura 2.3. (3) Paralelismo a nível de *thread*: baseada na execução concorrente de diferentes *tasks* (conjunto de instruções) em diferentes núcleos em um processador *multicore*.

2.2.1 Programação Paralela

Programação paralela é definida como a divisão de tarefas de uma aplicação que podem ser executadas de forma concorrente com o objetivo de reduzir o tempo total de computação (RAUBER; RüNGER, 2013).

Conforme (FOSTER, 1995), a paralelização pode ser feita através dos seguintes passos: (1) particionamento: transformar uma tarefa maior e os dados necessários para a computação em um conjunto de tarefas menores; (2) comunicação: a comunicação requirida para a coordenação e execução das tarefas, estrutura e protocolos de comunicação são definidos; (3) aglomeração: se necessário, combinar tarefas definidas no passo (1) em tarefas maiores para otimizar o desempenho ou reduzir custos de desenvolvimento; (4) mapeamento: atribuição das tarefas definidas nas etapas anteriores à núcleos com o objetivo de maximizar a utilização dos núcleos.

(FOSTER, 1995) descreve diferentes modelos de programação paralela:

- Troca de mensagens: processos paralelos se comunicam uns com os outros através

da troca de mensagens, do tipo *send*/*receive* em vez de compartilhar diretamente a memória ou recursos de processamento. Cada processo tem sua própria memória e executa suas tarefas independentemente, mas precisa comunicar-se com outros processos para trocar informações e coordenar suas atividades.

- Memória compartilhada: baseado na existência de uma memória principal que pode ser acessada por todos os núcleos. Cada núcleo, além de possuírem unidades de memória privadas, podem ler e escrever dados nessa espaçamento de memória compartilha.
- Paralelismo de dados: explora a concorrência de que uma mesma operação pode ser executada a vários elementos de uma estrutura de dados.

2.2.2 Padrões de Programação Paralela

Conforme (RAUBER; RüNGER, 2013), para estruturar um programa paralelo, há diversas formas de padronização da organização das tarefas a serem executadas, provendo estruturas de coordenação para os processos ou *threads*. As principais formas são:

- *Fork-Join*: a *thread* principal, também conhecida como *thread* mestre, cria um número de *threads* filhas, através do comando *fork*, que são alocadas para recursos computacionais físicos (núcleos) do sistema. As *threads* filhas trabalham em paralelo de forma independente para executar uma tarefa que lhe foi atribuída. A *thread* principal pode aguardar a finalização do trabalho das filhas através do comando *join*.
- *Parbegin-Parend*: permite que um conjunto de instruções sejam executadas em paralelo. Um programa é divido é múltiplas seções, onde cada uma pode ser executada em paralelo. *Parbegin* indica o início da seção a ser paralelizada e *parend* indica o fim dessa seção.
- *Mestre-escravo*: um processo ou *thread* "mestre" é responsável pela criação e distribuição do trabalho para múltiplos outros processos ou *threads* "escravas". As "escravas" executam a tarefa que foi atribuída e retornam o resultado para o "mestre".
- *Task Pools*: um estrutura de dados é criada, geralmente uma fila, para alocar tarefas a serem executadas. As *threads* podem salvar e recuperar, isso é, atribuir para si, as tarefas. Quando uma *thread* está disponível, ela recupera uma tarefa da fila, a

executa e ao finalizar a tarefa a *thread* pode recuperar outra da *pool*.

2.2.3 Interfaces de Programação Paralela

Para facilitar a exploração do paralelismo, interfaces de programação paralela (IPPs) vêm sendo amplamente utilizadas. Exemplos de IPPs incluem OpenMP [1], MPI (Message Passing Interface), CUDA (Compute Unified Device Architecture)[2], OpenCL (Open Computing Language)[3], PThreads. Essas interfaces proveem um conjunto de funções que permitem o programador a expressar paralelismo em seu código e mecanismos para gerenciar a comunicação e a sincronização entre diferentes processos ou *threads*.

Nas Subseções seguintes, vamos tratar das IPPs OpenMP e PThreads A PThreads é um padrão de programação paralela, e amplamente utilizada em ambientes Unix e a OpenMP por ser amplamente utilizada no contexto de computação de alto desempenho. Além disso, as bibliotecas BLAS selecionadas para a análise e comparação, conforme tratado na Seção 1.1, fazem uso dessas interfaces.

2.2.3.1 OpenMP

OpenMP é um IPP para memória compartilhada em C/C++ e FORTRAN, que consiste em um conjunto de diretivas do compilador, funções de biblioteca e variáveis de ambiente (CHAPMAN; JOST; PAS, 2007). Alguns exemplos de uso são para simulações de acidentes de carros e análise de impactos; simulações de estruturas; análise de sistemas de telecomunicações; simulações de astrofísica; mapeamento de DNA e outros. O paralelismo é explorado por meio da inclusão de diretivas, em um formato especial no código sequencial, que informam ao compilador como e quais partes do código devem ser executadas em paralelo. Cabe ao programador especificar explicitamente as ações a serem tomadas pelo compilador e pelo sistema em tempo de execução para executar o programa paralelo.

OpenMP implementa o padrão *Fork-Join* de programação paralela. Assim, para indicar ao compilador que um dado bloco de código será executado em paralelo, deve-se utilizar a diretiva `parallel`, através da chamada `#pragma omp parallel`. Quando uma *thread* encontra este construtor paralelo, um conjunto de *threads* é criada para exe-

[1] https://www.openmp.org
[2] https://developer.nvidia.com/cuda-toolkit
[3] https://www.khronos.org/opencl

cutar a região paralela associada, que é o código contido dentro do bloco. Embora este construtor garante que as tarefas sejam executados em paralelo, ele não distribui a carga de trabalho da região entre o conjunto de *threads*. Ao final de uma região paralela, há uma barreira implícita, isto é, um ponto na execução do programa além do qual nenhuma *thread* do conjunto pode avançar até que todas as *threads* tenham atingido a barreira e as tarefas atribuídas tenham sido executadas (OpenMP Architecture Review Board, 2013). A sintaxe do construtor `parallel` é definida em C/C++:

```
#pragma omp parallel [clause[ [, ]clause] ...]
```

Visto que a apenas o construtor `parallel` não distribui as tarefas entre as *threads*, outras diretivas devem ser utilizadas para esse objetivo. A exemplo do construtor `for`. O construtor `for` especifica quais iterações de um ou mais *loops* associados serão executadas em paralelo. As iterações são então divididas entre o conjunto de *threads*. A sintaxe do operador `for` é definida em C/C++:

```
#pragma omp for [clause[[,] clause] ...  ]
```

O uso dos construtores *parallel* e *for*, no bloco de código destacado no Algoritmo 1, executando num conjunto de 4 *threads*, e num total de 9 interações, tem a saída conforme a Figura 2.4. Pode-se notar que não há uma ordem específica para a execução tanto das *threads* quanto das tarefas (nesse caso as iterações) e que o fim da região paralela é encontrado após a finalização de todas as tarefas do bloco da região paralela.

Algorithm 1 Algoritmo paralelo simples em OpenMP

```
1: printf("Inicio regiao paralela.")
2: #pragma omp parallel for
3: for (int i = 0; i < n; i++) do
4:     printf("Thread: %d executa a iteracao: %d", omp_get_thread_num(), i)
5: end for
6: printf("Fim regiao paralela)
```

Figura 2.4 – Saída do exemplo dos construtores `parallel` e `for` com 4 *threads* e n=9

```
Inicio regiao paralela.
Thread=0 executa a iteracao i=0
Thread=0 executa a iteracao i=1
Thread=0 executa a iteracao i=2
Thread=3 executa a iteracao i=7
Thread=3 executa a iteracao i=8
Thread=2 executa a iteracao i=5
Thread=2 executa a iteracao i=6
Thread=1 executa a iteracao i=3
Thread=1 executa a iteracao i=4
Fim regiao paralela.
```

Fonte: o autor

O número total de *threads* do conjunto pode ser definido pela variável de am-

biente `OMP_NUM_THREADS`. Para estipular o número de *threads* do conjunto, pode-se atribuir um valor inteiro a variável `OMP_NUM_THREADS=N` ou através da chamada da função `omp_set_num_threads(N)` ou definir o número de *threads* para uma seção específica, através da diretiva `num_threads(N)`.

2.2.3.2 Posix Threads

Pthreads é uma IPP definida pelo padrão IEEE POSIX 1003.1c (IEEE..., 1997). É um modelo de execução que especifica como o trabalho é realizado e permite que um programa gerencie um conjunto de *threads*. É usada para implementar paralelismo através do compartilhamento de memória. Implementações da API estão disponíveis em muitos sistemas operacionais baseados em UNIX que são compatíveis com POSIX (Linux, macOS, Solaris, entre outros) e é incluída como uma biblioteca chamada *libpthread*. Assim como em OpenMP, o paralelismo é realizado através da criação de *threads*, divisão e atribuição de tarefas e sincronização.

A *thread* inicial de um processo é criada quando um processo é criado. *Threads* filhas adicionais são criadas explicitamente através da chamada da função `pthread_create`, conforme sintaxe em C/C++ definida como:

```
int pthread_create(pthread_t * tID, pthread_attr_t * attr,
void *(*start_routine)(void *), void *arg);
```

Onde *tID* guarda um identificador da *thread*; *attr* é um parâmetro adicional que define os atributos da *thread* e geralmente é enviado o argumento *null*, atribuindo valores padrão (como o algoritmo de escalonamento, tipo de *thread* e outros); *start_routine* é um ponteiro para a rotina que será executada pela *threads*; *args* é um ponteiro para os argumentos a serem enviados para a função *start_routine*. Em caso de sucesso na criação da *thread*, a função retorna o valor 0, caso contrário retorna um valor que indica o erro. Uma vez que a *thread* foi criada, a execução de *start_routine* é eventualmente iniciada.

A *thread* filha é encerrada no momento em que a função *start_routine* encerra a sua execução. Para ocorrer a sincronização de *threads*, é necessário a adição de uma barreira explicita através da chamada da função `pthread_join`. A função tem a sintaxe em C/C++ definida como:

```
int pthread_join(pthread_t tID, void **thread_return);
```

Onde *tID* é o identificador da *thread*, retornado no argumento *tID* da chamada *pthread_create*; e *thread_return* retorna o estado em que a *thread* foi encerrada.

Por PThreads ser uma IPP de memória compartilhada, é importante garantir que

não ocorram acessos de escrita e leitura em memória de forma concorrente. Para isso técnicas de exclusão mútua devem ser empregadas, a exemplo de *mutex*. Variáveis do tipo *mutex* são a principal forma que o PThreads apresenta para a proteção de regiões críticas (NICHOLS; BUTTLAR; FARRELL, 1996). Um *mutex* protege algum recurso, como um endereço de memória que duas *threads* podem estar acessando. Essa variável é criada e destrúida através das chamadas `pthread_mutex_init` e `pthread_mutex_destroy`, respectivamente. As funções tem a sintaxe em C/C++ definidas como:

```
int pthread_mutex_init(pthread_mutex_t *mutex,
const pthread_mutexattr_t *attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Onde o argumento *mutex* é uma referência para a variável do *pthread_mutex_t*; e *attr* é um argumento que define os atributos da variável (como o protocolo utilizado, se será feito a detecção de erros e outros). Para solicitar acesso à região da seção crítica, é necessário realizar a chamada da função *pthread_mutex_lock* e para liberar a região para que outra *thread* acesse, é necessário chamar a função *pthread_mutex_unlock*. As funções tem a sintaxe em C/C++ definidas como:

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Onde *mutex* é a referência para a variável do tipo *pthread_mutex_t*. O uso de *mutex* deve ser feito de forma correta para evitar *deadlocks*, i.e. uma *thread* ficar impedida de continuar sua execução (BUTENHOF, 1997).

No seguinte exemplo, em C, são utilizadas as funções descritas acima e tem a sua saída conforme a Figura 2.5:

Figura 2.5 – Saída do exemplo de uso das funções da PThreads com duas *thread* criadas

```
Faz algo 1 iniciou
Faz algo 1 terminou
Faz algo 2 iniciou
Faz algo 2 terminou
```

Fonte: o autor

2.3 Basic Linear Algebra Subprograms

Devido ao fato de álgebra linear permitir um grande reuso de dados, muitas rotinas podem ser executadas dezenas ou centenas de vezes mais rapidamente quando ajustadas

Algorithm 2 Algoritmo paralelo simples em PThreads

```
1: #include <pthread.h>
2: #include <stdio.h>
3: pthread_mutex_t mutex;
4: pthread_t thread_id[2];
5: int contador;
6: function void* fazalgo(void* arg)
7:     pthread_mutex_lock(&mutex);
8:     pthread_mutex_lock(&mutex);
9:     contador += 1;
10:    printf("Faz algo %d iniciou", contador);
11:    for (int i = 0; i < 99999999999; i++) do
12:    end for
13:    printf("Faz algo %d terminou", contador);
14:    pthread_mutex_unlock(&mutex);
15:    return NULL;
16: end function
17: function int main()
18:    int i = 0;
19:    pthread_mutex_init(&mutex, NULL);
20:    while (i < 2) do
21:        pthread_create(&(thread_id[i]), NULL, &fazalgo, NULL);
22:        i++;
23:    end while
24:    pthread_join(thread_id[0], NULL);
25:    pthread_join(thread_id[1], NULL);
26:    pthread_mutex_destroy(&mutex);
27:    return 0;
28: end function
```

para o hardware do que quando escritas de forma ingênua (ROBERT et al., 2011). Em muitas aplicações, o desempenho de operações de álgebra linear é a principal limitação que cientistas encontram ao modelarem problemas mais complexos, que aproximariam os problemas da realidade. BLAS (*Basic Linear Algebra Subprograms*) surgiu nesse contexto com o intuito de otimizar operações de álgebra linear e reduzir o tempo de processamento desses problemas.

BLAS são rotinas que fornecem blocos de construção padrão para realizar operações básicas de vetores e matrizes. Como as BLAS são eficientes, portáteis e amplamente disponíveis, são comumente usados no desenvolvimento de software de álgebra linear (AT&T Bell Laboratories, 1980). Inicialmente descritas por (LAWSON et al., 1979), as rotinas BLAS foram desenvolvidas com o objetivo de fornecer uma coleção padronizada de rotinas de baixo nível para operações de álgebra linear. Essas rotinas foram projetadas para serem eficientes e portáveis, permitindo que os usuários as utilizassem em diferentes plataformas e arquiteturas.

Baseado nos tipos de operandos, as rotinas podem ser divididas em três níveis:

- **Nível 1** (LAWSON et al., 1979): operações de ordem O(n) envolvendo vetor-vetor,

como produto escalar, vetor mais um escalar vezes um vetor, rotação de Givens, cópia, troca, norma Euclidiana, soma de magnitudes, multiplicação de um escalar por um vetor e localização de um elemento de maior magnitude.

- **Nível 2** (DONGARRA et al., 1988): operações de ordem O(nn) envolvendo matriz-vetor, como multiplicação e soma matriz-vetor, atualizações *rank-1* e *rank-2*, soluções de equações lineares.
- **Nível 3** (DONGARRA et al., 1990): operações de ordem O(nnn) envolvendo matriz-matriz. como multiplicação e soma matriz-matriz, atualizações *rank-k* e *rank-2k*, soluções de equações lineares.

As rotinas BLAS de Nível 3 serão o foco desse trabalho. É reconhecido que um alto desempenho resulta principalmente do uso de rotinas BLAS de Nível 3, especificamente a multiplicação matriz-matriz. Através da reutilização de dados armazenados em *cache*, essas rotinas diminuem a lacuna entre a alta velocidade do processador e o tempo de acesso à memoria (DONGARRA et al., 2003).

Conforme (DONGARRA et al., 1990), as rotinas BLAS de Nível 3 podem ser dividas em 4 grupos conforme o tipo do dado envolvido na operação: *real*, *double precision*, *complex* e *double complex*. A identificação da rotina segue a convenção: o primeiro caractere identifica o tipo do dado, e são "S" para dados do tipo *real*, "D" para *double precision*, "C" para *complex* e "Z" para *double complex*; o segundo e terceiro caractere identifica o tipo das matrizes envolvidas, sendo "GE" para matrizes gerais retangulares, "HE" para matrizes hermitiana, "SY" para matrizes simétrias e "TR" para matrizes triangulares; o quarto, quinto, e caso exita sexto caractere, denotam o tipo da operação, "MM" para multiplicação de matrizes, "RK", para atualizações *rank-k*, "R2K" para atualizações *rank-2k* e "SM" para a solução de sistemas de equações lineares.

Existem diversas bibliotecas que implementam as operações da BLAS, cada uma com particularidades, como as bibliotecas BLIS (*BLAS-like Library Instantiation Software*), ATLAS (*Automatically Tuned Linear Algebra Software*), GotoBLAS, OpenBLAS, CuBLAS (NVIDIA), Intel MKL (Math Kernel Library), Arm Performance Libraries, Eigen BLAS e outras.

Considerando a popularidade e facilidade de uso por parte do desenvolvedor de software, este trabalho abordará as bibliotecas BLIS, IntelMKL e OpenBLAS por serem as mais utilizadas e difundidas, além de terem documentado qual interface de programação paralela é utilizada para a exploração do paralelismo e principalmente, no contexto desse trabalho, por estarem atualmente em desenvolvimento otimizando as suas imple-

mentações para uma arquitetura específica.

2.3.1 BLIS

BLIS é uma biblioteca de software que permite a instanciação de rotinas BLAS (ZEE; GEIJN, 2015). Sua principal inovação é que as operações BLAS de nível 2 e 3 podem ser expressas e otimizadas em simples chamadas de *kernels*, i.e., chamadas de funções. A biblioteca é desenvolvida e mantida pelo grupo Science of High-Performance Computing (SHPC) do Oden Institute for Computational Engineering and Sciences na Universidade do Texas em Austin e pelo grupo de pesquisa Matthews na Southern Methodist University. Por ser um trabalho em desenvolvimento, BLIS vêm se adaptando aos processadores modernos e fornece otimizações de acordo com a arquitetura do computador.

BLIS permite múltiplos níveis de *multithreading* para quase todas as operações de nível 3. Embora BLIS por padrão não explore os benefícios de processamentos em *multithreads*, os desenvolvedores podem explicitamente habilitar essa funcionalidade e escolher entre usar a IPP OpenMP ou PThreads. Em sua documentação, os desenvolvedores da BLIS sugerem o uso de OpenMP ao invés de PThreads, devido ao fato de OpenMP permitir ao desenvolvedor vincular *threads* a núcleos. Isso é importante porque quando o sistema operacional faz com que uma *thread* migre de um núcleo para outro, a *thread* geralmente deixará para trás os dados que estava usando nas caches L1 e L2, reduzindo o desempenho da aplicação (ZEE; GEIJN, 2015).

BLIS tem como suas principais funcionalidades:

- Portabilidade: identifica chamadas de *kernels* e fornece otimizações transparentes para os desenvolvedores;
- Armazenamento de matrizes: exporta interfaces que permite o desenvolvedor escolher a melhor maneira de armazenar vetores e matrizes;
- Suporte ao domínio dos complexos: permite diversas operações envolvendo números complexos, inclusive garante suporte para *kernels* no domínio dos complexos, caso o desenvolvedor tenha esquecido de implementar;
- Suporte a *multithread*: como descrito, embora por padrão BLIS não aplique programação paralela por threads, é possível habilitar a funcionalidade e optar pela IPP OpenPM ou PThreads;

- Facilidade no uso: fácil de usar independente do usuário. Uma camada opcional de compatibilidade com BLAS fornece aos desenvolvedores compatibilidade com códigos BLAS já existentes. Também é possível ajustar ou escrever seu próprio código para aproveitar as funcionalidades providas pela BLIS;
- API e *kernels* expostos: BLIS expõem as suas funções permitindo que o desenvolvedor otimize o que desejar;

2.3.2 OpenBLAS

OpenBLAS é uma BLAS otimizada baseada na GotoBLAS (GOTO, 2002). Foi inicialmente desenvolvida para processadores com a arquitetura *Loongson 3A*, porém atualmente, assim como BLIS e ATLAS, é um trabalho de pesquisa em andamento e fornece otimizações para diferentes arquiteturas como x86, Itanium, Power, SPARC (*Scalable Processor Architecture*) e MIPS64 (XIANYI; QIAN; YUNQUAN, 2012).

Atualmente, OpenBLAS oferece aos seus usuários a possibilidade de escolherem a IPP OpenMP ou PThreads para a funcionalidade de paralelismo de *threads*. OpenBLAS implementa por padrão PThreads, e cabe ao usuário explicitamente, através de variáveis de ambiente, escolher caso deseje a utilização de OpenMP.

2.3.3 IntelMKL

Intel Math Kernel Library (MKL) é composta por bibliotecas de matemática de alto desempenho e *multithreading* para álgebra linear, transformadas de Fourier, matemática vetorial (Intel, 2003). Possui interfaces para desenvolvimento em C e Fortran. MKL acelera o desempenho das aplicações que as usam pois a biblioteca é otimizada para as últimas gerações de processadores Intel, como os processadores Intel Core, Xeon e Xeon Scalable Performance. Segundo a Intel, é bilbioteca mais rápida e utilizada em seus processadores.

Intel MKL habilita a IPP OpenMP por padrão. Assim, qualquer aplicação que faz uso de suas bibliotecas é paralelizada de forma automática em arquiteturas *multicore*. Diferente da BLIS e OpenBLAS, a MKL permite a alocação apenas de núcleos físicos, não fazendo uso da tecnologia de *multithreading*.

2.4 Trabalhos Relacionados

A biblioteca IntelMKL possui em sua página relatórios de desempenhos em diferentes processadores Intel para diferentes domínios, como operações `GEMM`. A biblioteca oferece documentes sobre configurações e detalhes sobre os relatórios de desempenho, além de fornecer resultados considerando diferentes conjuntos de *threads* e tamanhos das matrizes.

BLIS, por outro lado, possui documentos mais elaborados no contexto desse trabalho. BLIS compara o desempenho de seus *kernels* com demais bibliotecas, como a ATLAS, MKL, OpenBLAS, ARM Performance Libraries, além de executar seus testes em diferentes processadores com simulações com diferentes conjuntos de *threads*. BLIS possui documentado resultados de desempenho para os processadores com arquitetura ThunderX2, SkylakeX, Haswell, Zen, A64fx e NeoverseN1. BLIS comprovou que seu desempenho é superior as outras bibliotecas BLAS em todas as arquiteturas de processadores, exceto na arquitetura Haswell onde provaram melhores desempenhos que a biblioteca MKL ao rodarem os testes com conjunto de 12 e 24 *threads* e SkylakeX onde a biblioteca MKL se mostrou superior em todas as simulações (ZEE; GEIJN, 2015).

(FIBICH et al., 2020) também comparou o desempenho de diferentes bibliotecas BLAS, como a ATLAS, BLIS e OpenBLAS. Nesse trabalho As bibliotecas foram testadas em dois tipos de processadores ARM (Cortex-A53 e Cortex-A72) e um processador RISC-V (SiFive U540). Os resultados mostraram que as bibliotecas tiveram desempenho diferente em cada processador, com a OpenBLAS e a BLIS obtendo os melhores resultados em geral.

Em (SOLIMAN, 2008), foram avaliadas diferentes implementações dos *kernels* básicos, sem nenhuma biblioteca para otimizações, em uma variedade de tamanhos de matriz e tipos de dados, em processador Intel Xeon Dual Core. Os resultados mostraram que as operações de nível 3 mostraram melhor desempenho quando era explorado o paralelismo de *threads*. Por outro lado, as operações nível 1 e nível 2 se mostraram pior quando rodando com *multithread*. Eles provam que *multithread* é um técnica ineficiente para um problema com conjunto pequeno de dados devido ao *overhead*(sobrecarga) da criação das *threads*.

Em (LORENZON et al., 2022), os autores propuseram uma implementação de paralelismo baseado em tarefas para o *kernel* GEMM da BLIS, que é originalmente baseado no modelo *fork-join*. Eles realizaram simulações nas arquiteturas Intel Xeon e AMD

EPYC. A implementação forneceu melhoria de desempenho e maior maleabilidade para as aplicações que usam o *kernel* GEMM.

3 METODOLOGIA

Nesse Capítulo tratamos da aplicação alvo, introduzindo as interfaces genéricas para a implementação de cada biblioteca, explicando os argumentos de cada função, e trazendo exemplos práticos dos algoritmos para as implementações da `GEMM`. Após, tratamos sobre o ambiente de execução, com informações sobre o processador e a organização da memória, além de abordamos como serão obtido os resultados. Nesse Capítulo também organizamos a divisão em Grupos para a execução dos algoritmos, tratando sobre o domínio de entradas das execuções.

3.1 Aplicação Alvo

Como já citado, o foco desse trabalho foram as rotinas BLAS de Nível 3, em específico a multiplicação de matrizes retangulares (`GEMM`). Executou-se a rotina `SGEMM` para dados do tipo *real*, `DGEMM` para *double precision*, `CGEMM` para *complex* e `ZGEMM` para *double complex*. Para os experimentos, utilizou-se três matrizes A, B e C de dimensões m=n=k. As bibliotecas utilizadas nos experimentos foram as seguintes: BLIS, OpenBLAS e MKL, descritas na Seção 2.3.

3.1.1 Implementação com BLIS

Para a execução da BLIS, usou-se o tipo de dado *float* para representar os *reals*; *double* para *double precision*; *scomplex* para representar *complex*, sendo *scomplex* uma estrutura do tipo *float real; float imag;*; e *dcomplex* para representar *double complex*, sendo *dcomplex* uma estrutura do tipo *double real; double imag;*. Além disso, também utilizou-se os tipos inteiro *dim_t* para representar o tamanho das matrizes e *inc_t* para representar os incrementos para percorrer a matriz. Os algorítimos seguiriam inicialmente com a declaração das variáveis e seus respectivos tipos, após isso foi feita a alocação de memória para armazenar os valores das matrizes através da rotina *malloc*. Após foi a inicialização dos valores das matrizes: como os dados foram armazenados de forma sequencial, a inicialização se deu por um laço de repetição *for* inicializando cada valor da matriz A e B como 1 em caso de número reais e para $\{1, 0\}$ para números complexos; a matriz C foi inicializada como 0 para os reais e $\{0, 0\}$ para os complexos. Em seguida foi

calculado o número de elementos em cada matriz, sendo *sizeofa = m * m*, *sizeofb = n * n* e *sizeofc = k * k*. Após, foi a execução da função de multiplicação, que possui a equação:

$$C = beta * C + alpha * A * B$$

Por último, foi liberado os endereços de memória alocado pelas matrizes através da chama *free*. A declaração genérica da multiplicação da BLIS segue o Algoritmo 3.

Algorithm 3 Declaração genérica GEMM BLIS

1: **function** ***void*** *bli_?gemm(**trans_t** transa, **trans_t** transb, **dim_t** m, **dim_t** n, **dim_t** k, **<T>** *alpha, **<T>** *A, **inc_t** rsa, **inc_t** csa, **<T>** *B, **inc_t** rsb, **inc_t** csb, **<T>** *beta, **<T>** *C, **inc_t** rsc, **inc_t** csc*)
2: **end function**

Em que *transa* e *transb* indicam se as matrizes *A* e *B* serão utilizadas conforme foram iniciadas ou se serão implicitamente transpostas ou conjugadas; *m*, *n* e *k* representam os tamanhos das matrizes *A*, *B* e *C* respectivamente; *alpha* representa o escalar a ser multiplicado pelo resultado da multiplicação entre a matriz *A* e *B*, e *beta* representa o escalar que será multiplicado pela matriz *C*; *rsa*, *rsb* e *rsc* representam o incremento para percorrer as linhas das matrizes *A*, *B* e *C* respectivamente e *csa*, *csb* e *csc* representam o incremento para percorrer as colunas das matrizes *A*, *B* e *C* respectivamente. O Algoritmo 4 é um exemplo para a execução da `DGEMM` da BLIS.

3.1.2 Implementação com OpenBLAS

O algoritmo proposto para a OpenBLAS usou-se os tipos primitivos do C. *Float* para representar um dado *real*, *double* para *double precision*, *int complex* para um dado *complex* e *double complex* para representar *double complex*. Para os tamanhos *m*, *n* e *k* das matrizes utilizou-se o tipo *int*. Semelhante à BLIS, iniciou-se com as declarações da variáveis e seus respectivos tipos, após foi a alocação de memória para conter às matrizes *A*, *B* e *C*, através da chamada *malloc*. Em seguida foi calculado o número de elementos em cada matriz, sendo *sizeofa = m * m*, *sizeofb = n * n* e *sizeofc = k * k*. Em sequência iniciou-se as matrizes *A* e *B* com valor 1, e a matriz *C* com valor 0, através de laços de iteração percorrendo as matrizes. Após, foi a execução da função de multiplicação, que possui a equação:

$$C = beta * C + alpha * A * B$$

Algorithm 4 Execução `DGEMM` BLIS

```
1: #include <stdio.h>
2: #include "blis.h"
3: function int main(int argc, char **argv)
4:     dim_t m = 1000;
5:     dim_t n = 1000;
6:     dim_t k = 1000;
7:     inc_t rsa = m;
8:     inc_t csa = 1;
9:     inc_t rsb = n;
10:    inc_t csb = 1;
11:    inc_t rsc = k;
12:    inc_t csc = 1;
13:    dim_t sizeofa = m * m;
14:    dim_t sizeofb = n * n;
15:    dim_t sizeofc = k * k;
16:    double alpha = 1.0;
17:    double beta = 1.0;
18:    double* A = (double*)malloc(sizeof(double) * sizeofa);
19:    double* B = (double*)malloc(sizeof(double) * sizeofb);
20:    double* C = (double*)malloc(sizeof(double) * sizeofc);
21:    for (int i=0; i<sizeofa; i++) do
22:        A[i] = 1;
23:    end for
24:    for (int i=0; i<sizeofb; i++) do
25:        B[i] = 1;
26:    end for
27:    for (int i=0; i<sizeofc; i++) do
28:        C[i] = 0;
29:    end for
30:    bli_dgemm(BLIS_NO_TRANSPOSE, BLIS_NO_TRANSPOSE, m, n, k, &alpha, A, rsa, csa, B, rsb,
       csb, &beta, C, rsc, csc);
31:    free(A);
32:    free(B);
33:    free(C);
34:    return 0;
35: end function
```

Por último, foi liberado os endereços de memória alocado pelas matrizes através da chama *free*. A declaração genérica da multiplicação da OpenBLAS segue o Algoritmo 5.

Algorithm 5 Declaração genérica GEMM OpenBLAS

```
1: function void cblas_?gemm(CBLAS_ORDER order, CBLAS_TRANSPOSE transa,
   CBLAS_TRANSPOSE transb, int m, int n, int k, <T> alpha, <T> *A, int m, <T> *B, int n,
   <T> beta, <T> *C, int k)
2: end function
```

Em que *order* indica se as matrizes são do tipo linha ou do tipo coluna; *transa* e *transb* indicam se as matrizes A e B serão utilizadas conforme foram iniciadas ou se serão implicitamente transpostas ou conjugadas; *alpha* representa o escalar a ser multiplicado pelo resultado da multiplicação entre a matriz A e B, e *beta* representa o escalar que será multiplicado pela matriz C; m, n e k representam os tamanhos das matrizes A, B e C

respectivamente. O Algoritmo 6 é um exemplo para a execução da `DGEMM` da OpenBLAS.

Algorithm 6 Execução `DGEMM` OpenBLAS

```
1: #include <stdio.h>
2: #include <stdlib.h>
3: #include <cblas.h>
4: function int main(int argc, char **argv)
5:     int m = 1000;
6:     int n = 1000;
7:     int k = 1000;
8:     int sizeofa = m * m;
9:     int sizeofb = n * n;
10:    int sizeofc = k * k;
11:    double alpha = 1.0;
12:    double beta = 1.0;
13:    double* A = (double*)malloc(sizeof(double) * sizeofa);
14:    double* B = (double*)malloc(sizeof(double) * sizeofb);
15:    double* C = (double*)malloc(sizeof(double) * sizeofc);
16:    for (int i=0; i<sizeofa; i++) do
17:        A[i] = 1;
18:    end for
19:    for (int i=0; i<sizeofb; i++) do
20:        B[i] = 1;
21:    end for
22:    for (int i=0; i<sizeofc; i++) do
23:        C[i] = 0;
24:    end for
25:    cblas_dgemm(cblascolmajor, CblasNoTrans, CblasNoTrans, m, n, k, alpha, A, m, B, n, beta, C, k);
26:    free(A);
27:    free(B);
28:    free(C);
29:    return 0;
30: end function
```

3.1.3 Implementação com Intel MKL

O algoritmo proposto para a MKL utilizou o tipo *float* para representar os *reals*, *double* para *double precision*, o tipo interno da MKL *MKL_Complex8* para os *complex* e *MKL_Complex16* para *double complex*. Usou-se o tipo *MKL_INT* para armazenar o tamanho das matrizes. A implementação do algoritmo seguiu o mesmo da BLIS e OpenBLAS. Inicialmente declarou-se as variáveis e os tipos associados, após foi feita a alocação da memória para conter as matrizes, através da chamada *mkl_malloc*. Em seguida foi calculado o número de elementos em cada matriz, sendo $sizeofa = m * m$, $sizeofb = n * n$ e $sizeofc = k * k$. Após iniciou-se as matrizes A e B com valor 1 e a matriz C com valor 0 através de laços de iteração percorrendo as matrizes. Seguindo, foi a execução da função

de multiplicação, que possui a equação:

$$C = beta * C + alpha * A * B$$

Por último, foi liberado os endereços de memória alocado pelas matrizes através da chama *mkl_free*. A declaração genérica da multiplicação da MKL segue o Algoritmo 7.

Algorithm 7 Declaração genérica GEMM Intel MKL

```
1: function void ?gemm(char *transa, char *transb, MKL_INT *m, MKL_INT *n, MKL_INT *k, <T>
   *alpha, <T> *A, MKL_INT m, <T> *B, MKL_INT n, <T> *beta, <T> *C, MKL_INT k)
2: end function
```

Em que *transa* e *transb* indicam se as matrizes *A* e *B* serão utilizadas conforme foram iniciadas ou se serão implicitamente transpostas ou conjugadas; *alpha* representa o escalar a ser multiplicado pelo resultado da multiplicação entre a matriz *A* e *B*, e *beta* representa o escalar que será multiplicado pela matriz *C*; *m*, *n* e *k* representam os tamanhos das matrizes *A*, *B* e *C* respectivamente. O Algoritmo 8 é um exemplo para a execução da `DGEMM` da Intel MKL.

Algorithm 8 Execução `DGEMM` Intel MKL

```
 1: #include <stdio.h>
 2: #include "mkl.h"
 3: function int main(int argc, char **argv)
 4:     MKL_INT m = 1000;
 5:     MKL_INT n = 1000;
 6:     MKL_INT k = 1000;
 7:     MKL_INT sizeofa = m * m;
 8:     MKL_INT sizeofb = n * n;
 9:     MKL_INT sizeofc = k * k;
10:     double alpha = 1.0;
11:     double beta = 1.0;
12:     double* A = (double*)mkl_malloc(sizeof(double) * sizeofa, 64);
13:     double* B = (double*)mkl_malloc(sizeof(double) * sizeofb, 64);
14:     double* C = (double*)mkl_malloc(sizeof(double) * sizeofc, 64);
15:     for (int i=0; i<sizeofa; i++) do
16:         A[i] = 1;
17:     end for
18:     for (int i=0; i<sizeofb; i++) do
19:         B[i] = 1;
20:     end for
21:     for (int i=0; i<sizeofc; i++) do
22:         C[i] = 0;
23:     end for
24:     dgemm("N", "N", &m, &n, &k, &alpha, A, &m, B, &n, &beta, C, &k);
25:     mkl_free(A);
26:     mkl_free(B);
27:     mkl_free(C);
28:     return 0;
29: end function
```

3.2 Ambiente de Execução

O Parque Computacional de Alto Desempenho (PCAD) dispõe de uma infraestrutura computacional utilizada por diferentes grupos de pesquisa em computação. Ele consiste em um conjunto de nós computacionais (servidores)[1]. Para o ambiente de execução, utilizou-se o nó **blaise**, que possui dois processador Intel Xeon E5-2699 v4 Broadwell, totalizando 44 núcleos (22 por CPU) e 88 *threads*. Cada processador tem a sua especificação descrita na Tabela 3.1 e as informações da arquitetura de memória na Tabela 3.2.

Tabela 3.1 – Especificações do processador

Microarquitetura	Broadwell
Núcleos	22
Threads	44
Frequência base	2.20 GHz
Frequência máxima	3.60 GHz
Velocidade barramento	9.6 GT/s
TDP	145 W

Fonte: Intel[2]

Tabela 3.2 – Especificações da memória

Tipo de memória	DDR4
L1 Dados	22x32 KiB - 704 KiB
L1 Instruções	22x32 KiB - 704 KiB
L2	22x256 KiB - 5.5 MiB
L3	22x2.5 MiB - 55 MiB
Largura de banda máxima	85 GB/s

Fonte: WikiChip[3]

As execuções dos algoritmos foram divididas em três grupos de acordo com o número de *threads* utilizadas:

- Grupo A: 1 *thread*, considerando a execução sequencial dos algoritmos;
- Grupo B: 22 *threads*, considerando a execução paralela dos algoritmos;
- Grupo C: 44 *threads*, considerando a execução paralela dos algoritmos;

Para definir o número de *threads*, foram exportadas variáveis de ambiente. Embora as três bibliotecas utilizem a IPP OpenMP, além de definir a variável `OMP_NUM_THREADS`,

[1] https://gppd-hpc.inf.ufrgs.br/

[2] https://www.intel.com/content/www/us/en/products/sku/91317/intel-xeon-processor-e52699-v4-55m-cache-2-20-ghz/specifications.html

[3] https://en.wikichip.org/wiki/intel/xeon_e5/e5-2699_v4

também definiu-se as variáveis de controle de *threads* de cada biblioteca: `BLIS_NUM_THREADS`, `OPENBLAS_NUM_THREADS` e `MKL_NUM_THREADS`. Além disso, também se definiu o valor *cores* para a variável de ambiente `OMP_PLACES` para definir que apenas os núcleos físicos seriam utilizados para a aplicação alvo. Também utilizou-se o valor *true* para definir a variável `OMP_PROC_BIND`, fazendo assim com que as *threads* de execução fiquem ligadas aos núcleos, evitando a migração de uma *thread* para outro núcleo em tempo de execução.

Quanto às matrizes, os três grupos utilizaram matrizes quadradas de tamanho *m=n=k*, tamanho mínimo de *m=n=k=50* e incrementos de 50 no tamanho até totalizar o tamanho máximo, de acordo com o grupo. Para o Grupo A (1 *thread*), o tamanho máximo das matrizes foi *m=n=k=3000*, totalizando o domínio de 60 tamanhos de matrizes. Para o Grupo B (22 *threads*), o tamanho máximo das matrizes foi *m=n=k=5000*, totalizando o domínio de 100 tamanhos de matrizes. Para o Grupo C (44 *threads*), o tamanho máximo das matrizes foi *m=n=k=7000*, totalizando o domínio de 140 tamanhos de matrizes.

Para cada tamanho de matriz de cada grupo, rodou-se a execução 10 vezes, para termos uma média das execuções. Como descrito na Seção 3.1, as funções de multiplicação de matrizes utilizadas foram `SGEMM`, `DGEMM`, `CGEMM` e `ZGEMM`. Ou seja, considerando uma biblioteca, cada função foi executada 600 vezes para o Grupo A, 1000 vezes para o Grupo B e 1400 vezes para o Grupo C. Levando em conta as três bibliotecas e as 4 funções de multiplicação, ao todo foram feitas 36000 execuções.

Para a obtenção dos resultados, optou-se por utilizar a ferramenta "Perf" (LINUX..., 2009). Perf é uma ferramenta de *profiling* para sistemas Linux 2.6+, é baseado na interface **perf_events** exportada em versões mais recentes do *kernel* Linux. A interface expõe contadores de desempenho de *hardware* disponíveis em CPUs mais modernas. Esses contadores podem rastrear eventos a nível de *hardware*, como ciclos da CPU, eventos de memória, instruções executadas e outros. Os eventos monitorados foram:

- *fp_arith_inst_retired.single*: número de operações em ponto flutuante de precisão simples;
- *fp_arith_inst_retired.double*: número de operações em ponto flutuante de precisão dupla;
- *LLC-loads*: número de instruções *load* executadas no último nível da cache;
- *LLC-stores*: número de instruções *store* executadas no último nível da cache;
- *LLC-load-miss*: número de instruções *load* executadas no LLC e em que o dado

não estava na *cache*;

- *LLC-store-miss*: número de instruções *store* executadas no LLC e em que o dado não estava na *cache*;
- *power/energy-pkg*: energia consumida por todos os núcleos;
- *power/energy-ram*: energia consumida pela memória RAM;

Os eventos que monitoram o número de operações em ponto flutuante foram escolhidos pois serão utilizados para calcular o número de instruções de ponto flutuante por segundo (FLOPS) e servirão como métrica de desempenho, isso é, quanto maior o número de FLOPS, melhor o desempenho da biblioteca para determinado tipo de dado. Os eventos envolvendo o LLC foram escolhidos pois conforme (MA et al., 2019), o acesso a memória pode ser o gargalo de desempenho em multiplicação de matrizes e algoritmos como o proposto em (GOTO; GEIJN, 2008) visam amortizar esse atraso. Os eventos de energia foram escolhidos para demonstrar a eficiência de cada biblioteca.

4 RESULTADOS

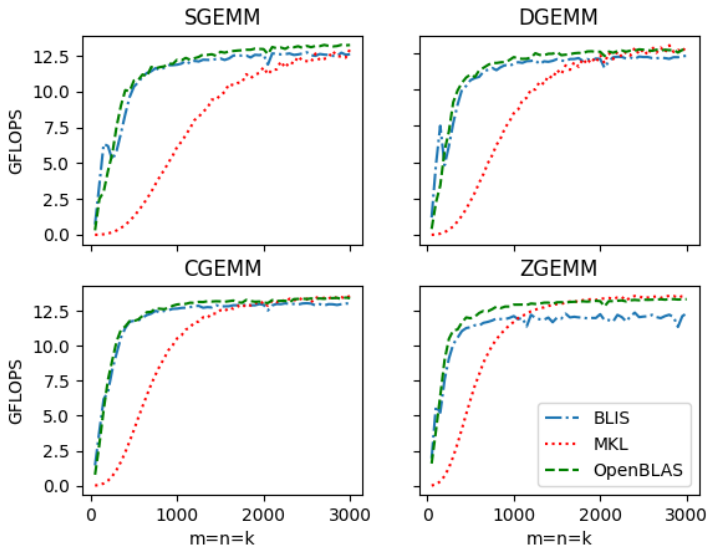
Neste Capítulo os resultados experimentais com relação ao desempenho e consumo de energia da execução das funções `GEMM` para o processador alvo são apresentados e discutidos. Assim, na Seção 4.1 foi analisado o desempenho e a relação dos acessos no LLC com o desempenho, comparando os resultados de cada biblioteca para uma dada função. Por outro lado, na Seção 4.2, foi analisado o consumo de energia realizando um paralelo com o desempenho. Em ambas as Seções, há a divisão em grupos conforme tratado na Metodologia.

4.1 Análise de desempenho

4.1.1 Grupo A

Neste grupo, objetivamos avaliar o desempenho das implementações de cada biblioteca quando executadas com apenas uma *thread*. Assim, a Figura 4.1 destaca o número de giga operações de ponto flutuante por segundo para as funções `SGEMM`, `DGEMM`, `CGEMM` e `ZGEMM`. De modo geral, considerando as quatro funções, OpenBLAS e BLIS foram as bibliotecas que apresentaram melhores resultados para a maior parte do domínio de tamanho das matrizes, com desempenho até quatro vezes superior que a MKL para ma-

Figura 4.1 – Número de operações em ponto flutuante do Grupo A para as funções `SGEMM`, `DGEMM`, `CGEMM` e `ZGEMM`



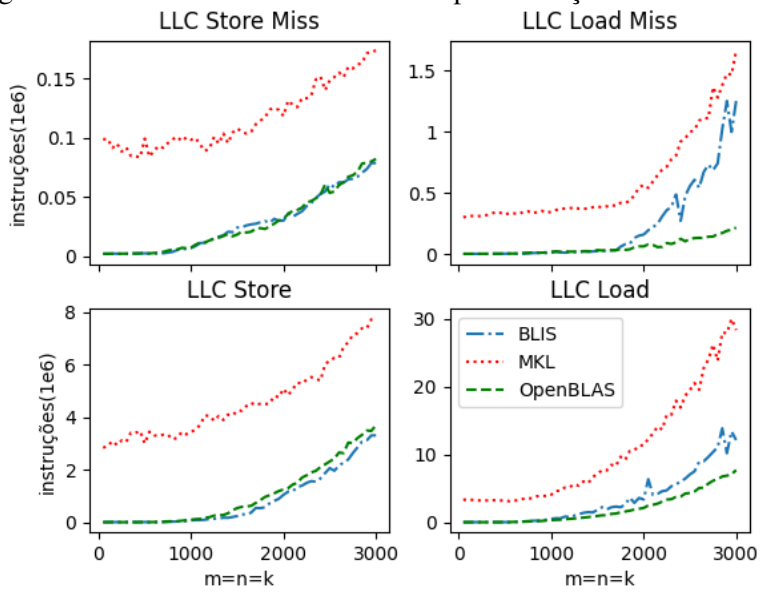
Fonte: o autor

trizes de tamanho *1000x1000*. Já para matrizes de tamanho maior que *2000x2000* as três bibliotecas apresentaram resultados semelhante. Adicionalmente, somente para a função `ZGEMM`, a BLIS foi pior, com desempenho aproximadamente 20% inferior que as outras.

Para as quatro funções, a BLIS e OpenBLAS se mostraram competitivas no desempenho para matrizes de até *2000x2000*, com números semelhantes de acessos no LLC, conforme ilustrado nas Figuras 4.2, 4.4, 4.6 e 4.8. Por outro lado, a MKL apresentou elevado número de acessos no LLC para matrizes de tamanho menor que *1000x1000*, implicando em desempenho inferior às demais bibliotecas. Já para matrizes com tamanho maior que *2000x2000*, a OpenBLAS apresentou menos acessos no LLC enquanto que a BLIS teve uma taxa de *misses* no LLC 30% maior. Por fim, para matrizes maiores que *2000x2000* na `ZGEMM`, a BLIS apresentou um resultado de desempenho de até 20% inferior às demais, pois foi a que apresentou a maior taxa de *misses*.

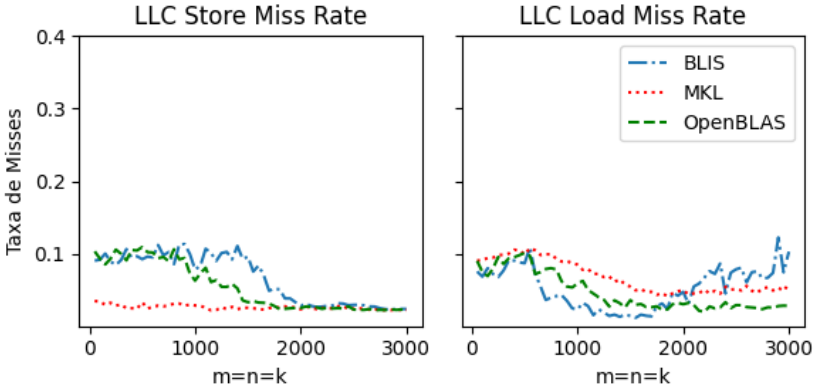
Considerando apenas a função `SGEMM` (Figura 4.2), a MKL foi a biblioteca que obteve maior número de acessos no LLC, tanto para leituras quanto escritas. A BLIS e OpenBLAS apresentaram um baixo número de *misses* na leitura para matrizes menores que *2000x2000*, diferente da MKL. Embora a BLIS e OpenBLAS tenham apresentado resultados semelhantes em operações de escrita, a BLIS apresentou um número de leituras *20%* superior para tamanho de matrizes maiores que *2000x2000*. Já para o número de *misses* de operações de leitura, a MKL apresentou um número quatro vezes maior e a BLIS duas vezes maior, quando comparadas à OpenBLAS, para matrizes maiores que *2000x2000*.

Figura 4.2 – Número de acessos no LLC para a função `SGEMM` do Grupo A



Fonte: o autor

Figura 4.3 – Taxa de *misses* de acessos no LLC para a função `SGEMM` do Grupo A

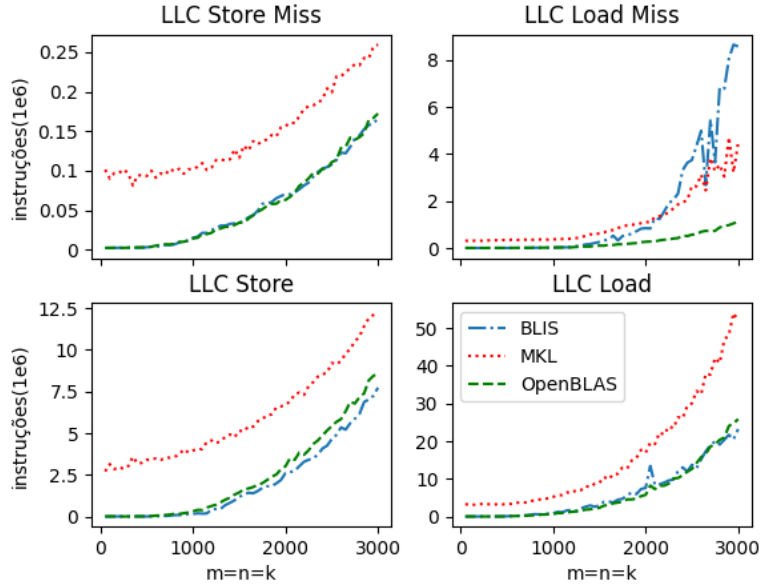


Fonte: o autor

Ao considerar a taxa de misses de operações de escrita e leitura (Figura 4.3), para matrizes de tamanho menor que *1500x1500*, a BLIS apresentou uma baixa taxa de *miss* de operações de leitura mostrando um desempenho competitivo em relação à OpenBLAS. No entanto, conforme o tamanho da matriz aumenta, essa taxa também aumenta, em que para matrizes maiores que *2000x2000*, a BLIS apresentou até 10% mais *miss* na leitura. Adicionalmente, embora a BLIS tenha apresentado maior taxa de *miss* para matrizes de tamanho maior que *2000x2000*, o desempenho foi semelhante às demais pois o número total de acessos no LLC foi similar ao da OpenBLAS e até três vezes menor que a MKL.

Considerando os resultados de operações no LLC da função `DGEMM` (Figura 4.4), a BLIS e OpenBLAS atingiram resultados semelhantes ao longo de todo domínio de tamanho de matrizes. Porém, para o número de *miss* na leitura, a BLIS apresentou um valor

Figura 4.4 – Número de acessos no LLC para a função `DGEMM` do Grupo A

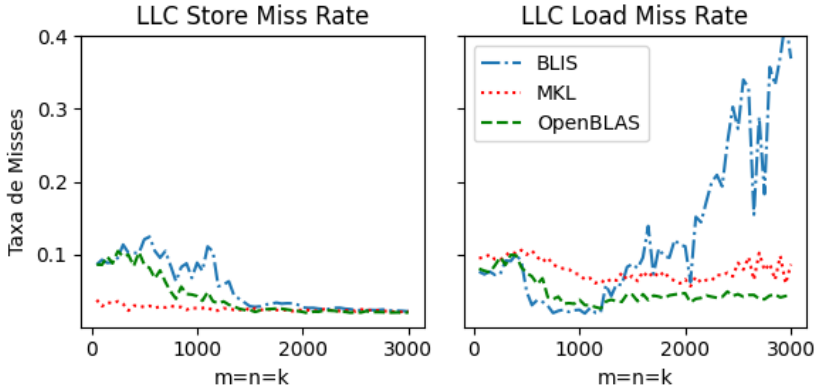


Fonte: o autor

até quatro vezes maior comparado à OpenBLAS para matrizes maiores que *2000x2000*. A MKL possuiu o maior número de acessos no LLC, com até duas vezes mais instruções de escrita e leitura se comparado às outras. Para o número de *miss* na leitura, a BLIS atingiu um resultado maior que às outras, com até aproximadamente oito vezes mais que a OpenBLAS e duas vezes mais que a MKL para matrizes com tamanho maior que *2000x2000*.

Ao avaliar a taxa de *misses* no LLC durante a execução da função `DGEMM` (Figura 4.5), a BLIS demonstrou uma menor taxa em leituras para matrizes menores que *1000x1000*. Porém, conforme o tamanho das matrizes aumentou para valores maiores que *2500x2500*, a BLIS apresentou até *30%* mais *misses* que a MKL e até *40%* mais que a OpenBLAS. Por fim, em matrizes de tamanho maior que *2000x2000*, as três bibliotecas atingiram resultados semelhantes de desempenho.

Figura 4.5 – Taxa de *misses* de acessos no LLC para a função `DGEMM` do Grupo A

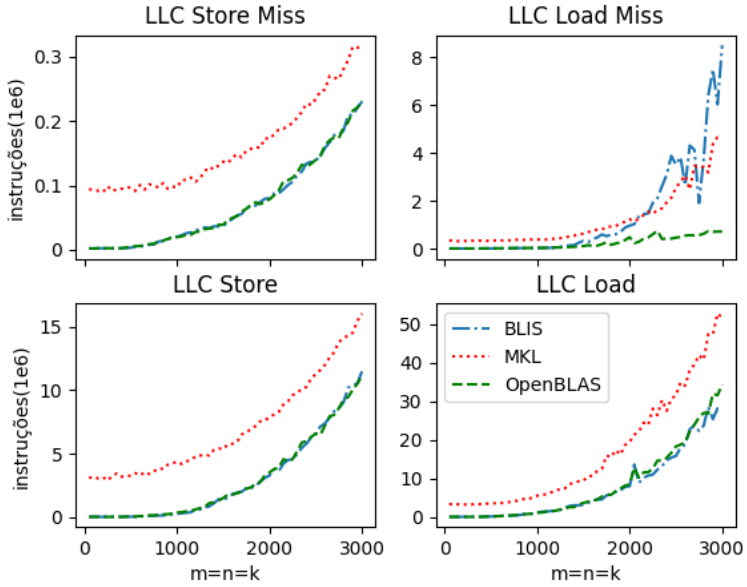


Fonte: o autor

A Figura 4.6 apresenta o número de operações no LLC da função `CGEMM`. Assim como para as demais funções vistas anteriormente, a MKL foi a que mais executou operações no LLC, com aproximadamente $4e6$ e $25e6$ mais operações de escrita e leitura, respectivamente, para matrizes de tamanho maior que *2500x2500*. A BLIS, no entanto, apresentou até aproximadamente oito vezes mais misses de leitura que a OpenBLAS e até duas vezes mais que a MKL para matrizes de tamanho *3000x3000*.

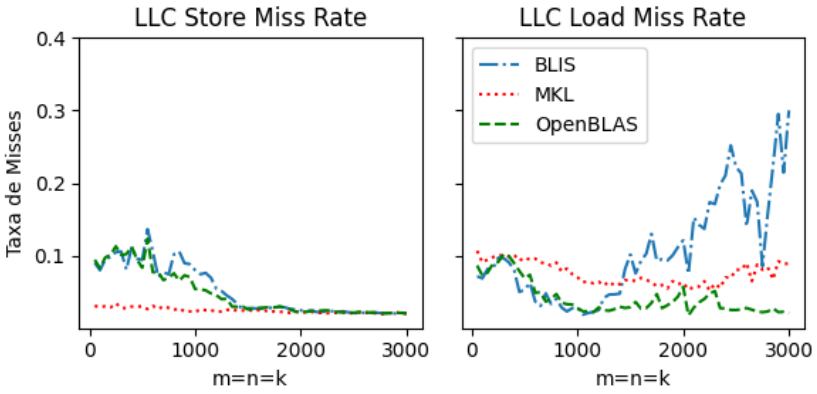
Ao considerar a taxa de *misses* de operações de escrita e leitura (Figura 4.7), as três bibliotecas apresentaram resultados semelhantes para matrizes maiores que *1500x1500*, enquanto que, para matrizes menores, a MKL possui aproximadamente uma taxa de *5%* menos *miss*. Para a taxa de *miss* na escrita, a BLIS apresentou até oito vezes e duas vezes mais *miss* que a OpenBLAS e MKL respectivamente. Mesmo a BLIS apresentando um total de instruções no LLC semelhante à OpenBLAS e inferior a MKL, a sua taxa de *miss* foi maior para matrizes maiores que *2000x2000*. É a partir desse tamanho de matrizes que o desempenho da BLIS diminui em relação às demais, conforme a Figura 4.1.

Figura 4.6 – Número de acessos no LLC para a função `CGEMM` do Grupo A



Fonte: o autor

Figura 4.7 – Taxa de *misses* de acessos no LLC para a função `CGEMM` do Grupo A

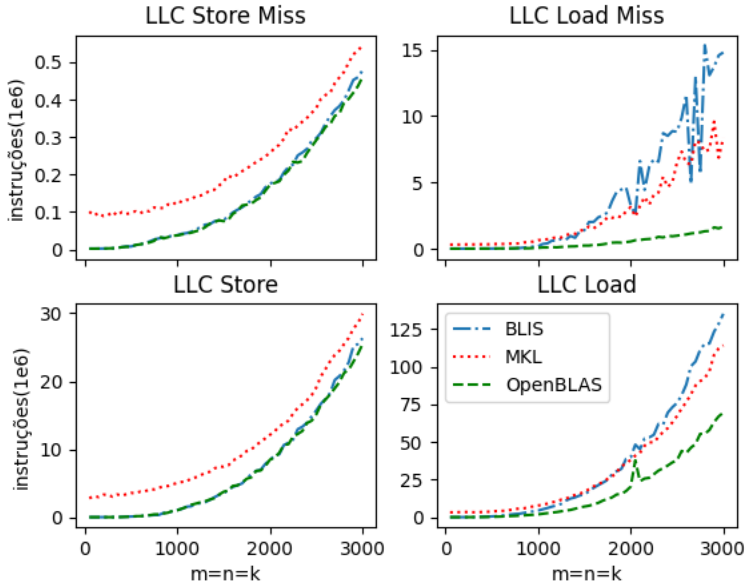


Fonte: o autor

A Figura 4.8 apresenta o número de operações no LLC da função `ZGEMM`. Para acessos de escrita no LLC, a MKL apresentou aproximadamente $2e6$ mais acessos ao longo de todo o domínio de tamanho de matrizes comparado às demais bibliotecas. Em acessos para leitura, para matrizes de tamanho maior que *2000x2000*, a BLIS apresentou até 1.2 vezes mais acessos em relação à MKL e 2 vezes mais em relação a OpenBLAS.

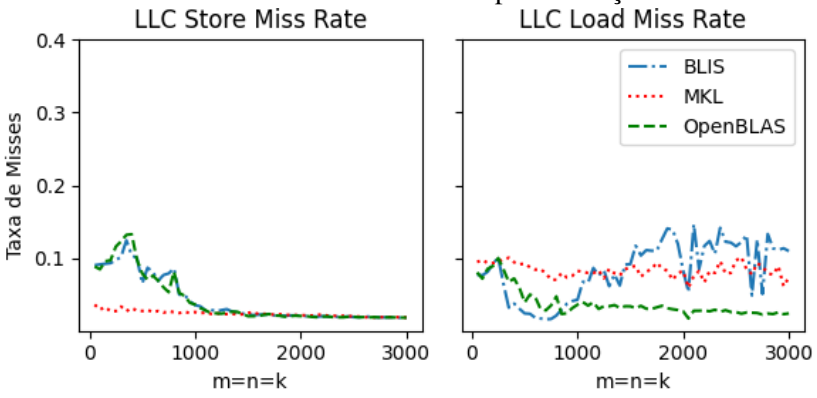
Ao considerar a taxa de *misses* de escrita e leitura (Figura 4.9), as três bibliotecas apresentaram resultados semelhantes para a taxa de *miss* na escrita para matrizes maiores que *1200x1200*, e para tamanho de matrizes menores, a MKL possuiu uma taxa de *5%* menos em relação às demais. Para a taxa de *miss* na leitura, para matrizes de até *1000x1000,* a MKL apresentou uma taxa de até *10%* maior quando comparada às demais, porém, após esse ponto, a BLIS ultrapassou as outras bibliotecas, com uma taxa de até *5%* a mais que a MKL e *10%* a mais que a OpenBLAS.

Figura 4.8 – Número de acessos no LLC para a função ZGEMM do Grupo A



Fonte: o autor

Figura 4.9 – Taxa de *misses* de acessos no LLC para a função ZGEMM do Grupo A



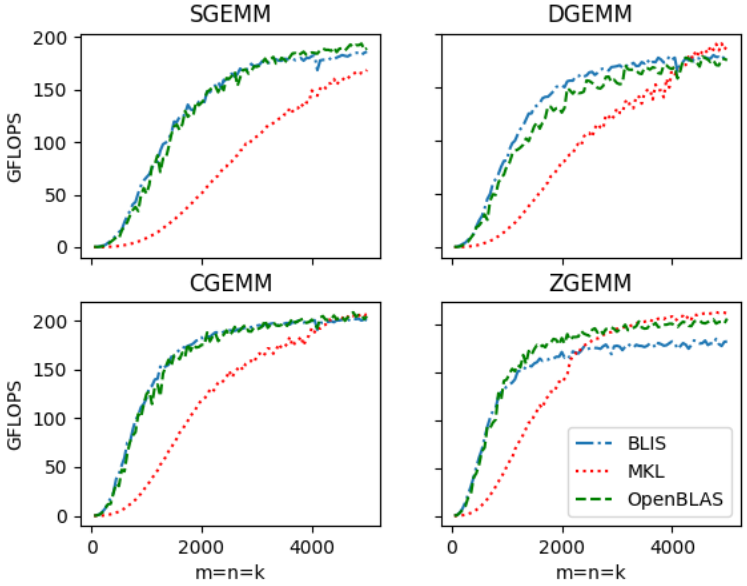
Fonte: o autor

Em resumo, a MKL apresentou o pior desempenho para matrizes de tamanho menor que *1000x1000*. Porém, para matrizes maiores, o desempenho se torna competitivo com as demais, superando a BLIS, que apresentou o pior resultado em matrizes maiores que *2000x2000*. O desempenho inferior da BLIS para matrizes maiores está relacionado à sua taxa de *miss* em acessos para leituras no LLC. Por fim, mesmo a MKL possuindo um maior número de acessos no LLC, exceto para a ZGEMM, esse valor elevado aparenta ser compensado pela sua taxa de *miss*.

4.1.2 Grupo B

Neste grupo, objetivamos avaliar o desempenho das implementações de cada biblioteca quando executadas com 22 *threads*. Assim, a Figura 4.10 destaca o número de giga operações de ponto flutuante por segundo para as funções `SGEMM`, `DGEMM`, `CGEMM` e `ZGEMM` levando em conta todos os núcleos de um nodo NUMA da máquina **blaise**. De modo geral, considerando as quatro funções, as bibliotecas BLIS e OpenBLAS apresentaram melhores resultados para grande parte do domínio de tamanho das matrizes, com desempenho até quatro vezes superior que a MKL para matrizes de tamanho *1000x1000*. Adicionalmente, em matrizes de tamanho *5000x5000*, a MKL superou o desempenho nas funções `DGEMM` e `ZGEMM`, desempenhando *5%* a mais que as demais bibliotecas na `DGEMM` e até *25%* a mais que a BLIS na `ZGEMM`, enquanto na `SGEMM` apresentou um desempenho *20%* menor que as outras. Para a `CGEMM`, em matrizes de tamanho maior que *4000x4000* as três bibliotecas apresentaram resultados semelhantes.

Figura 4.10 – Número de operações em ponto flutuante do Grupo B para as funções `SGEMM`, `DGEMM`, `CGEMM` e `ZGEMM`

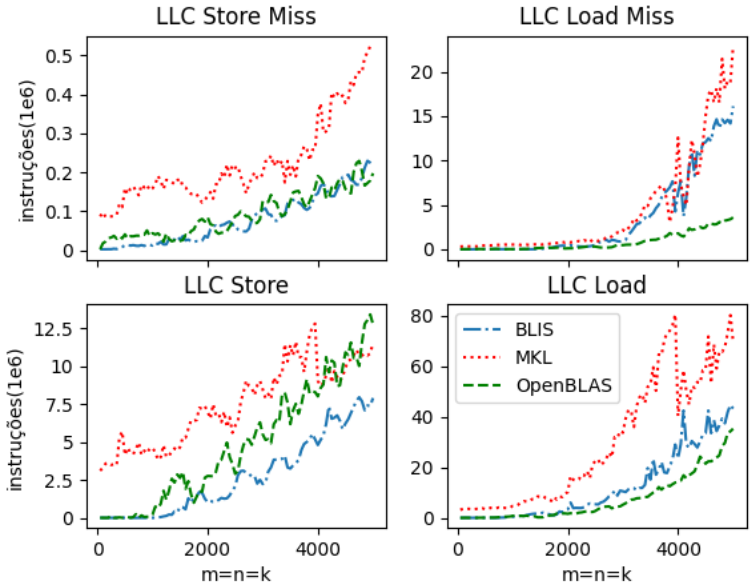


Fonte: o autor

Para as quatro funções, a BLIS e OpenBLAS apresentaram competitividade no desempenho para matrizes de até *2000x2000*, em que ambas atingiram números não muito distintos de acessos no LLC, conforme ilustrado nas Figuras 4.11, 4.13, 4.15 e 4.17. O desempenho inferior da MKL para matrizes menores que *2000x2000* se deve ao superior número de acessos no LLC quando comparado às demais. Para a função `SGEMM` em matrizes de tamanho *3000x3000* a BLIS e OpenBLAS apresentaram um desempenho

duas vezes maior. Para as funções DGEMM e CGEMM, a BLIS apresentou o menor número de acessos no LLC para matrizes maiores que *2000x2000*. A MKL na DGEMM, apresentou o dobro de acessos no LLC para leitura, porém, é a que possuiu a menor taxa de *misses* em acessos no LLC para leitura. Já para a execução da função ZGEMM com matrizes de tamanho *5000x5000*, a OpenBLAS realizou até quatro vezes menos acessos no LLC para leitura que as demais.

Figura 4.11 – Número de acessos no LLC para a função SGEMM do Grupo B



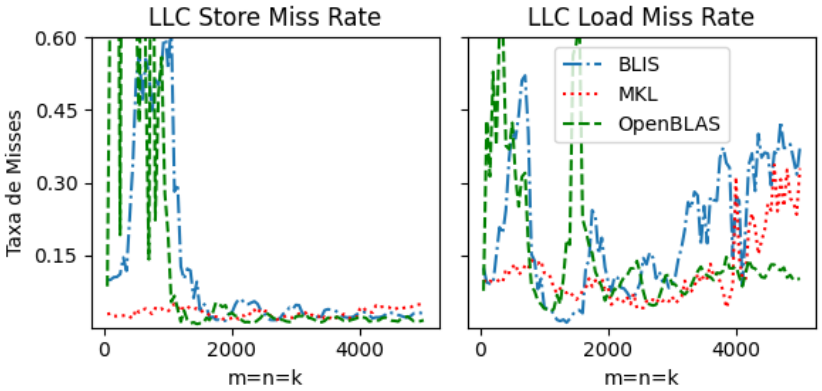
Fonte: o autor

Considerando apenas a função SGEMM, a Figura 4.11 contém o número de acessos de operações no LLC. A MKL além de possuir um maior número de acessos no LLC que as demais, também apresentou um maior número de *misses*, em que para escritas, possui até duas vezes mais *misses* que as demais e até quatro vezes mais *misses* em leitura, se comparada à OpenBLAS. Em acessos no LLC para escrita, a BLIS possui até duas vezes menos acessos que às demais para matrizes maiores que *3000x3000*, no entanto, possui mais acessos no LLC para leituras que a OpenBLAS.

A Figura 4.12 considera a taxa de *misses* de operações de escrita e leitura da função SGEMM, a baixa taxa em *misses* na escrita e taxa inferior de até *30%* em *misses* na leitura da OpenBLAS para matrizes maiores que *3000x3000*, mostram o desempenho superior às demais.

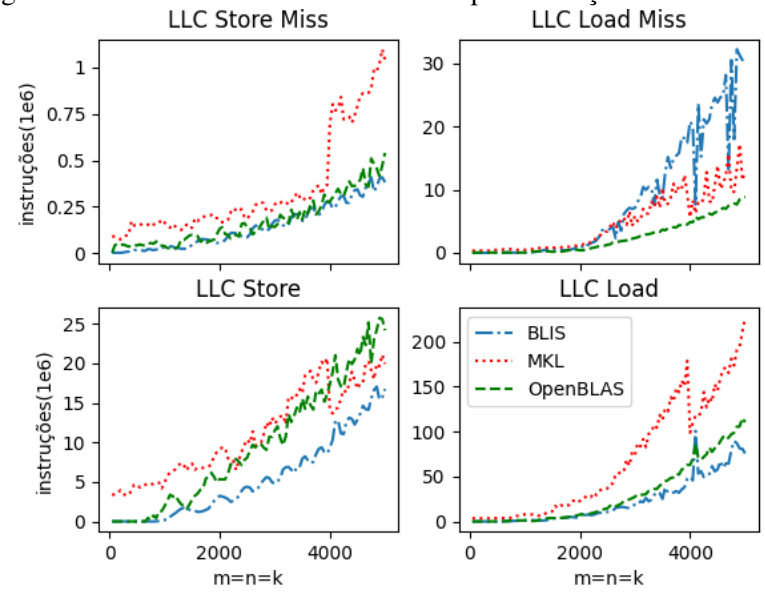
Considerando os resultados de operações no LLC da função DGEMM (Figura 4.13), a BLIS apresentou o menor número de acessos no LLC que as demais ao longo de todo o domínio de tamanho de matrizes. Porém para matrizes maiores que *2000x2000*, ela obteve até três vezes mais *misses* em acessos para leitura, sendo que após esse ponto a

Figura 4.12 – Taxa de *misses* de acessos no LLC para a função `SGEMM` do Grupo B



Fonte: o autor

Figura 4.13 – Número de acessos no LLC para a função `DGEMM` do Grupo B

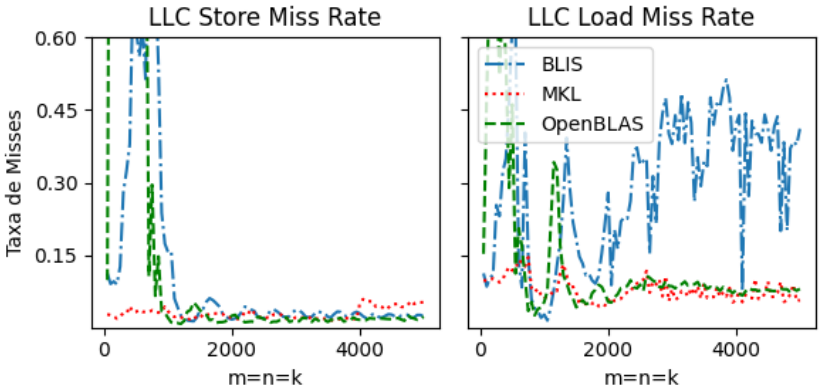


Fonte: o autor

OpenBLAS alcançou o desempenho da BLIS. A MKL apresentou até duas vezes mais *misses* em acessos para escrita e até duas vezes menos *misses* em acessos para leitura (se comparada à BLIS) em matrizes maiores que *4000x4000*, dimensões em que o desempenho da MKL superou o das demais.

Considerando os resultados de operações no LLC da função `CGEMM` (Figura 4.15), a MKL possuiu um maior número de acessos no LLC para escrita até matrizes de tamanho *4000x4000*, ponto em que a OpenBLAS a ultrapassa. Em acessos no LLC para leitura, a MKL apresentou maior número ao longo de todo domínio de tamanho de matrizes e a BLIS apresenta o menor número, com até *25%* menos acessos que as demais para matrizes de tamanho *5000x5000*. Para matrizes maiores que *4000x4000*, em que o desempenho das três bibliotecas se aproximam da igualdade, embora a BLIS apresente o menor número de acessos no LLC para leitura, apresenta um número maior de *misses* para essa operação,

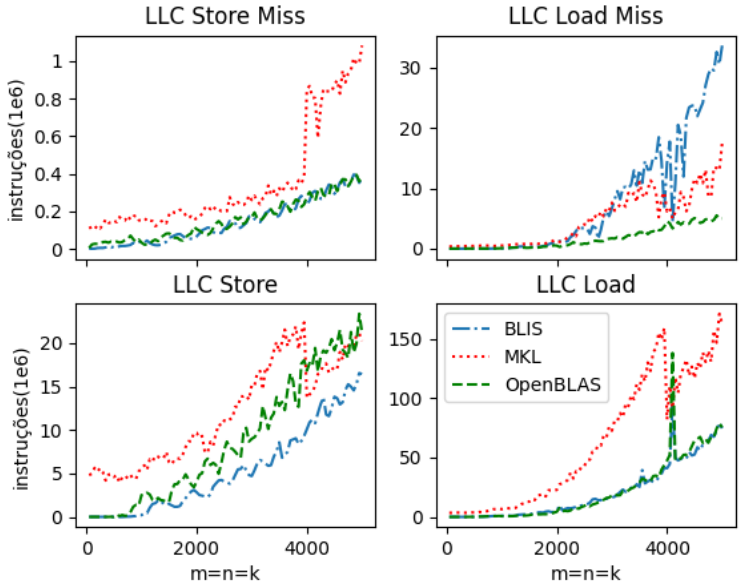
Figura 4.14 – Taxa de *misses* de acessos no LLC para a função `DGEMM` do Grupo B



Fonte: o autor

com até três vezes mais comparada à OpenBLAS e até duas vezes à MKL.

Figura 4.15 – Número de acessos no LLC para a função `CGEMM` do Grupo B

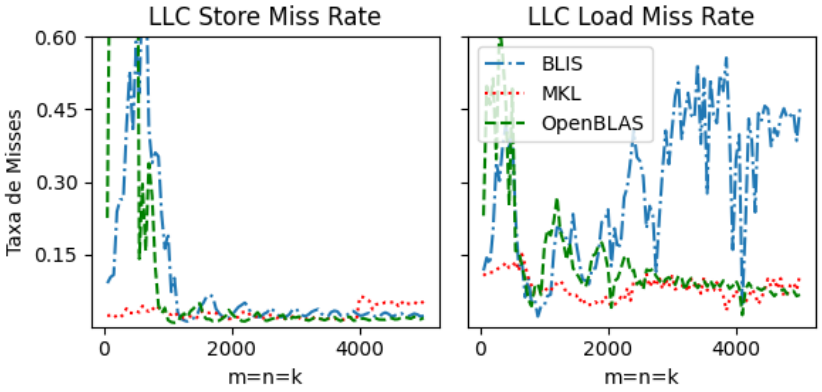


Fonte: o autor

Considerando os resultados de operações no LLC da função `ZGEMM` (Figura 4.17), a BLIS, em que para matrizes maiores que *2000x2000* apresentou o pior desempenho entre as três, embora possua um número de acessos no LLC para escrita semelhante à OpenBLAS e *25%* menor se comparado à MKL, apresentou um alto número de acessos no LLC para leitura, com até quatro vezes mais que a OpenBLAS. Somado a isso, a BLIS também apresentou um alto número de *misses* em acessos no LLC para leitura, com até três vezes mais que a MKL e até seis vezes mais que a OpenBLAS.

Ao avaliar a taxa de miss no LLC durante a execução da função `ZGEMM` (Figura 4.18)), é possível verificar que a taxa de *misses* da BLIS é superior às demais para matrizes maiores que *2000x2000*. A MKL, em que para matrizes maiores que *3000x3000*

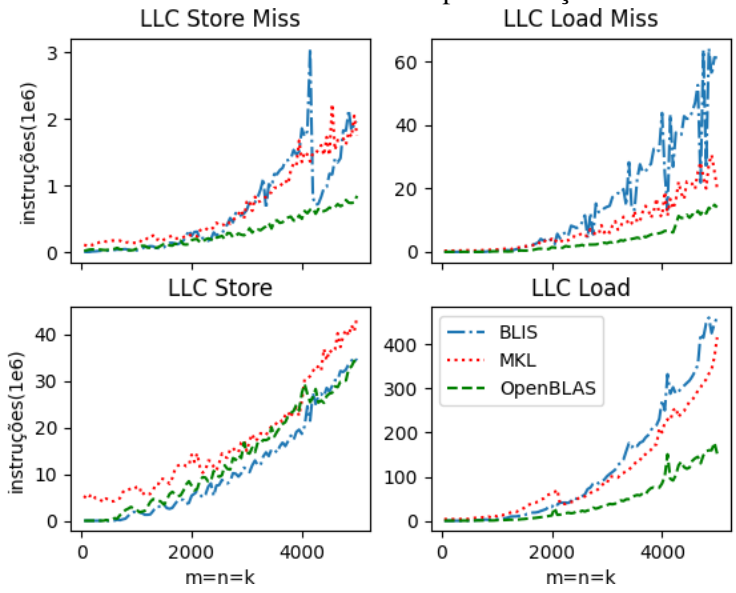
Figura 4.16 – Taxa de *misses* de acessos no LLC para a função `CGEMM` do Grupo B



Fonte: o autor

apresentou o melhor desempenho, possui uma taxa de *misses* em acessos no LLC para leitura semelhante à OpenBLAS, onde em matrizes de tamanho *5000x5000* apresentou uma taxa de aproximadamente *5%* menor em *misses* na leitura.

Figura 4.17 – Número de acessos no LLC para a função `ZGEMM` do Grupo B

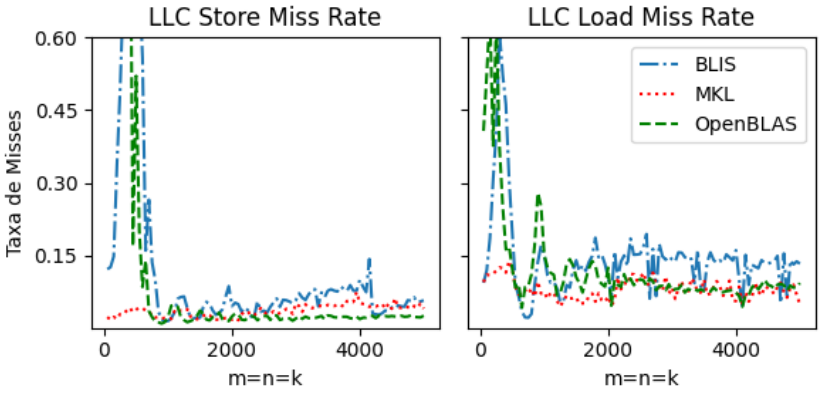


Fonte: o autor

4.1.3 Grupo C

Neste grupo, objetivamos avaliar o desempenho das implementações de cada biblioteca quando executadas com 44 *threads*, utilizando dois nodos NUMA da máquina *blaise*. Assim, a Figura 4.19 destaca o número de giga operações de ponto flutuante por segundo para as funções `SGEMM`, `DGEMM`, `CGEMM` e `ZGEMM`. De modo geral, a MKL apresentou o pior desempenho nas quatro funções para matrizes menores que *3000x3000*,

Figura 4.18 – Taxa de *misses* de acessos no LLC para a função ZGEMM do Grupo B

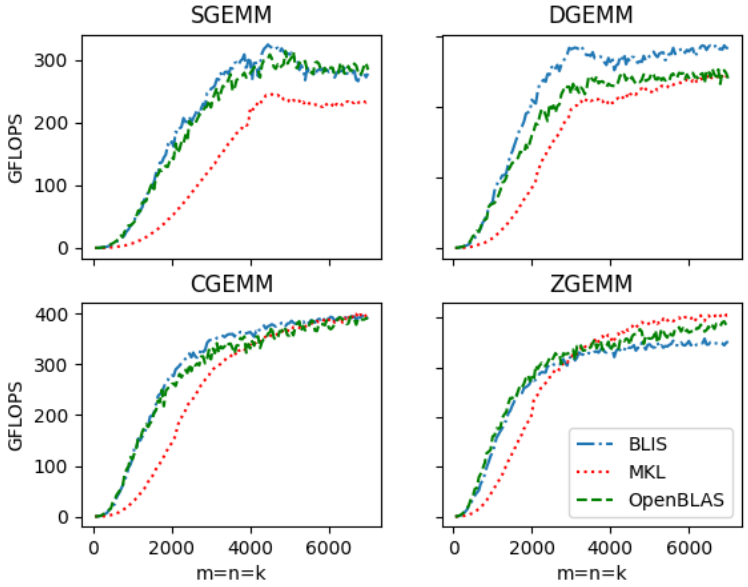


Fonte: o autor

em que na SGEMM para matrizes de tamanho *2000x2000* as outras bibliotecas apresentaram o dobro de desempenho. A BLIS apresentou o melhor desempenho nas funções SGEMM e DGEMM, sendo que na DGEMM para matrizes maiores que *2000x2000* o seu desempenho foi *20%* superior às demais. No entanto, na ZGEMM, para matrizes maiores que *3000x3000* a BLIS apresentou o pior desempenho, com resultado até *20%* inferior das demais. Na CGEMM, a BLIS e OpenBLAS apresentaram desempenhos semelhantes em todo o domínio de tamanho de matrizes, onde a MKL foi inferior para matrizes menores que *3000x3000* e após o seu desempenho equipara às outras bibliotecas.

Embora a MKL apresentou desempenho inferior nas quatro funções para matrizes menores que *3000x3000*, os seus números de acessos no LLC, conforme ilustrado nas Figuras 4.20, 4.22, 4.24 e 4.26, são similares aos das demais bibliotecas.

Figura 4.19 – Número de operações em ponto flutuante do Grupo C para as funções SGEMM, DGEMM, CGEMM e ZGEMM

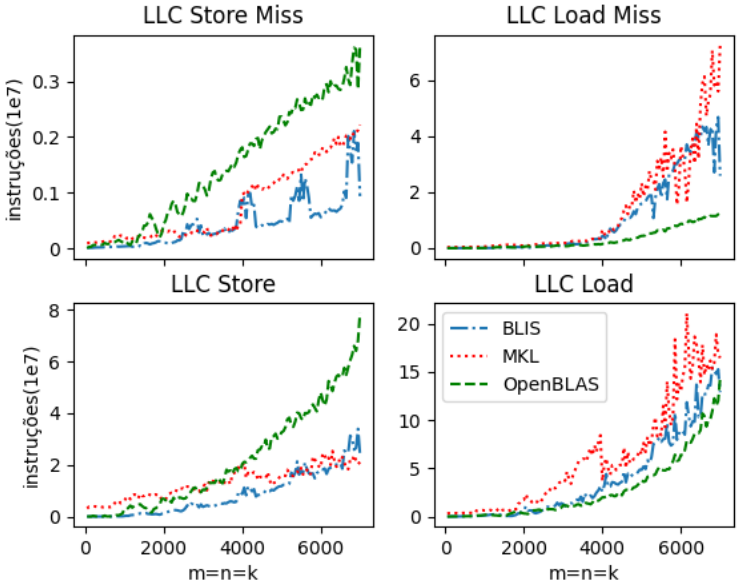


Fonte: o autor

Considerando apenas a função SGEMM, a Figura 4.20 contém o número de acessos de operações no LLC. Nesta operação, a MKL, com o pior desempenho, foi a biblioteca que teve o maior número acessos no LLC para leituras em todo o domínio de tamanho de matrizes e um maior número de acessos no LLC para escritas até matrizes de *3000x3000*, ponto em que a OpenBLAS ultrapassa as demais bibliotecas realizando até quatro vezes mais acessos no LLC para escrita. A BLIS e OpenBLAS com desempenho semelhante, possuem números semelhantes de acessos no LLC, enquanto a OpenBLAS apresentou maiores números para escrita, a BLIS apresentou maiores para leituras, sendo que para matrizes de tamanho *5000x5000* o número de *misses* na escrita da OpenBLAS foi aproximadamente três vezes maior que o da BLIS e para *misses* na leitura três vezes menor.

Ao considerar a taxa de *misses* de operações de escrita e leitura (Figura 4.21) a MKL para matrizes menores que *4000x4000* possui a menor taxa de *misses* na escrita, porém após esse ponto supera às demais com uma taxa de *5%* maior. Para a taxa de *misses* na leitura em matrizes maiores que *4000x4000* a OpenBLAS apresentou valores menores, onde para matrizes de tamanho *5000x5000*, enquanto a OpenBLAS possui uma taxa de *10%*, a BLIS apresenta uma de *30%* e a MKL de *40%*.

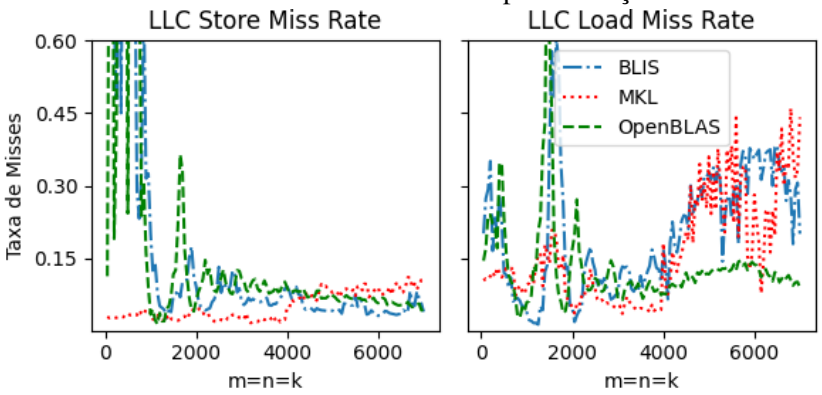
Figura 4.20 – Número de acessos no LLC para a função SGEMM do Grupo C



Fonte: o autor

Considerando os resultados de operações no LLC da função DGEMM (Figura 4.22), o número de acessos no LLC para escrita das três bibliotecas são similares até matrizes de tamanho *2500x2500*, após esse ponto a BLIS e MKL apresentam resultados similares, enquanto a OpenBLAS apresenta maiores, com até quatro vezes mais acessos para escrita que as demais. O desempenho superior da BLIS em relação as demais, se mostra pelo

Figura 4.21 – Taxa de *misses* de acessos no LLC para a função `SGEMM` do Grupo C

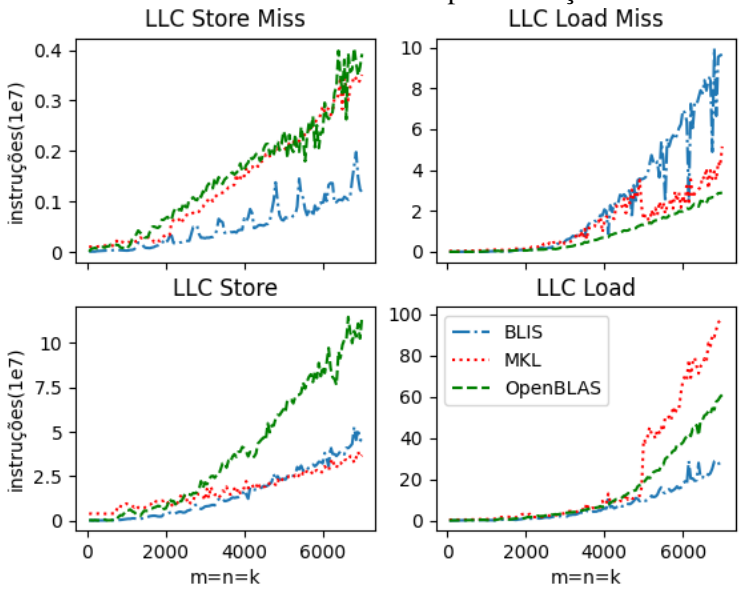


Fonte: o autor

inferior número de acessos no LLC. Para o número de *misses* em acesso no LLC para escrita, a BLIS teve menor número para todo o domínio de matrizes, sendo que esse número foi quatro vezes menor que as demais para matrizes maiores que *6000x6000*. Para acessos no LLC para leitura em matrizes maiores que *4000x4000* a BLIS apresentou duas vezes menos que a OpenBLAS e quatro vezes que a MKL. No entanto, para o número de *misses* em acessos no LLC para leitura, a BLIS apresenta maior número para matrizes maiores que *5000x5000*, com duas vezes mais *misses* que a MKL e quatro vezes mais que OpenBLAS.

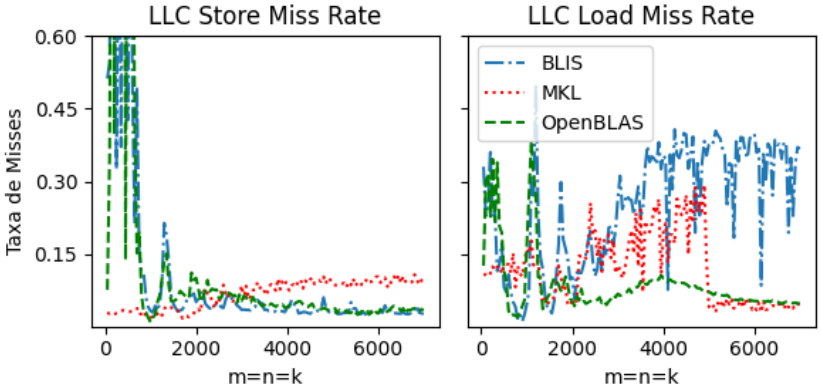
Ao avaliar a taxa de *misses* no LLC, ilustrado na Figura 4.23, para matrizes maiores que *3000x3000* a MKL possui uma taxa maior de *5%* em *misses* na escrita que as demais. Já para *misses* na leitura, a taxa da BLIS foi de até *30%* maior que as demais.

Figura 4.22 – Número de acessos no LLC para a função `DGEMM` do Grupo C



Fonte: o autor

Figura 4.23 – Taxa de *misses* de acessos no LLC para a função DGEMM do Grupo C



Fonte: o autor

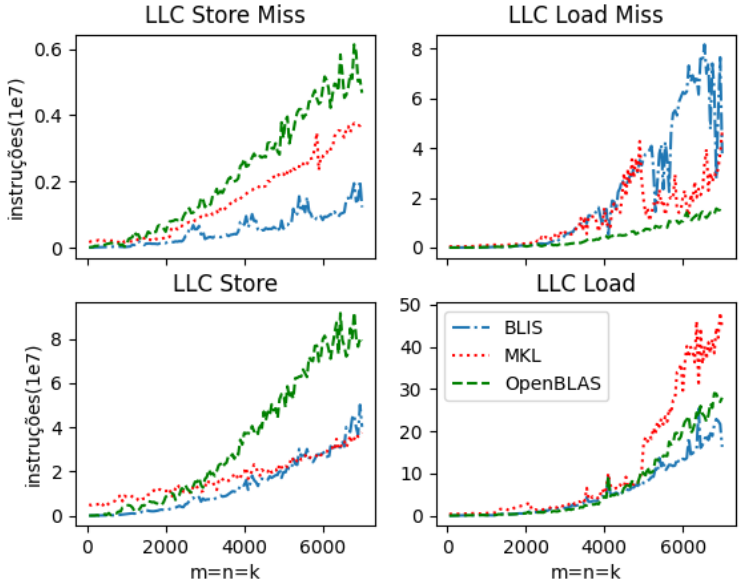
Considerando os resultados de operações no LLC da função CGEMM (Figura 4.24), a MKL com pior desempenho para matrizes menores que *3000x3000* apresentou o maior número de acessos no LLC. Para matrizes maiores que *3000x3000*, em que o desempenho das três bibliotecas é semelhante, a OpenBLAS apresentou mais acessos e *misses* no *LLC* em instruções de escrita, com um número duas vezes maior que as demais para matrizes de tamanho *7000x7000*. No entanto, em acessos no LLC para leitura a MKL apresentou o maior valor, em que para matrizes de tamanho *7000x7000*, o seu número é três vezes maior comparado à OpenBLAS e quatro vezes comparado à BLIS.

Ao avaliar a taxa de *misses* no LLC para a função CGEMM (Figura 4.25), para matrizes maiores que *4000x4000* a MKL apresentou um taxa superior de *5%* que a OpenBLAS e *10%* superior que a MKL para *misses* na escrita. Em *misses* na leitura, para matrizes maiores que *5000x5000*, a BLIS apresentou a maior taxa e a OpenBLAS e MKL, com resultados semelhantes as menores. Para matrizes de tamanho *6000x6000* a taxa de *misses* na leitura da BLIS foi de aproximadamente *40%* maior que as demais.

Considerando os resultados de operações no LLC da função DGEMM (Figura 4.22), as três bibliotecas tiveram resultados semelhantes para matrizes menores que *2000x2000*, porém após esse ponto, a BLIS ultrapassa as demais no número de acessos. A BLIS também apresentou maior número de *misses* em acesso no LLC para leitura, em que para matrizes de tamanho *7000x7000* esse número foi cinco vezes maior que o da OpenBLAS e quatro vezes que o da MKL. Embora a OpenBLAS para matrizes maiores que *4000x4000* apresente um menor número de acessos no LLC para leitura, a MKL com o melhor desempenho, apresentou menos acessos no LLC para escrita.

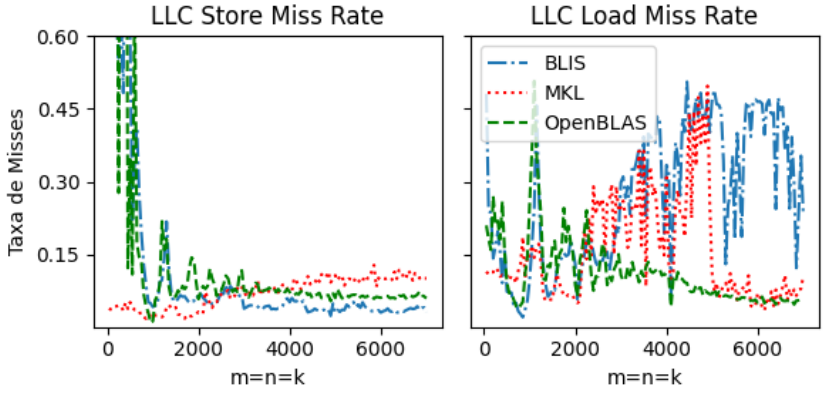
Ao avaliar a taxa de *misses* no LLC durante a execução da ZGEMM (Figura 4.23), para matrizes maiores que *2000x2000* a BLIS apresentou a menor taxa em escritas, com até *5%* menos que a OpenBLAS e *10%* menos que a MKL. Entretanto, em *misses* na

Figura 4.24 – Número de acessos no LLC para a função `CGEMM` do Grupo C



Fonte: o autor

Figura 4.25 – Taxa de *misses* de acessos no LLC para a função `CGEMM` do Grupo C



Fonte: o autor

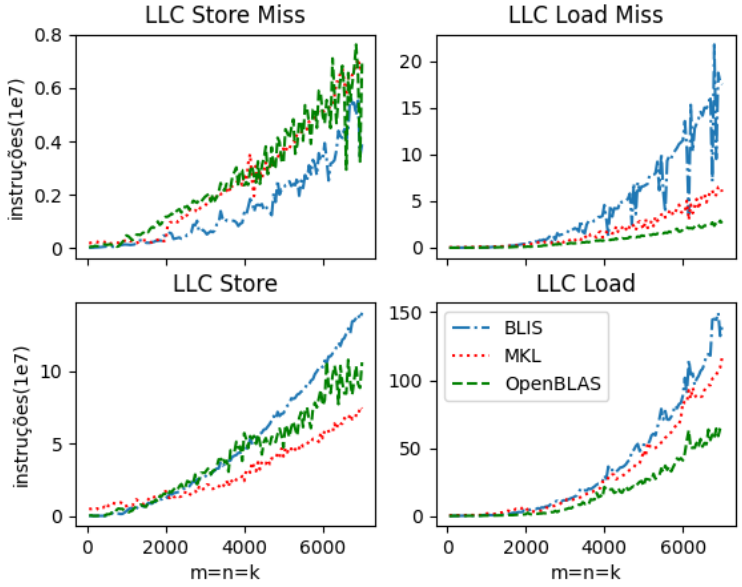
leitura, para matrizes maiores que *3000x3000* a BLIS apresentou a maior taxa, com até *10%* mais *misses* na leitura que as demais. O desempenho inferior da BLIS em relação as demais para matrizes maiores que *3000x3000* se mostram pelo maior número de acessos no LLC e pela maior taxa de *misses* na leitura.

4.2 Análise de consumo de energia

4.2.1 Grupo A

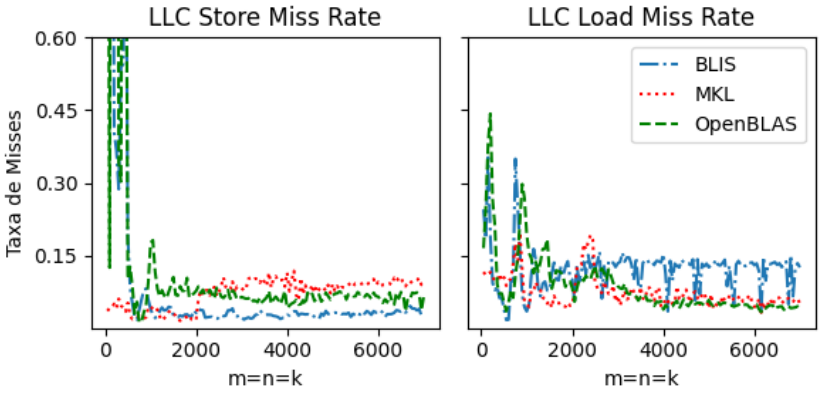
A Figura 4.28 contém o consumo de energia para as quatro funções de multiplicação. Ao considerar as quatro funções, a BLIS e OpenBLAS apresentaram resultados de

Figura 4.26 – Número de acessos no LLC para a função ZGEMM do Grupo C



Fonte: o autor

Figura 4.27 – Taxa de *misses* de acessos no LLC para a função ZGEMM do Grupo C



Fonte: o autor

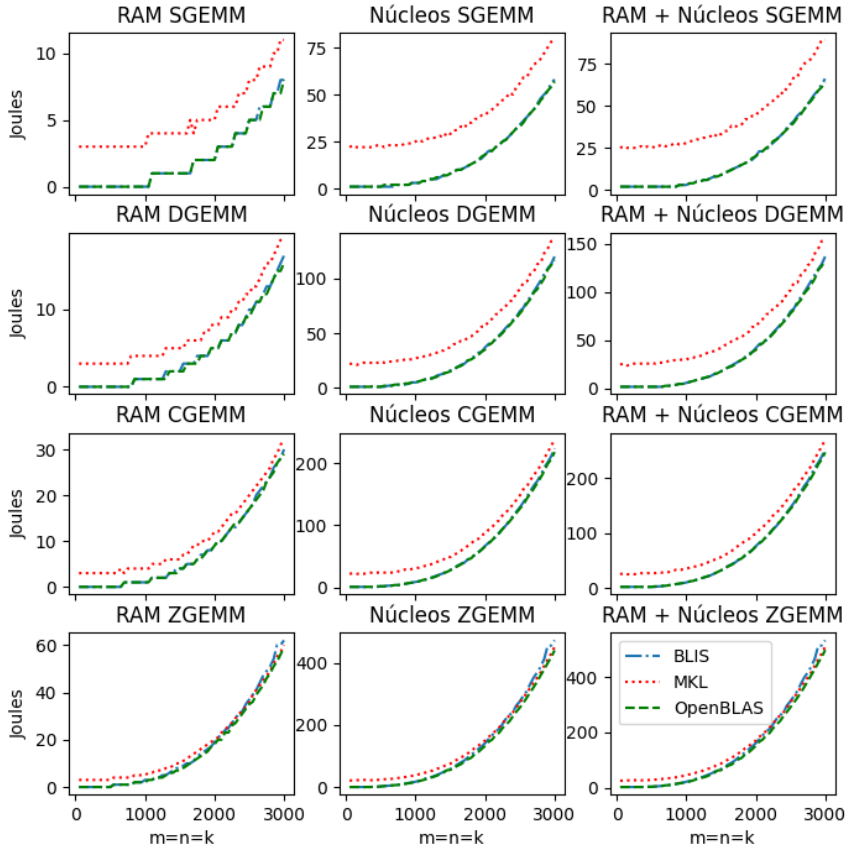
consumo semelhantes e inferiores à MKL para grande parte do domínio de tamanho de matrizes.

Na SGEMM, DGEMM e CGEMM, a energia consumida pela RAM ao executar a MKL foi maior em, aproximadamente, *100%* para matrizes de *1000x1000* e *60%* para matrizes de *3000x3000*, em relação as demais. Enquanto a energia consumida pelos núcleos foi maior em, aproximadamente, *300%* para matrizes de *1000x1000* e *50%* para matrizes de *3000x3000*.

Na ZGEMM, as três bibliotecas possuíram resultados semelhantes. Para matrizes menores que *2000x2000*, a energia da MKL consumida pela RAM é em média *50%* maior do que as demais enquanto pelos núcleos é *100%* maior. Porém para matrizes maiores que 2000x2000 a energia da MKL consumida tanto pela RAM quanto pelos núcleos iguala às demais. Ao fim do domínio dos tamanhos, é possível verificar que a BLIS possui um

consumo levemente maior em comparação às demais.

Figura 4.28 – Consumo de energia Grupo A para as funções SGEMM, DGEMM, CGEMM e ZGEMM



Fonte: o autor

Através dos gráficos de desempenho das funções do Grupo A (Figura 4.1), é verificado que o consumo de energia possui relação com o desempenho das bibliotecas. A MKL com pior desempenho para matrizes menores que *1000x1000* também apresenta maior com consumo. Na ZGEMM, em que a BLIS desempenhou *20%* a menos que as demais para matrizes de *3000x3000* também apresentou um maior consumo total de energia.

4.2.2 Grupo B

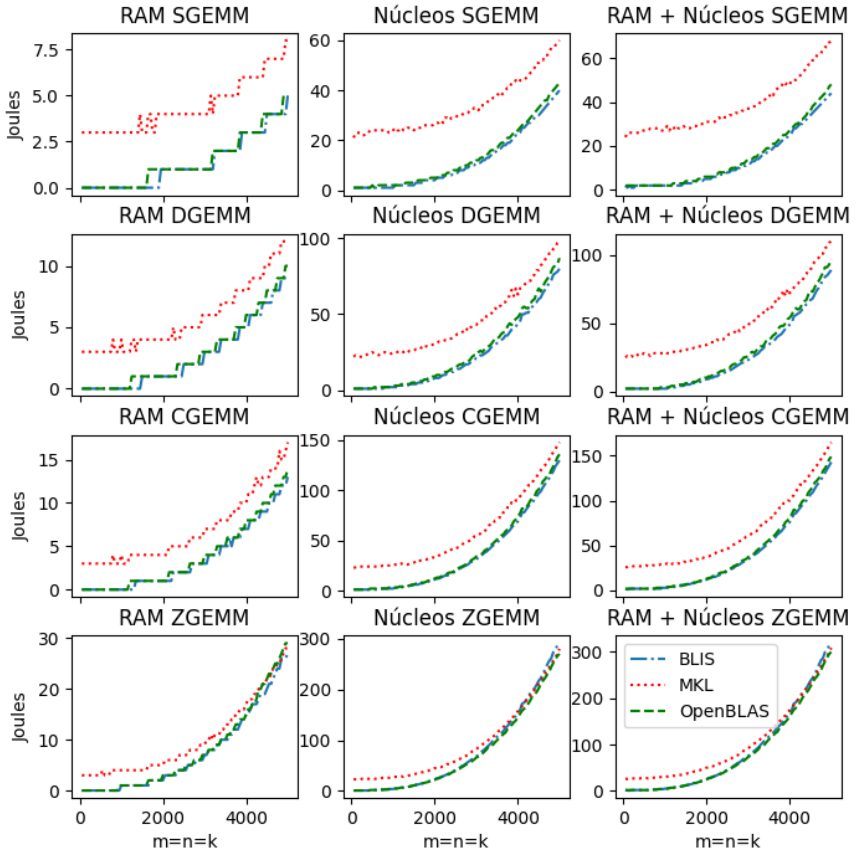
A Figura 4.29 contém o consumo de energia para as quatro funções de multiplicação. Ao considerar as quatro funções, assim como no Grupo A, a BLIS e OpenBLAS apresentaram resultados semelhantes e superiores à MKL para grande parte do domínio de tamanho de matrizes.

Na SGEMM, DGEMM e CGEMM, a energia consumida pela RAM ao executar a MKL quando comparada às demais bibliotecas, em média, foi de aproximadamente *100%* maior, enquanto a energia consumida pelos núcleos foi maior em aproximadamente *200%*,

para matrizes menores que *2000x2000*. Em matrizes de *3000x3000*, a energia total consumida pela MKL foi maior em *100%* na `SGEMM`, *30%* na `DGEMM` e *10%* na `CGEMM`.

Na `ZGEMM`, as três bibliotecas possuem resultados semelhantes. Para matrizes menores que *3000x3000*, a energia da MKL consumida pela RAM e pelos núcleos do processador é, em média, *50%* maior do que as demais. Porém para matrizes maiores que 4000x4000 a energia da MKL consumida tanto pela RAM quanto pelos núcleos iguala às demais. Ao fim do domínio dos tamanhos, é possível verificar que a BLIS possui um consumo levemente maior em comparação às demais.

Figura 4.29 – Consumo de energia Grupo B para as funções `SGEMM`, `DGEMM`, `CGEMM` e `ZGEMM`



Fonte: o autor

Através dos gráficos de desempenho das funções do Grupo B (Figura 4.10), é verificado que o consumo de energia possui relação com o desempenho das bibliotecas, principalmente em matrizes menores que *2000x2000*. Para essas dimensões a MKL, além de apresentar o menor desempenho entre as três, também apresenta o maior consumo total de energia.

4.2.3 Grupo C

A Figura 4.30 contém o consumo de energia do Grupo C para as quatro funções de multiplicação. Ao considerar as quatro funções, assim como nos Grupo A e B, a BLIS e OpenBLAS apresentaram resultados semelhantes e superiores à MKL para grande parte do domínio de tamanho de matrizes.

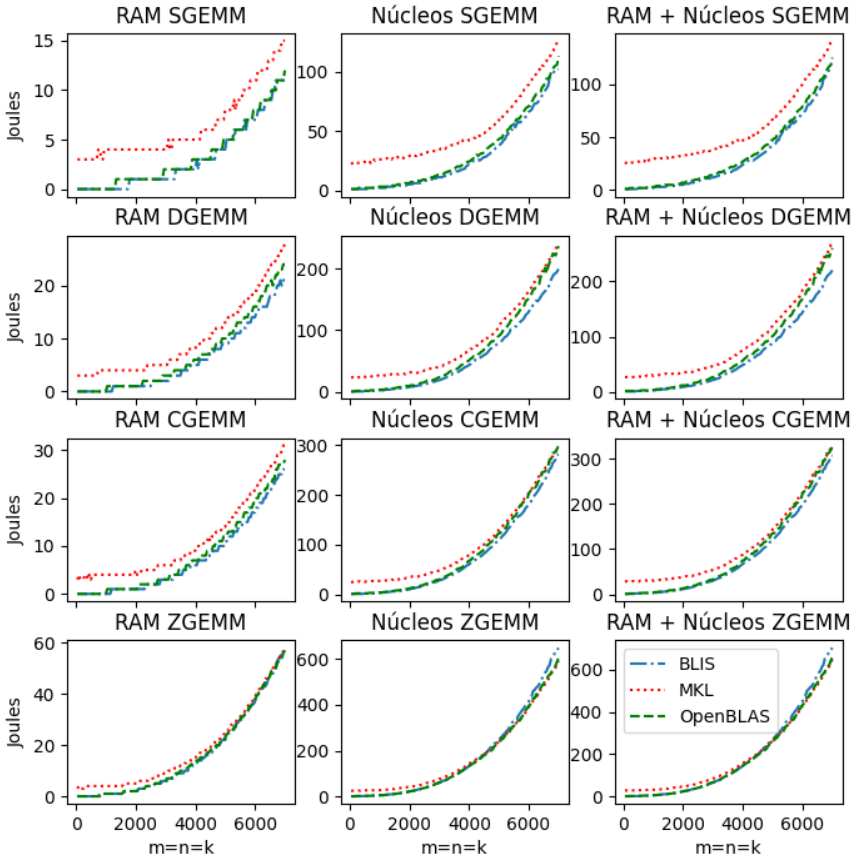
Na `SGEMM`, a BLIS e a OpenBLAS apresentaram consumos semelhantes, tanto pelo processador quanto pela RAM. A MKL apresentou o maior consumo, em que pela RAM e pelos núcleos esse consumo foi maior, em média, de *100%*, para matrizes menores que *4000x4000*. Para matrizes maiores que *4000x4000*, o consumo total da MKL foi, em média, aproximadamente *50%* maior.

Na `DGEMM` e `CGEMM`, a BLIS apresentou o menor consumo, seguido pela OpenBLAS e após pela MKL. Para matrizes de *7000x7000*, o consumo de energia total da BLIS na `DGEMM` foi menor em aproximadamente *15%*. A MKL, para matrizes menores que *4000x4000*, teve um consumo pela RAM e pelos núcleos maior, em média, de aproximadamente *50%*.

Na `ZGEMM`, o consumo pela RAM à execução da MKL foi maior, em média, de *100%* que as demais, até matrizes de *5000x5000* em que o consumo pela RAM das três se equiparou. O consumo pelos núcleos também foi maior à execução da MKL, em média, de *100%*, até matrizes de *5000x5000* em que o consumo da BLIS ultrapassa as demais, em que para matrizes de *7000x7000* a BLIS consumiu *10%* a mais.

Através dos gráficos de desempenho das funções do Grupo C (Figura 4.19), é verificado que o consumo de energia possui relação com o desempenho das bibliotecas. Ao analisar matrizes de tamanho *7000x7000*, para as quatro funções, a biblioteca que apresentou o menor desempenho também apresentou o maior consumo. Na `CGEMM`, em que as três bibliotecas apresentam desempenho semelhante, elas também apresentam consumo total similares.

Figura 4.30 – Consumo de energia Grupo C para as funções SGEMM, DGEMM, CGEMM e ZGEMM



Fonte: o autor

5 DISCUSSÃO E TRABALHOS FUTUROS

Nesse trabalho, avaliamos o desempenho e o consumo de energia de três bibliotecas BLAS (BLIS, OpenBLAS e MKL), ao executarem as funções `SGEMM`, `DGEMM`, `CGEMM` e `ZGEMM` em um processador Intel *multicore* com arquitetura Broadwell, fazendo uso da IPP OpenMP, para diversos conjuntos de entradas. Para tanto, três grupos de execuções foram criados, em que os grupos diferem entre si no número de *threads* criadas para a execução das funções, bem como o domínio de tamanho de matrizes utilizadas.

Através dos resultados, verificamos que a MKL desempenha de forma inferior às demais, independente do número de *threads* utilizadas, para matrizes menores que tamanho *2000x2000*. Porém, a MKL apresentou melhor desempenho na função `ZGEMM` em matrizes maiores que *3000x3000*. A BLIS e OpenBLAS apresentaram resultados muito semelhantes de desempenho e consumo de energia, exceto para a `ZGEMM`, em que a OpenBLAS apresenta um desempenho superior de *20%* para matrizes maiores que *2000x2000*.

A MKL também apresentou maior consumo de energia que as demais bibliotecas, onde nas funções `GEMM` quando executadas em um único *core*, a MKL consumiu, em média, 20 *Joules* a mais que as outras bibliotecas.

Os resultados obtidos podem auxiliar futuros desenvolvedores à optarem pela biblioteca BLAS que melhor se enquadra nos objetivos do projeto, seja na busca por desempenho ou por eficiência energética, tanto para aplicações *singlecore* quanto *multicore*. Somado-se à isso, os resultados aqui obtidos também poder ser utilizados para identificar gargalos de desempenho ou consumo, e então, desenvolver otimizações visando superar esses gargalos. À medida que as cargas de trabalho e os conjuntos de dados aumentam, é de extrema importância a escalabilidade das bibliotecas BLAS.

Para trabalhos futuros, poderíamos coletar mais métricas, para possuir uma explicação mais detalhada dos desempenhos, como o número de acessos à memória RAM e demais níveis da *cache*. Além disso, poderíamos comparar o resultado das bibliotecas em diferentes arquiteturas de computador, e avaliar o comportamento de cada uma.

REFERÊNCIAS

ALMASI, G. S.; GOTTLIEB, A. **Highly Parallel Computing**. USA: Benjamin-Cummings Publishing Co., Inc., 1989. ISBN 0805301771.

AT&T Bell Laboratories. **Netlib**. 1980. <https://netlib.org/>. Acesso em 04/03/2023.

BUTENHOF, D. R. **Programming with POSIX Threads**. USA: Addison-Wesley Longman Publishing Co., Inc., 1997. ISBN 0201633922.

CHAPMAN, B. M.; JOST, G.; PAS, R. van der. Using llc - portable shared memory parallel programming. In: **Scientific and engineering computation**. [S.l.: s.n.], 2007.

CULLER, D.; SINGH, J. P.; GUPTA, A. **Parallel Computer Architecture: A Hardware/Software Approach**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998. ISBN 9780080573076.

DONGARRA, J. et al. (Ed.). **Sourcebook of Parallel Computing**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003. ISBN 1558608710.

DONGARRA, J. J. et al. An extended set of fortran basic linear algebra subprograms. **ACM Trans. Math. Softw.**, Association for Computing Machinery, New York, NY, USA, v. 14, n. 1, p. 1–17, mar 1988. ISSN 0098-3500. Disponível em: <https://doi.org/10.1145/42288.42291>.

DONGARRA, J. J. et al. A set of level 3 basic linear algebra subprograms. **ACM Trans. Math. Softw.**, Association for Computing Machinery, New York, NY, USA, v. 16, n. 1, p. 1–17, mar 1990. ISSN 0098-3500. Disponível em: <https://doi.org/10.1145/77626.79170>.

FIBICH, C. et al. Evaluation of open-source linear algebra libraries targeting arm and risc-v architectures. In: **2020 15th Conference on Computer Science and Information Systems (FedCSIS)**. [S.l.: s.n.], 2020. p. 663–672.

FOSTER, I. **Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering**. USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN 0201575949.

GOTO, K. **GotoBLAS**. 2002. <https://www.tacc.utexas.edu/research-development/tacc-software/gotoblas2>. Acesso em 14/03/2023.

GOTO, K.; GEIJN, R. V. D. High-performance implementation of the level-3 blas. Association for Computing Machinery, New York, NY, USA, v. 35, n. 1, jul 2008. ISSN 0098-3500. Disponível em: <https://doi.org/10.1145/1377603.1377607>.

IEEE Standard for Information Technology–POSIX(TM) Ada Language Interfaces–Part 1: Binding for System Application Program Interface (API)–Amendment 1: Realtime Extensions. **IEEE Std 1003.5b-1996 (Includes IEEE Std 1003.5-1992)**, 1997.

Intel. **Math Kernel Labrary**. 2003. <https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html>. Acesso em 07/08/2023.

JADON, S.; YADAV, R. S. Multicore processor: Internal structure, architecture, issues, challenges, scheduling strategies and performance. In: **2016 11th International Conference on Industrial and Information Systems (ICIIS)**. [S.l.: s.n.], 2016.

LAWSON, C. L. et al. Basic linear algebra subprograms for fortran usage. **ACM Trans. Math. Softw.**, Association for Computing Machinery, New York, NY, USA, v. 5, n. 3, 1979. ISSN 0098-3500. Disponível em: <https://doi.org/10.1145/355841.355847>.

LINUX perf. 2009. <https://perf.wiki.kernel.org/>. Acesso em 10/05/2023.

LORENZON, A. F. et al. Seamless optimization of the gemm kernel for task-based programming models. In: **Proceedings of the 36th ACM International Conference on Supercomputing**. New York, NY, USA: Association for Computing Machinery, 2022. (ICS '22). ISBN 9781450392815. Disponível em: <https://doi.org/10.1145/3524059.3532385>.

MA, S. et al. Coordinated dma: Improving the dram access efficiency for matrix multiplication. **IEEE Transactions on Parallel and Distributed Systems**, v. 30, n. 10, p. 2148–2164, 2019.

NICHOLS, B.; BUTTLAR, D.; FARRELL, J. P. **Pthreads programming - a POSIX standard for better multiprocessing.** [S.l.]: O'Reilly, 1996. ISBN 978-1-56592-115-3.

OpenMP Architecture Review Board. **OpenMP Application Program Interface Version 4.0**. 2013. Disponível em: <https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf>.

PAPAMARCOS, M. S.; PATEL, J. H. A low-overhead coherence solution for multiprocessors with private cache memories. Association for Computing Machinery, New York, NY, USA, v. 12, n. 3, jan 1984. ISSN 0163-5964. Disponível em: <https://doi.org/10.1145/773453.808204>.

RAUBER, T.; RüNGER, G. **Parallel Programming: for Multicore and Cluster Systems**. 2. ed. [S.l.]: Springer Publishing Company, Incorporated, 2013. ISBN 978-3-642-04817-3.

ROBERT, Y. et al. Encyclopedia of parallel computing. In: ____. [S.l.: s.n.], 2011. p. 2025–2029. ISBN 978-0-387-09765-7.

ROSE, C. A. F. D. Fundamentos de processamento de alto desempenho. In: **In. ERAD 2002. São Leopoldo – RS: SBC. 2002**. [S.l.: s.n.], 2002.

ROTEM, E. et al. Intel alder lake cpu architectures. **IEEE Micro**, v. 42, n. 3, p. 13–19, 2022.

SOLIMAN, M. I. Performance evaluation of multi-core intel xeon processors on basic linear algebra subprograms. In: **2008 International Conference on Computer Engineering Systems**. [S.l.: s.n.], 2008. p. 3–9.

TAN, G. et al. Fast implementation of dgemm on fermi gpu. In: **SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis**. [S.l.: s.n.], 2011. p. 1–11.

VAJDA, A. Multi-core and many-core processor architectures. **Programming Many-Core Chips.**, 06 2011.

XIANYI, Z.; QIAN, W.; YUNQUAN, Z. Model-driven level 3 blas performance optimization on loongson 3a processor. In: **2012 IEEE 18th International Conference on Parallel and Distributed Systems**. [S.l.: s.n.], 2012. p. 684–691.

ZEE, F. G. Van; GEIJN, R. A. van de. BLIS: A framework for rapidly instantiating BLAS functionality. **ACM Transactions on Mathematical Software**, v. 41, n. 3, p. 14:1–14:33, June 2015. Disponível em: <https://doi.acm.org/10.1145/2764454>.