UNIVERSIDADE FEDERAL DO PARÁ
CAMPUS UNIVERSITÁRIO DE TUCURUÍ
ENGENHARIA DE COMPUTAÇÃO

ERICK VINÍCIUS DAMASCENO DA SILVA

NO CERNE DA EFICIÊNCIA: UMA ANÁLISE COMPARATIVA SOBRE COMPILADORES E SEU DESEMPENHO

Tucuruí-PA
Novembro de 2023

UNIVERSIDADE FEDERAL DO PARÁ
CAMPUS UNIVERSITÁRIO DE TUCURUÍ
ENGENHARIA DE COMPUTAÇÃO

ERICK VINÍCIUS DAMASCENO DA SILVA

NO CERNE DA EFICIÊNCIA: UMA ANÁLISE COMPARATIVA SOBRE COMPILADORES E SEU DESEMPENHO

Trabalho de Conclusão de Curso apresentado para obtenção do grau de Bacharel em Engenharia de Computação.

Orientador: Prof. Dr. Marcos Túlio Amaris González

Tucuruí-PA
Novembro de 2023

Damasceno, Erick

No Cerne Da Eficiência: Uma Análise Comparativa sobre Compiladores e seu Desempenho/ Erick Vinícius Damasceno da Silva. – Tucuruí-PA, Novembro de 2023.

83 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. Marcos Túlio Amaris González

Monografia – UNIVERSIDADE FEDERAL DO PARÁ
CAMPUS UNIVERSITÁRIO DE TUCURUÍ
, Novembro de 2023.

1. HPC. 2. Compiladores. 3. Computação-Paralela. I. Título.

Erick Vinícius Damasceno da Silva

No Cerne Da Eficiência: Uma Análise Comparativa sobre Compiladores e seu Desempenho

Trabalho de Conclusão de Curso apresentado para obtenção do grau de Bacharel em Engenharia de Computação.

Data da Defesa: 14 de Dezembro de 2023
Conceito: EXCELENTE.

Banca Examinadora

Prof. Dr. Marcos Túlio Amaris González
Faculdade de Engenharia de Computação -
UFPA (CAMTUC)
Orientador

Prof. Dr. Otávio Noura Teixeira
Faculdade de Engenharia de Computação -
UFPA (CAMTUC)
Membro da Banca

Prof. Dr. Daniel da Conceição Pinheiro
Faculdade de Engenharia de Computação -
UFPA (CAMTUC)
Membro da Banca

Tucuruí-PA
Novembro de 2023

AGRADECIMENTOS

Expresso minha profunda gratidão à minha família, em especial aos meus pais, Maria Anunciação Damasceno e Moisés Oliveira, e ao meu falecido irmão, Douglas dos Santos, que sempre proporcionou alegria em minha vida e a quem dedico este trabalho.

Estendo meus agradecimentos a um grande amigo que se tornou parte fundamental da minha vida e desta jornada, Fellipe Augusto, e ao meu orientador, o Professor Dr. Marcos Amaris, cuja orientação possibilitou a criação e condução desta pesquisa.

Além disso, agradeço a todos os professores que tive na Faculdade de Engenharia de Computação, UFPA- Tucuruí, por contribuir com o meu crescimento pessoal e profissional.

Minha gratidão se estende aos meus amigos e a todos que, de alguma forma, colaboraram e apoiaram-me ao longo desta jornada. Suas contribuições foram essenciais para o êxito deste trabalho.

“Aquele que se empenha a resolver as dificuldades resolve-as antes que elas surjam. Aquele que se ultrapassa a vencer os inimigos triunfa antes que as suas ameaças se concretizem.”

(Sun Tzu)

RESUMO

Desde a década de 1950, o crescimento exponencial na demanda por poder de processamento impulsionou o avanço da Computação de Alto Desempenho (HPC). Nesse cenário, os Compiladores desempenham um papel crucial, ultrapassam as barreiras dos elementos de hardware e das linguagens de programação. Essas ferramentas essenciais têm a responsabilidade vital de traduzir o código de alto nível elaborado pelos programadores para a linguagem de máquina compreendida pelo computador. Este estudo realiza uma análise comparativa da eficiência de Compiladores e utiliza métricas de consumo de energia em diversos domínios da Computação de Alto Desempenho. O objetivo é identificar quais compiladores se destacam em diferentes áreas de análise e assim proporcionar uma base sólida para a escolha criteriosa dessas ferramentas, alinhada à natureza específica de cada tarefa. A análise indica que, do total de energia consumida, o GCC foi responsável por 33.23%, o Clang por 36.01%, e o ICC por 30.76%, respectivamente. Além disso, o ICC demonstrou ser 7.43% mais eficiente que o GCC, enquanto o Clang foi 8.35% menos eficiente.

Palavras-chave: HPC. Compiladores. Computação Paralela. Clang. ICC. GCC.

ABSTRACT

Since the 1950s, the exponential growth in demand for processing power has propelled the advancement of High-Performance Computing (HPC). In this scenario, compilers play a crucial role, surpassing the barriers of hardware elements and programming languages. These essential tools bear the vital responsibility of translating the high-level code crafted by programmers into the machine language understood by the computer. This study conducts a meticulous comparative analysis of compiler efficiency, utilizing energy consumption metrics across various domains of high-performance computing. The aim is to identify which compilers stand out in different areas of analysis, providing a solid foundation for the discerning selection of these tools, aligned with the specific nature of each task. The analysis indicates that, of the total energy consumed, GCC accounted for 33.23%, Clang for 36.01%, and ICC for 30.76%, respectively. Furthermore, ICC proved to be 7.43% more efficient than GCC, while Clang was 8.35% less efficient.

Keywords: HPC. Compilers. Parallel Computing. Clang. ICC. GCC.

LISTA DE ILUSTRAÇÕES

Figura 1 – Comparação de Consumo de Energia do Stream Cluster 57
Figura 2 – Correlação entre duração e consumo energético total do Stream Cluster . . . 58
Figura 3 – Distribuição Média Percentual entre CPU e DRAM em relaçao ao consumo energético total do Stream Cluster 58
Figura 4 – Média Percentual de ganho de consumo em relação ao compilador GCC do Stream Cluster 59
Figura 5 – Comparação de Consumo de Energia do LU Decomposition 60
Figura 6 – Correlação entre duração e consumo energético total do LU Decomposition 60
Figura 7 – Distribuição Média Percentual entre CPU e DRAM em relaçao ao consumo energético total do LU Decomposition 61
Figura 8 – Média Percentual de ganho de consumo em relação ao compilador GCC do LU Decomposition 62
Figura 9 – Comparação de Consumo de Energia do LavaMD 62
Figura 10 – Correlação entre duração e consumo energético total do LavaMD 63
Figura 11 – Distribuição Média Percentual entre CPU e DRAM em relaçao ao consumo energético total do LavaMD 64
Figura 12 – Média Percentual de ganho de consumo em relação ao compilador GCC do LavaMD 65
Figura 13 – Comparação de Consumo de Energia do Back Propagation 65
Figura 14 – Correlação entre duração e consumo energético total do Back Propagation . 66
Figura 15 – Distribuição Média Percentual entre CPU e DRAM em relaçao ao consumo energético total do Back Propagation 66
Figura 16 – Média Percentual de ganho de consumo em relação ao compilador GCC do Back Propagation 67
Figura 17 – Comparação de Consumo de Energia do Myocyte 68
Figura 18 – Correlação entre duração e consumo energético total do Myocyte 68
Figura 19 – Distribuição Média Percentual entre CPU e DRAM em relaçao ao consumo energético total do Myocyte 69
Figura 20 – Média Percentual de ganho de consumo em relação ao compilador GCC do Myocyte 70
Figura 21 – Verificação e Analise de Outlier do Stream Cluster 71
Figura 22 – Verificação e Analise de Outlier do LU Decomposition 71
Figura 23 – Verificação e Analise de Outlier do LavaMD 71
Figura 24 – Verificação e Analise de Outlier do Back Propagation 72
Figura 25 – Verificação e Analise de Outlier do Myocyte 72
Figura 26 – Detecção de anomalia em Myocyte compilado em GCC utilizando One Class SVM 73

Figura 27 – Detecção de anomalia em Myocyte compilado em Clang utilizando One Class SVM 73
Figura 28 – Detecção de anomalia em Myocyte compilado em ICC utilizando One Class SVM 73
Figura 29 – Distribuição de Consumo dos Compiladores para cada Benchmark 74
Figura 30 – Média Percentual de consumo total dos compiladores 75
Figura 31 – Média Percentual de consumo total dos compiladores excluindo Myocyte . . 75
Figura 32 – Comparação dos tamanhos dos binários gerados 76
Figura 33 – Comparação do aumento Percentual do tamanho dos arquivos em relação ao GCC 77

LISTA DE TABELAS

Tabela 1 – Exemplificação do arquivo resultante da coleta padrão do Joule It 50
Tabela 2 – Exemplificação do arquivo CSV gerado no Script de coleta 52
Tabela 3 – Exemplificação dos Dados de Tamanho dos Arquivos 52
Tabela 4 – Exemplificando Dataset resultante após tratamento com Pandas 53

LISTA DE ABREVIATURAS E SIGLAS

IBM - International Business Machines Corporation

CPU - Central Processing Unit

GPU - Graphics Processing Unit

OpenMP - Open Multi-Processing

CUDA - Compute Unified Device Architecture

OpenCL - Open Computing Language

MPI - Message Passing Interface

FORTRAN - Formula Translation

LISP - List Processing

GCC - GNU Compiler Collection

MSR - Model-Specific Registers

DSP - Digital Signal Processor

SSA - Static Single Assignment

GPL - General Public License

GNU - GNU's Not Unix

ICC - Intel C Compiler

C - Linguagem de Programação C

COBOL - Common Business-Oriented Language

ADA - Named after Ada Lovelace

API - Application Programming Interface

LLVM - Low-Level Virtual Machine

SUMÁRIO

1	**INTRODUÇÃO**	**14**
1.1	**Problemática**	**16**
1.2	**Objetivo de Estudo**	**16**
1.2.1	Objetivo Geral	16
1.2.2	Objetivo Especifico	17
1.2.2.1	Coletar Dados de Consumo	17
1.2.2.2	Comparar Eficiência dos Compiladores	17
1.2.2.3	Relacionar os Resíduos Gerados durante a Compilação	17
1.3	**Organização do Trabalho**	**17**
2	**REFERENCIAL TEÓRICO**	**19**
2.1	***Green Computing***	**19**
2.2	**Linguagem C**	**19**
2.2.1	OpenMP	20
2.2.2	GCC - GNU C Compiler	20
2.2.3	LLVM e CLANG	21
2.2.4	ICC - Intel C Compiler	23
2.3	**Suite de Benchmarks de HPC**	**24**
2.3.1	Parsec-Ompss	24
2.3.2	*Nas Parallel Benchmark*	24
2.3.3	Rodinia	25
2.3.4	*LavaMD*	29
2.3.5	*LU Decomposition*	29
2.3.6	*Back Propagation*	30
2.3.7	*Myocyte*	31
2.3.8	*Stream Cluster*	31
2.4	**Ferramentas de *Profiling***	**32**
2.4.1	Linux - Ubuntu	32
2.4.2	Intel RAPL	33
2.4.3	*Power Api* e *Joule It*	34
3	**TRABALHOS RELACIONADOS**	**36**
3.1	**Síntese dos trabalhos relacionados**	**44**
4	**METODOLOGIA**	**45**
4.1	**Componentes de Hardware**	**45**
4.2	**Componentes de Software**	**46**
4.2.1	Sistema Operacional utilizado	46
4.2.2	Escolha do *Benchmark*	46
4.2.3	Compiladores selecionados	47

4.2.4 Compilação dos *benchmarks* 48
4.2.5 Dando Permissão ao Intel RAPL 49
4.2.6 *Script* de Coleta e Joule It 50
4.2.7 *Script* de Coleta do tamanho dos Binários 52
4.2.8 Pré-Processamento dos dados Coletados 53
4.2.9 Análise Exploratória 53
4.2.9.1 Consumo Total de Execução 54
4.2.9.2 Tempo Total de Execução 54
4.2.9.3 Uma Comparação detalhada 54
4.2.9.4 Utilizando *Outliers* para identificar Anomalias 54
4.2.9.5 Avaliação Residual dos Binários 55
5 RESULTADOS 56
5.1 Avaliando individualmente cada *Benchmark* 56
5.1.1 Resultado: *Stream Cluster* 56
5.1.2 Resultado: LU Decomposition 59
5.1.3 Resultado: LavaMD 62
5.1.4 Resultado: Back Propagation 65
5.1.5 Resultado: Myocyte 67
5.2 Análise de *Outliers* 70
5.2.1 Pré Visualização com *BoxPlots* 70
5.2.2 Uso de *One Class SVM* para detectar anomalia em *Myocyte* 72
5.3 Comparação dos Compiladores em Relação aos *Benchmarks* 74
5.3.1 Distribuição do consumo dos Compiladores por *Benchmark* 74
5.3.2 Média Percentual do Consumo Total 74
5.4 Comparando os Binários Gerados para cada *Benchmark* 76
5.4.1 Tamanho dos arquivos gerados 76
5.4.2 Aumento Percentual 76
6 CONCLUSÕES E TRABALHOS FUTUROS 78
6.1 Trabalhos Futuros 79
REFERÊNCIAS 80

1 INTRODUÇÃO

Com o crescente aumento na demanda por poder de processamento, motivado pela necessidade de resoluções mais rápidas e da capacidade de lidar com problemas de maior complexidade, a IBM identificou, já na década de 1950, que a utilização de um único nó de computação ou um único núcleo de processamento se mostrava insuficiente para atender a essa demanda em constante crescimento. Contudo, somente na década de 1960, as pesquisas e desenvolvimentos necessários para abordar essa necessidade emergente foram plenamente realizados (DONGARRA; LUSZCZEK, 2017).

Nesse contexto, o Departamento de Defesa dos Estados Unidos desempenhou um papel crucial no desenvolvimento dessa tecnologia. Durante o período da Guerra Fria, o departamento reconheceu a extrema importância da Computação de Alto Desempenho para tarefas militares, como simulações de armas, comunicações criptografadas e análise de informações de inteligência. Foi nesse ambiente de crescente demanda e desafios de segurança que o Departamento de Defesa iniciou os esforços para desenvolver tecnologias capazes de atender a essas necessidades complexas (DONGARRA, 2006).

No contexto descrito, emerge a Computação de Alto Desempenho (*High Performance Computing*, HPC, na sigla em inglês). Em 1964, um marco significativo neste domínio foi estabelecido com a introdução do computador mainframe CDC 6600, notório por ser o sistema computacional de maior desempenho no período compreendido entre 1964 e 1969, sendo posteriormente sucedido pelo CDC 7600. A Computação de Alto Desempenho se define por sua finalidade primordial de atender às crescentes demandas por cálculos e simulações de extrema complexidade e visa significativa melhoria nos resultados obtidos (THORNTON, 1980).

Esse cenário histórico destaca a evolução da Computação de Alto Desempenho como resposta às necessidades de poder computacional que se acentuaram ao longo do tempo. A introdução do CDC 6600 representou um marco tecnológico que permitiu a realização de cálculos e simulações anteriormente inatingíveis em escalas de tempo aceitáveis. A busca por melhorias constantes no desempenho computacional impulsionou a sucessão de sistemas mais avançados, como o CDC 7600 e destaca-se assim a necessidade de contínua inovação nesse campo (ZWAKENBERG, 1970).

Na década de 1950, o campo das linguagens de programação viu um avanço, destinado a aprimorar substancialmente a maneira como os problemas computacionais da época eram abordados. Entre as linguagens que se destacaram na esfera da Computação Paralela e que permanecem relevantes até os dias atuais, continuamente atualizada e amplamente empregada com esse propósito, destaca-se a linguagem de programação FORTRAN. Criada por John Warner Backus em 1954, o FORTRAN desempenhou um papel pioneiro e fundamental no desenvolvimento de soluções de alto desempenho para cálculos computacionais complexos (BACKUS; HEISING, 1964).

Além do FORTRAN, que é uma linguagem notável e pioneira na Computação Paralela, outras linguagens também se destacaram durante o mesmo período e desempenharam papéis importantes no desenvolvimento da programação e da computação em geral.

O COBOL (Common Business-Oriented Language) se destaca como uma das linguagens notáveis. Sua criação remonta ao final da década de 1950, e a linguagem foi concebida com o propósito de atender às demandas de empresas e organizações. O COBOL obteve uma ampla adoção em aplicações relacionadas a negócios e processamento de dados. Embora sua origem não estivesse diretamente ligada à Computação Paralela, o COBOL desempenhou um papel fundamental na automatização de processos empresariais. Esse avanço, por sua vez, estimulou o desenvolvimento de sistemas computacionais de alto desempenho, que se mostraram essenciais para lidar com enormes volumes de dados e tarefas operacionais complexas (SAMMET, 1978).

Por contraste, a linguagem de programação LISP (List Processing), concebida por John McCarthy na década de 1950, emergiu como um protagonista no domínio da Inteligência Artificial e simulações complexas. A notoriedade do LISP decorre de sua notável flexibilidade e habilidade de manipular estruturas de dados complexas. Essas características fazem dela uma ferramenta de grande valor para a resolução de problemas computacionais que abrangem áreas como lógica, simulações e Aprendizado de Máquina (JR, 1982).

Essas linguagens representam o início da evolução das linguagens de programação e a diversificação de suas aplicações. O FORTRAN, o COBOL e o LISP desempenharam papéis fundamentais na história da computação, cada um atendendo a necessidades específicas e contribui para o desenvolvimento da Computação Paralela e de Alto Desempenho. O legado dessas linguagens ainda é evidente na programação contemporânea e nas tecnologias que dependem de cálculos computacionais avançados.

Além das linguagens mencionadas, a principal abordagem adotada neste estudo é a linguagem C. Conforme destacado por (RITCHIE, 1993), a criação da linguagem C ocorreu entre os anos de 1969 e 1973, em paralelo com o desenvolvimento inicial do sistema operacional Unix, sendo o ano de 1972 considerado um período particularmente criativo. Um conjunto adicional de transformações significativas ocorreu entre 1977 e 1979, durante a demonstração da portabilidade do sistema Unix.

No contexto de programação para computadores paralelos, a linguagem de escolha é frequentemente a linguagem imperativa, como é o caso da linguagem C, complementada por diversas construções de passagem de mensagens, notavelmente através do MPI (*Message Passing Interface*), conforme ressaltado por (HATCHER et al., 1991).

1.1 Problemática

Quando abordamos a Computação Paralela e Alto Desempenho, é importante compreender que não apenas os elementos de *hardware*, as Linguagens de Programação utilizadas e os métodos de escrita de algoritmos desempenham um papel crítico na definição do consumo, da velocidade e da eficiência global de uma aplicação. Os Compiladores, nesse contexto, emergem como ferramentas de importância vital, pois têm um impacto significativo no comportamento final de uma aplicação.

Um compilador é uma ferramenta que traduz programas escritos em linguagens de programação de alto nível para versões de baixo nível, adequadas para serem processadas por montadores e vinculadores. Em termos mais simples, sua principal função é converter código compreensível para programadores em instruções específicas que um computador pode executar diretamente. Em vez de abordar questões de design de linguagem, um compilador concentra-se na eficaz seleção de instruções de máquina para implementar um programa específico, permitindo que ele seja executado de maneira eficiente pelo hardware do computador (DANIEL, 2009).

À medida que os *softwares* avançaram e foram acompanhados por *hardware* projetado para enfrentar desafios de alta performance, surgiu a necessidade de estabelecer critérios de eficiência e uma metodologia para avaliar o desempenho dessas plataformas. Para atender a essa demanda, foram desenvolvidas ferramentas projetadas para submeter esses sistemas a testes rigorosos, com o objetivo de mensurar sua capacidade de processamento. Essas ferramentas, conhecidas como *benchmarks*, têm a finalidade de avaliar a capacidade da máquina em solucionar uma variedade de problemas matemáticos ou mesmo realizar simulações de cenários do mundo real.

Atualmente, uma crescente atenção é dedicada à demanda de energia e à eficiência geral de qualquer plataforma que tenha impacto no meio ambiente. Nesse contexto, o propósito deste estudo é avaliar a capacidade de *benchmarks* empregados na medição do desempenho de sistemas de alto processamento em relação ao consumo de energia e ao tempo médio necessário para executar tarefas específicas. Para atingir esse objetivo, foram selecionados diversos *benchmarks* do conjunto Rodinia e executados em três diferentes compiladores da linguagem C, nomeadamente o GCC, CLANG e ICC.

1.2 Objetivo de Estudo

1.2.1 Objetivo Geral

Analisar e comparar a eficiência energética e o tempo de execução de aplicações de Computação de Alto Desempenho usando diferentes compiladores da linguagem C e a biblioteca OpenMP para máquinas multi-core.

1.2.2 Objetivo Especifico

1.2.2.1 Coletar Dados de Consumo

Efetuar uma coleta abrangente dos dados de consumo energético de cada *benchmark* selecionado e realizar três coletas distintas para cada *benchmark*. Cada coleta será conduzida utilizando um Compilador específico, permitindo uma análise comparativa detalhada do impacto da escolha do Compilador no consumo energético.

1.2.2.2 Comparar Eficiência dos Compiladores

Realizar uma avaliação comparativa da eficiência dos *benchmarks* ao utilizar cada compilador selecionado. Este processo visa elucidar como cada compilador se comporta em cenários distintos e fornecer informações sobre a performance relativa dos compiladores diante de diferentes demandas e características dos *benchmarks*.

1.2.2.3 Relacionar os Resíduos Gerados durante a Compilação

Estabelecer uma correlação entre o tamanho residual de cada binário gerado após a compilação e seu impacto no desempenho final avaliado. Esta análise pretende avaliar o papel dos resíduos gerados durante a compilação na eficiência e performance global dos *benchmarks* e contribuir para uma compreensão mais abrangente dos fatores que influenciam o desempenho do sistema.

1.3 Organização do Trabalho

A estruturação deste documento proposto segue uma organização sistemática, que visa conduzir o estudo deste trabalho de maneira a facilitar a compreensão do tema abordado.

Capítulo 1: A introdução apresenta um panorama histórico da computação de Alto Desempenho e explorar como a evolução de ferramentas e Linguagens de Programação impacta esse campo. Destaca a problemática associada à influência dos compiladores na eficiência de sistemas HPC.

Capítulo 2: Oferece uma abordagem teórica dos tópicos que serão discutidos e proporciona uma base para a compreensão das técnicas abordadas nesta pesquisa.

Capítulo 3: Explora trabalhos que serviram como guia para a condução desta pesquisa e possibilita abordagens dinâmicas na elaboração deste trabalho.

Capítulo 4: Explica as ferramentas de *hardware* e *software* utilizadas para obter e analisar os dados e validar a concepção desta pesquisa.

Capítulo 5: Demonstra os resultados obtidos a partir dos métodos abordados e explicar de maneira gráfica e descritiva os dados coletados e suas relações de eficiência para cada

compilador.

Capítulo 6: Finaliza a monografia com uma conclusão que aborda os principais aspectos observados ao longo da pesquisa e orientar de maneira eficiente a escolha de um compilador em determinados problemas.

Por fim, a seção de referências lista todas as fontes acadêmicas, artigos e materiais utilizados para fundamentar a pesquisa e as discussões ao longo da monografia e oferecer reconhecimento e a possibilidade dos leitores explorarem as fontes originais para fins de pesquisa futura.

2 REFERENCIAL TEÓRICO

Este capítulo tem como propósito empregar referências teóricas para fundamentar o uso de conceitos e ferramentas adotados ao longo deste trabalho.

Ao estabelecer uma base teórica sólida, buscasse respaldo na literatura especializada para justificar as escolhas conceituais e metodológicas feitas durante a condução da pesquisa. Serão abordados e discutidos os fundamentos teóricos que sustentam os conceitos centrais, modelos analíticos e abordagens metodológicas aplicadas.

As referências teóricas desempenham um papel crucial ao fornecer uma estrutura conceitual que orienta e valida as decisões tomadas ao longo do desenvolvimento da pesquisa. Além disso, ao ancorar o trabalho em teorias consolidadas, busca-se contribuir para a robustez e a validade das análises e conclusões apresentadas.

Através da exploração dessas referências, o leitor terá a oportunidade de compreender a fundamentação teórica que sustenta a escolha e a aplicação de cada conceito e ferramenta para fortalecer assim a credibilidade e a contribuição deste trabalho para o campo de estudo em questão.

2.1 *Green Computing*

A Computação Verde, ou "Green Computing", refere-se ao uso eficiente e ambientalmente responsável de computadores e seus recursos. Esta abordagem abrange várias práticas e estratégias.

Com o aumento da disseminação dos meios computacionais, houve também a necessidade do aumento do poder computacional, o que por consequência aumentou também o consumo de energia e, por consequência, a emissão de dióxido de carbono (CO_2) no meio ambiente.

Os problemas ambientais gerados pelo aumento da emissão de CO_2 e o custo financeiro ocasionado pelo consumo de energia têm impulsionado estudos que visam o desenvolvimento de mecanismos e tecnologias que façam uso eficiente da energia. Esses mecanismos e tecnologias são denominados Computação Verde ou *green computing* (WILLIAMS; CURTIS, 2008).

2.2 Linguagem C

Em termos mais simples, a linguagem C é uma linguagem de programação versátil, inicialmente desenvolvida para o sistema operacional UNIX, que não é considerada de alto nível. Essa linguagem tem se destacado como uma ferramenta valiosa para tarefas em tempo real e computacionalmente intensivas. Ainda que existissem preferências históricas por outras linguagens, como FORTRAN, para processamento digital de sinais, a transição para C é apontada

como inevitável devido às suas vantagens superiores. (EMBREE; KIMBLE; BARTRAM, 1991)

2.2.1 OpenMP

(KIESSLING, 2009) diz que a interface de programação de aplicativos (API) OpenMP (*Open Multi-Processing*) foi inicialmente lançada em 1997 e é um padrão para escrever aplicações paralelas de memória compartilhada em C, C++ e Fortran. O OpenMP oferece a vantagem de ser facilmente implementado em códigos seriais existentes que permite uma paralelização incremental. Além disso, destaca-se por ser amplamente utilizado, altamente portátil e idealmente adequado para arquiteturas *multi-core*, que tornam-se cada vez mais populares em computadores *desktop* do dia a dia.

2.2.2 GCC - GNU C Compiler

O criador original do GNU C Compiler (GCC) é Richard Stallman, o visionário por trás do Projeto GNU. Em 1984, o Projeto GNU foi lançado com a missão de desenvolver um sistema operacional baseado em *software* livre, assemelhando-se ao UNIX, com o objetivo de fomentar a liberdade e a colaboração entre os usuários de computadores e programadores. Naquela época, não havia compiladores de código aberto disponíveis, e, portanto, o Projeto GNU teve que começar do zero. O financiamento para esse trabalho foi obtido por meio de doações de indivíduos e empresas, canalizadas pela *Free Software Foundation*, uma organização sem fins lucrativos (GOUGH; STALLMAN, 2004).

A Evolução do Conjunto de Compiladores GNU (GCC) é notável, tendo começado como um modesto compilador C e se transformado em uma poderosa ferramenta multilíngue capaz de gerar código para mais de 30 arquiteturas diferentes. Essa versatilidade em lidar com diversos idiomas e arquiteturas solidificou a posição do GCC como um dos compiladores mais amplamente utilizados. É amplamente empregado como o compilador-padrão em todas as distribuições Linux e também goza de grande popularidade nos meios acadêmicos, onde é empregado em pesquisas relacionadas a compiladores (NOVILLO, 2006).

Ao longo do tempo, o GCC foi expandido para suportar uma variedade de idiomas adicionais e inclui Fortran, ADA, Java e Objective-C. A sigla GCC agora é comumente usada para se referir à "Coleção de Compiladores GNU."(GOUGH; STALLMAN, 2004).

Apesar de sua difundida aceitação, o GCC historicamente apresentou desafios significativos em termos de manutenção e aprimoramento e chega ao ponto de tornar algumas funcionalidades praticamente impossíveis de serem implementadas. Uma das barreiras reside no tamanho considerável de seu código-fonte. Embora não seja colossal em termos da indústria de software, o GCC ainda é um projeto substancial, composto por cerca de 1,3 milhão de linhas de código-fonte (MLOC). Ao incorporar as bibliotecas de tempo de execução necessárias para dar suporte a diversos idiomas, a contagem cresce para aproximadamente 2,2 MLOC. Contudo, o

tamanho não é o único obstáculo que o GCC enfrenta.

Compiladores, por natureza, são intrinsecamente complexos e demandam um profundo conhecimento teórico, especialmente no campo de otimização e análise (NOVILLO, 2006).

O desenvolvimento do GCC é supervisionado pelo Comitê Diretor do GCC, composto por representantes das comunidades de usuários do GCC na indústria, pesquisa e academia. Um dos destaques do GCC é sua portabilidade - ele é executado em praticamente todas as plataformas disponíveis atualmente e pode gerar código para uma ampla gama de processadores. Além de processadores usados em computadores pessoais, o GCC também oferece suporte a microcontroladores, DSPs e CPUs de 64 bits.

Além de ser um compilador nativo, o GCC também tem a capacidade de compilar programas para sistemas diferentes daquele em que está sendo usado. Isso permite que o software seja compilado para sistemas embarcados que não possuem seu próprio compilador. O GCC é escrito em C com um foco sólido na portabilidade, o que facilita sua adaptação a novos sistemas, uma vez que ele próprio pode ser compilado.

O GCC é modular por *design* e permite a adição de suporte para novas linguagens e arquiteturas. A incorporação de um novo *front-end* no GCC possibilita a utilização desse idioma em qualquer arquitetura e fornece recursos, como bibliotecas, durante a execução. Da mesma forma, quando se adiciona suporte para uma nova arquitetura, essa disponibilidade se estende a todas as linguagens.

Em um de seus lançamentos, o GCC passou por uma reformulação em sua infraestrutura interna para enfrentar esses desafios. Essa revisão possibilitou a introdução de um novo otimizador global baseado em SSA, análises de dependência de dados altamente sofisticadas, um vetorizador multissistema, um verificador de limites de memória conhecido como "mudflap"e diversas outras funcionalidades inovadoras (NOVILLO, 2006).

Por último, mas não menos importante, o GCC é um *software* livre distribuído sob a Licença Pública Geral GNU (GNU GPL). Isso significa que você tem a liberdade de utilizar e modificar o GCC, como é o caso de todos os *softwares* GNU. Isso também facilita o suporte para novos tipos de CPUs, idiomas ou funcionalidades quando necessário. (GOUGH; STALLMAN, 2004)

2.2.3 LLVM e CLANG

LLVM (*Low Level Virtual Machine*), um *framework* de compiladores projetado para suportar análise e transformação transparentes e contínuas de programas ao longo de toda a sua vida, sendo aplicável a programas arbitrários. O LLVM fornece informações de alto nível para transformações de compiladores em tempo de compilação, tempo de ligação, tempo de execução e durante os intervalos ociosos entre execuções. (LATTNER; ADVE, 2004)

O LLVM define uma representação comum de código em um formato de atribuição única estática (SSA), com características inovadoras e inclui um sistema de tipos simples e independente de linguagem que expõe as primitivas comumente utilizadas para implementar recursos de linguagens de alto nível. Além disso, apresenta uma instrução para aritmética de endereços tipada e um mecanismo simples que pode ser empregado para uniformemente e eficientemente implementar recursos de tratamento de exceção em linguagens de alto nível (e setjmp/longjmp em C).

O *framework* de compiladores LLVM e a representação de código juntos proporcionam uma combinação de capacidades essenciais para a análise e transformação prática e contínua de programas. Até onde sabemos, nenhuma abordagem de compilação existente oferece todas essas capacidades. Descrevesse o *design* da representação LLVM e do *framework* de compiladores e avaliar o *design* de três maneiras (LATTNER; ADVE, 2004):

- O tamanho e a eficácia da representação e incluir as informações de tipo que ela fornece;
- O desempenho do compilador para vários problemas interprocedurais [1];
- Exemplos ilustrativos dos benefícios que o LLVM proporciona para vários problemas desafiadores de compiladores [2].

O Clang atua como o *driver* C/C++ para o LLVM e serve como o principal ponto de entrada para a construção de aplicativos. Sua função consiste em selecionar os comandos corretos a partir das cadeias de ferramentas apropriadas, com base nas opções fornecidas pelo usuário e nas informações disponíveis sobre os arquivos de entrada, como extensões de arquivo conhecidas. A incorporação de suporte para *offloading* OpenMP introduz complexidade adicional na implementação, pois, para o mesmo conjunto de arquivos de entrada, diferentes cadeias de ferramentas precisam ser invocadas, e as saídas parciais de cada cadeia de ferramentas são combinadas em um arquivo binário totalmente funcional que contém imagens executáveis para o host e o dispositivo. (ANTAO et al., 2016)

Ele realiza a análise do código-fonte, verifica possíveis erros e cria uma Árvore Sintática Abstrata (AST) para representar a entrada de código. Essa AST é, então, convertida para um *Intermediate* LLVM IR (Representação Intermediária do LLVM), onde todas as otimizações do LLVM são aplicadas. Por fim, o *backend* traduz esse código intermediário para gerar o código de máquina. Um princípio fundamental no *design* do Clang é a adoção de uma arquitetura baseada em bibliotecas, na qual diferentes componentes do *frontend* são separados em várias bibliotecas e permite sua combinação para atender a diversas necessidades e usos. Essa abordagem favorece a criação de interfaces sólidas e facilita a integração de novos desenvolvedores. (MISHRA; MALIK; CHAPMAN, 2020a)

[1] Problemas interprocedurais: Desafios e questões relacionadas à análise e otimização de código que envolve múltiplos procedimentos ou funções em um programa.

[2] Desafio para Compiladores: Implementação de *offloandings*.

2.2.4 ICC - Intel C Compiler

O Conjunto de Compiladores Intel C++ (ICC) é uma compilação de ferramentas destinadas à linguagem de programação C e C++ e oferece suporte para sistemas operacionais como Linux, Microsoft Windows e Mac OS X. Desenvolvido pela Intel Corporation, o ICC é uma ferramenta comercial disponibilizada para instituições acadêmicas sem custos associados que promove o acesso facilitado a recursos avançados de compilação.

Com foco na arquitetura x86, o ICC permite a compilação eficiente para diversas plataformas baseadas nessa arquitetura que proporciona flexibilidade e desempenho otimizado. Além disso, destaca-se pelo suporte integral ao OpenMP 4.0 que possibilita a implementação eficaz de paralelismo em programas, e pela capacidade de realizar paralelização automática, especialmente útil em ambientes de multiprocessamento simétrico.

O conjunto de suportes *front-end* do compilador abrange não apenas as linguagens de programação C e C++, mas também Fortran, ampliando sua utilidade e aplicabilidade em projetos que envolvem diversas linguagens de programação. (MACHADO et al., 2017)

O *Intel C Compiler* (ICC) representa um conjunto avançado de compiladores C e C++ desenvolvidos pela Intel, conforme mencionado em (REN, 2014). Esta ferramenta, embora comercial, destaca-se pelo desempenho na geração de código otimizado, especialmente para arquiteturas IA-32 e Intel 64. É importante observar que, mesmo em processadores não Intel, mas compatíveis, como alguns processadores AMD, o ICC é capaz de produzir código funcional, embora não otimizado de maneira ideal.

Os compiladores Intel são cuidadosamente projetados para sistemas de computador que utilizam processadores compatíveis com as arquiteturas Intel, que visa minimizar travamentos e otimizar a execução do código com o menor número possível de ciclos de *clock*.

O *Intel C Compiler* vai além da simples compilação e incorpora três técnicas distintas de otimização de alto nível para programas compilados:

1. **Otimização Interprocedural (IPO):** Esta técnica visa aprimorar o desempenho considerando a inter-relação entre diferentes partes do código e otimiza a execução de funções e procedimentos de forma conjunta.

2. **Otimização Guiada por Perfil (PGO):** A otimização PGO baseia-se na análise do comportamento do programa durante a execução onde ajusta o código para as características específicas de uso, o que pode resultar em melhorias significativas de desempenho.

3. **Otimizações de Alto Nível (HLO):** Essas otimizações visam melhorar o código em um nível mais abstrato que incorpora melhorias que transcendem a estrutura individual das funções e contribui para um desempenho global aprimorado.

Portanto, o Intel C Compiler destaca-se não apenas pela sua capacidade de compilar para arquiteturas Intel, mas também pela sofisticação de suas técnicas de otimização, proporcionando um ambiente de desenvolvimento eficiente e eficaz para programadores em busca de desempenho superior.

2.3 Suite de Benchmarks de HPC

2.3.1 Parsec-Ompss

A suíte original de *benchmarks* PARSEC é composta por um conjunto diversificado e representativo de aplicações de *benchmark*, sendo útil para avaliar arquiteturas *multi-core* de memória compartilhada. No entanto, ela suporta apenas três modelos de programação: Pthreads [3] (SPMD), OpenMP (parallel for), TBB [4] (parallel for, pipeline), que carece de suporte para modelos emergentes e generalizados de programação paralela de tarefas. Neste trabalho, apresentasse um PARSEC paralelizado por tarefas (TP-PARSEC), no qual adicionamos traduções para cinco modelos diferentes de programação paralela de tarefas (Cilk Plus, MassiveThreads, OpenMP Tasks, Qthreads [5], TBB).

O paralelismo de tarefas possibilita uma descrição mais intuitiva de algoritmos paralelos em comparação com a abordagem SPMD de encadeamento direto que garante um melhor equilíbrio de carga em um grande número de núcleos de processador com a comprovada técnica de escalonamento de roubo de trabalho. O TP-PARSEC não é apenas útil para desenvolvedores de sistemas paralelos de tarefas analisarem seus sistemas de tempo de execução com uma ampla variedade de cargas de trabalho de diversas áreas, mas também permite comparar as diferenças de desempenho entre sistemas. O TP-PARSEC é integrado a uma ferramenta de análise e visualização de desempenho centrada em tarefas, o que ajuda efetivamente os usuários a entender o desempenho, identificar gargalos de desempenho e, especialmente, analisar as diferenças de desempenho entre sistemas. (HUYNH et al., 2019)

2.3.2 *Nas Parallel Benchmark*

Os *Nas Parallel Benchmark* (NPB) foram desenvolvidos para medir as capacidades computacionais de Sistemas de Alto Desempenho. Inicialmente propostos em 1992, a implementação paralela completa com MPI [6] ocorreu em 1997 (NPB, versão 2.3). À medida que os sistemas paralelos evoluíram, surgiu um novo gargalo na velocidade de escrita e leitura de arquivos, especialmente em aplicações práticas.

[3] Pthreads: Padrão POSIX para programação de threads em sistemas operacionais UNIX e similares.
[4] TBB: Intel Threading Building Blocks, biblioteca C++ para programação paralela.
[5] QThread: Classe na biblioteca Qt para manipulação de threads em aplicações C++.
[6] MPI: Message Passing Interface, padrão para comunicação em programação paralela distribuída.

A limitação de desempenho de I/O decorreu da necessidade de estruturas de arquivos independentes do número de processadores. O foco recaiu nas capacidades de saída do sistema, especialmente em aplicações científicas com decomposição de domínio, onde muitos processadores contribuíam para um arquivo de saída que causa competição e degradação de desempenho. Para superar isso, foi reconhecido que a comunicação interprocessador era mais rápida que as velocidades de I/O que levou ao desenvolvimento da Interface MPI I/O. Essa interface aproveita as relações entre dados na memória e no disco para otimizar o tráfego de dados, sem interferência adicional do programador que usa *buffering* coletivo. (WONG; WIJNGAART, 2003)

2.3.3 Rodinia

O *Rodinia Benchmark Suite*, uma amplamente adotada coleção de *benchmarks*, é empregado para avaliar o desempenho de Sistemas de Computação Paralela que abrangem CPUs multi-núcleo, GPUs e ambientes com múltiplos nós. A criação desta suíte teve como objetivo fundamental estabelecer uma abordagem normalizada para a avaliação de desempenho onde abarcam tanto *hardware* quanto *software*, especialmente em cenários de Computação Paralela e Distribuída.

Originária da Universidade da Virgínia, nos Estados Unidos, esta suíte de *benchmarks*, conhecida como *Rodinia Benchmark Suite*, foi concebida com a finalidade de fornecer um conjunto diversificado de *benchmarks* que abrangesse uma ampla variedade de domínios que incluem simulações físicas, processamento de imagens, análise de dados, entre outros. O projeto teve seu início em 2009 e, desde então, tem sido cuidadosamente mantido e atualizado pela ativa comunidade de pesquisa em computação paralela.

Rodinia não se limita apenas a abranger aplicações em domínios emergentes, tais como Bioinformática, Mineração de Dados e Processamento de Imagens; abrange também implementações de aceleradores para algoritmos clássicos de grande importância, como decomposição LU e Travessia de Grafos. A Taxonomia *Berkeley Dwarf* foi originalmente empregada como um guia para a seleção de aplicações a serem incluídas na suíte Rodinia, com o objetivo de assegurar a preservação de padrões paralelos cruciais. (CHE et al., 2010)

A Taxonomia Berkeley Dwarf é um modelo conceitual desenvolvido pela Universidade da Califórnia, Berkeley, que visa classificar e categorizar as tarefas e padrões de Computação Paralela em Sistemas De Alto Desempenho. Essa taxonomia ajuda a entender e organizar as diversas dimensões da computação paralela onde auxilia os pesquisadores a analisar, comparar e otimizar o desempenho de diferentes aplicações paralelas. A Taxonomia Berkeley Dwarf se concentra em sete "anões"ou padrões fundamentais de Computação Paralela, que são os seguintes:

1. Operações de Laço (*Loop*): Esse padrão lida com a paralelização de laços ou loops em códigos, nos quais a mesma operação é repetida várias vezes, tornando-o adequado para CPUs

multi-core e GPUs;

2. Tarefa Paralela (*Task Parallelism*): Refere-se à paralelização de tarefas independentes que podem ser executadas simultaneamente. Isso é particularmente relevante para sistemas com muitos núcleos de CPU e ambientes distribuídos;

3. Memória Distribuída (*Distributed Memory*): Trata da comunicação e sincronização entre processos que têm sua própria memória local. É importante em Sistemas Distribuídos, como *clusters*;

4. Memória Compartilhada (*Shared Memory*): Lida com a coordenação e comunicação entre threads que compartilham a mesma memória. Isso é relevante para sistemas *multi-core* e muitos processadores;

5. Memória Compartilhada Distribuída (*Hybrid Shared Distributed Memory*): Combina os modelos de memória compartilhada e memória distribuída, sendo útil em sistemas que têm múltiplos nós com memória compartilhada local;

6. Comunicação (*Communication*): Enfoca a comunicação entre processos onde inclui trocas de mensagens, uma parte essencial em sistemas distribuídos;

7. Sincronização (*Synchronization*): Lida com a coordenação temporal das tarefas em sistemas paralelos que garantem que as operações sejam realizadas na ordem correta.

Essa taxonomia ajuda os pesquisadores e desenvolvedores a identificar as características de desempenho e os desafios de programação associados a cada tipo de anão e a escolher as abordagens de paralelização mais adequadas para suas aplicações. Ela também é útil para o desenvolvimento de ferramentas e técnicas de Programação Paralela e para a compreensão dos requisitos de *hardware* necessários para lidar com diferentes padrões de Computação Paralela. A Taxonomia *Berkeley Dwarf* é uma contribuição significativa para a compreensão e o desenvolvimento de sistemas de alto desempenho. (ASANOVIC et al., 2006)

O Rodinia se destaca de outros conjuntos de *benchmarks* por meio de várias características distintas (CHE et al., 2010):

1. Aproveita Hierarquias de Memória Não Tradicionais: As implementações do Rodinia exploram hierarquias de memória não convencionais, como *scratchpad* e unidades de textura, para fins de computação geral. Isso inclui a utilização de memórias alternativas ao cache gerenciado por hardware, exemplificado pelo uso de tecnologias como *Cell* e *ClearSpeed*. Essa abordagem requer o desenvolvimento de *benchmarks* que acompanhem essa evolução;

2. Versões Múltiplas com Camadas de Otimização: O Rodinia disponibiliza várias versões de algumas aplicações, cada uma com diferentes camadas de otimização. Essa variedade permite que projetistas avaliem o impacto de diversas implementações em sua arquitetura ou projetos de Compiladores que facilita a otimização do desempenho;

3. Modelo de "*Offloading*": As aplicações do Rodinia seguem um modelo de "offloa-

ding"que pressupõe que os aceleradores utilizam uma área de memória separada da memória principal. Isso reflete a arquitetura de muitos aceleradores e é considerado nas implementações dos *benchmarks*;

4. Desafio para Compiladores: O Rodinia fornece um conjunto de aplicativos que, devido à complexidade ou características específicas, podem representar um desafio para os compiladores na geração automática de código acelerador. Isso impulsiona a pesquisa em técnicas de compilação e otimização.

Os principais objetivos da suíte Rodinia são:

Fornecer um conjunto de *benchmarks* diversificado: A suíte inclui vários aplicativos de benchmark, representando diferentes tipos de cargas de trabalho paralelas e distribuídas.

Facilitar a medição de desempenho: Cada *benchmark* incluído na suíte é acompanhado por métricas de desempenho que ajudam os pesquisadores e profissionais a avaliar o desempenho de seus sistemas.

Promover a pesquisa em Computação Paralela: O *Rodinia Benchmark Suite* é uma ferramenta essencial para pesquisadores que estudam Arquiteturas Paralelas, Sistemas Distribuídos e Otimização de Código.

Os *benchmarks* incluídos no Suite Rodinia são:

- Leukocyte: Realiza a segmentação de células sanguíneas brancas em micrografias digitais para análise de imagens médicas;
- Heart Wall: Simula a propagação do potencial elétrico pelo músculo cardíaco para análise de atividade cardíaca;
- MUMmerGPU: Realiza alinhamento de sequências de DNA para detecção de similaridade entre genomas;
- CFD Solver: Resolve numericamente as equações de fluidos computacionais para simulações de fluxo de fluido;
- LU Decomposition: Realiza a decomposição LU de uma matriz para resolver sistemas lineares de equações;
- HotSpot: Modela a dissipação de calor em circuitos integrados para análise de temperatura;
- Back Propagation: Implementa o algoritmo de retropropagação usado em Redes Neurais Artificiais para treinamento;
- Needleman-Wunsch: Alinha sequências de DNA para análise de similaridade e busca de padrões;

- Kmeans: Implementa o algoritmo de agrupamento *k-means* usado em Aprendizado de Máquina;
- Breadth-First Search: Realiza uma busca em largura em grafos para encontrar caminhos ou componentes conectados;
- SRAD: Realiza a redução de ruído em imagens médicas através de uma técnica de difusão anisotrópica;
- Streamcluster: Implementa um algoritmo de clusterização para análise de fluxo de dados contínuos;
- Particle Filter: Implementa um filtro de partículas usado em rastreamento e estimativa de estado em sistemas dinâmicos;
- PathFinder: Implementa um algoritmo de busca de caminho em grafos para encontrar o caminho mais curto;
- Gaussian Elimination: Resolve sistemas lineares de equações através da eliminação de Gauss;
- k-Nearest Neighbors: Implementa o algoritmo de k-vizinhos mais próximos usado em classificação e regressão;
- LavaMD: Realiza simulações de dinâmica molecular para análise de interações entre partículas;
- Myocyte: Modela a propagação de potenciais elétricos em células cardíacas para análise de atividade cardíaca;
- B+ Tree: Implementa a estrutura de dados B+ Tree para armazenamento e busca eficientes de dados;
- GPUDWT: Realiza a transformada *Wavelet* discreta em imagens para Análise de Sinal;
- Hybrid Sort: Implementa um algoritmo de ordenação híbrido para ordenar grandes conjuntos de dados;
- Hotspot3D: Modela a dissipação de calor em circuitos integrados tridimensionais para análise térmica;
- Huffman: Implementa o algoritmo de compressão de dados Huffman para compactar e descompactar dados.

2.3.4 *LavaMD*

O LavaMD é um *benchmark* que representa uma aplicação de dinâmica molecular e é amplamente utilizado em pesquisas para avaliar a capacidade de Sistemas Paralelos e Distribuídos no processamento de tarefas intensivas (STREITZ et al., 2005).

Características e Funcionamento do LavaMD:

1. Dinâmica Molecular: O LavaMD simula a dinâmica molecular de partículas interagindo em um sistema. Ele calcula as forças de atração e repulsão entre as partículas ao longo do tempo para prever seus movimentos subsequentes. Isso é fundamental em muitas áreas da pesquisa científica, incluindo química, biologia, física e materiais;

2. Interação entre Partículas: O *benchmark* considera a interação entre partículas, como íons, átomos ou moléculas, e calcula as forças que agem sobre elas. Essas forças são determinadas por modelos matemáticos que descrevem as interações eletrostáticas e as ligações químicas;

3. Método de N-Body: O LavaMD geralmente utiliza o método N-Body para calcular as forças entre todas as partículas no sistema. Esse método envolve a avaliação das interações entre todas as combinações possíveis de partículas, o que pode ser altamente intensivo em termos computacionais (SZAFARYN et al., 2011);

4. Paralelização: Para avaliar o desempenho de sistemas paralelos, o LavaMD é projetado para ser executado em paralelo, aproveitando vários núcleos de CPU ou aceleradores, como GPUs. Isso permite que o *benchmark* avalie o desempenho de *hardware* em cargas de trabalho intensivas em computação;

5. Medição de Desempenho: Os resultados do LavaMD são usados para avaliar a taxa de processamento das interações entre partículas em sistemas paralelos. Ele é uma ferramenta valiosa para determinar quão eficaz um sistema é na execução de cálculos complexos de dinâmica molecular.

2.3.5 *LU Decomposition*

A Decomposição LU (Lower-Upper ou L-U) é um método numérico utilizado na Álgebra Linear para fatorar uma matriz em dois componentes: uma matriz triangular inferior (L) e uma matriz triangular superior (U). (GOLUB; LOAN, 2013)

Como a Decomposição LU Funciona:

1. Matriz de Entrada: O processo começa com uma matriz quadrada de ordem n (n x n), que precisa ser decomposta em L e U;

2. Eliminação Gaussiana: A Decomposição LU envolve a aplicação da Eliminação Gaussiana para transformar a matriz original em duas matrizes:

- Matriz triangular superior (*Upper*)

- Matriz triangular inferior (*Lower*)

A eliminação gaussiana é um método que utiliza operações elementares para transformar a matriz em sua forma triangular superior;

3. Matriz Triangular Superior (U): A matriz U contém os coeficientes da matriz original após a aplicação da eliminação gaussiana. Ela tem zeros abaixo da diagonal principal e não precisa ser modificada após a decomposição;

4. Matriz Triangular Inferior (L): A matriz L é criada durante o processo de eliminação gaussiana. Ela é uma matriz triangular inferior com uns na diagonal principal;

5. Matriz Composta: A matriz original é fatorada em duas partes, L e U, de modo que A = LU.

2.3.6 *Back Propagation*

A Retro-propagação (ou *backpropagation*) é um algoritmo essencial na área de Aprendizado de Máquina e Redes Neurais Artificiais. Ele é usado para treinar redes neurais a aprender a partir de dados e a ajustar seus pesos de maneira a minimizar o erro na tarefa desejada (DE, 1986).

Como a Retropropagação Funciona:

1. Inicialização dos Pesos: Inicialmente, os pesos das conexões entre os neurônios da rede neural são atribuídos a valores aleatórios ou pequenos valores próximos a zero;

2. Forward Pass (Passagem Direta): O algoritmo realiza uma passagem direta para calcular as saídas da rede neural com base nos dados de entrada. Cada neurônio calcula uma soma ponderada das entradas, aplica uma função de ativação e passa o resultado para a próxima camada;

3. Cálculo do Erro: A diferença entre as saídas previstas e as saídas reais é calculada, representando o erro. O erro é geralmente quantificado usando uma função de custo, como o erro quadrático médio (MSE);

4. *Backward Pass* (Passagem de Retorno): A partir da saída, o algoritmo inicia uma passagem de retorno para ajustar os pesos. Isso envolve o cálculo das derivadas parciais do erro em relação a cada peso que usa a regra da cadeia (GOODFELLOW; BENGIO; COURVILLE, 2016);

5. Atualização dos Pesos: Os pesos são ajustados com as derivadas calculadas na etapa anterior. A ideia é ajustar os pesos de forma que o erro seja minimizado. Isso é feito através do método do gradiente descendente, onde os pesos são atualizados em direção ao gradiente negativo da função de custo (BISHOP; NASRABADI, 2006);

6. Iteração: Os passos 2 a 5 são repetidos várias vezes (épocas) para melhorar gradualmente o desempenho da rede neural. A Retro-propagação continua até que o erro convirja para um valor mínimo ou até que um número máximo de épocas seja atingido.

2.3.7 *Myocyte*

O Myocyte é um benchmark que simula o comportamento de células cardíacas, sendo usado para testar e avaliar o desempenho de sistemas de alto desempenho em tarefas relacionadas à modelagem biológica. (CHAVAN et al., 2021)

Como o *Benchmark Myocyte* Funciona:

1. Simulação de Células Cardíacas: O *Myocyte* é projetado para simular o comportamento de células cardíacas no coração humano. Especificamente, ele se concentra na modelagem das células miocárdicas, que são responsáveis pela contração do músculo cardíaco;

2. Modelo Matemático: O *benchmark* utiliza modelos matemáticos complexos para representar o comportamento elétrico e mecânico das células cardíacas. Isso inclui a modelagem de canais iônicos, potenciais de ação e interações mecânicas;

3. Paralelização: Para avaliar o desempenho de Sistemas Paralelos, o *Myocyte* é paralelizado para tirar proveito de vários núcleos de CPU ou aceleradores, como GPUs. Isso permite que o *benchmark* avalie o desempenho de *hardware* em cargas de trabalho intensivas em computação;

4. Medição de Desempenho: Os resultados do *Myocyte* são usados para medir a taxa de processamento de simulações de células cardíacas. Ele é uma ferramenta valiosa para determinar quão eficaz um sistema é na execução de cálculos complexos de biologia celular.

2.3.8 *Stream Cluster*

O *Streamcluster* é uma aplicação de Análise de Dados que lida com fluxos contínuos de dados em tempo real, como aqueles gerados por sensores, dispositivos *IoT* (Internet das Coisas) ou registros de eventos. A principal tarefa do *Streamcluster* é agrupar (clusterizar) esses dados à medida que eles chegam e identificam padrões e estruturas nos dados em movimento. (DASH; SAHU,)

Como o *Streamcluster* Funciona:

1. Entrada de Dados Contínuos: O *benchmark Streamcluster* recebe uma entrada contínua de dados à medida que eles são gerados. Esses dados podem ser vetores multidimensionais representando observações ou eventos em tempo real;

2. Agrupamento em Tempo Real: À medida que os dados chegam, o Streamcluster realiza a clusterização em tempo real, ou seja, ele agrupa os dados em clusters à medida que eles entram. Ele usa algoritmos de clusterização, como o algoritmo de K-médias (K-means), para atribuir pontos de dados a clusters com base em suas características e similaridades;

3. Atualização Contínua: Conforme novos dados entram, o *Streamcluster* pode atualizar os *clusters* existentes ou criar novos *clusters*, conforme necessário. Isso permite que ele se adapte a padrões de dados em evolução;

4. Paralelização: Para avaliar o desempenho de sistemas paralelos, o *Streamcluster* é frequentemente paralelizado para aproveitar vários núcleos de CPU ou aceleradores de *hardware*, como GPUs. Isso é particularmente importante para lidar com a análise de grandes volumes de dados em tempo real;

5. Medição de Desempenho: O objetivo do *Streamcluster* é medir a taxa de processamento de dados e a capacidade de um sistema em realizar a clusterização em tempo real. Os resultados do *benchmark* são usados para avaliar o desempenho de *hardware* e *software* em cenários de análise de fluxos de dados contínuos.

2.4 Ferramentas de *Profiling*

2.4.1 Linux - Ubuntu

O Linux é um sistema operacional de código aberto criado por Linus Torvalds em 1991. Diferentemente de sistemas operacionais proprietários, o código-fonte do Linux é acessível e pode ser modificado por qualquer pessoa. O núcleo do sistema, conhecido como *kernel* Linux, é responsável pela interação com o *hardware* do computador. Além do *kernel*, um sistema operacional Linux completo é composto por uma variedade de utilitários, bibliotecas e *software* de aplicação de código aberto que proporciona estabilidade, segurança e eficiência.

O Linux é amplamente utilizado em servidores, supercomputadores, dispositivos embarcados e *desktops*. Existem diversas distribuições Linux, ou "distros", que oferecem conjuntos de *software* e configurações específicos para diferentes propósitos. Entre as distribuições mais conhecidas estão Ubuntu, Fedora, Debian, CentOS e Arch Linux, cada uma com sua interface gráfica do usuário, gerenciador de pacotes e filosofias de *design*.

O sistema GNU/Linux é centrado na liberdade e proporciona aos usuários a capacidade de escolher a distribuição que melhor atende às suas necessidades. Não há uma distribuição superior a outra, nem todas são isentas de imperfeições. Cada distribuição apresenta suas vantagens e desvantagens que torna a escolha de uma distribuição um processo que requer análise cuidadosa. Uma decisão inadequada pode resultar em complicações futuras. Embora o *Kernel* Linux seja uniforme em todas as distribuições, as estruturas individuais de cada uma variam. Para usuários novos e inexperientes, a transição entre distribuições pode ser desafiadora. Portanto, é crucial reiterar a importância de uma escolha criteriosa ao selecionar a distribuição mais adequada. (FILHO, 2012)

Ubuntu é um Sistema Operacional (SO) baseado no Debian, cujo desenvolvimento é liderado pela Canonical Ltd., propriedade de Mark Shuttleworth. A distribuição Ubuntu é um

software livre e de código aberto, licenciado sob a *GNU General Public License*, juntamente com várias outras licenças gratuitas. O sistema operacional Ubuntu possui diversas versões, incluindo a versão para *desktop*, a versão para servidor e a versão para dispositivos móveis. As versões de *desktop* e servidor diferem no pacote de interface gráfica do usuário utilizado. Na versão para *desktop*, as interfaces gráficas UNITY [7] ou GNOME [8] são pré-instaladas por padrão, enquanto a versão para servidor possui uma interface de linha de comando por padrão. A versão móvel do Ubuntu é integrada a um *kernel* de tempo real para lidar com operações determinísticas, como chamadas telefônicas. Vale notar que a versão móvel do Ubuntu é relativamente recente e encontra-se em estágios iniciais de desenvolvimento, sendo omitida deste estudo. (TABASSUM; MATHEW, 2014)

2.4.2 Intel RAPL

A funcionalidade Intel RAPL (*Running Average Power Limit*) é um recurso de *hardware* incorporado em processadores Intel mais modernos, proporcionando a capacidade de monitorar o consumo de energia da CPU, e em alguns casos, da DRAM e da GPU integrada, de maneira programática. No estudo intitulado "RAPL in Action"(KHAN et al., 2018), foram investigadas as implicações e benefícios da interface RAPL. Para essa análise, foram empregadas abordagens que incluem *microb-enchmarks*, avaliações de desempenho em aplicativos reais e conjuntos de dados complementares obtidos do *cluster* Taito, mantido pelo Centro Finlandês de Computação Científica.

Os resultados do estudo apontaram que "as leituras do RAPL demonstram uma correlação sólida com o consumo de energia da tomada que apresenta um potencial promissor e um impacto quase imperceptível no desempenho do sistema". Portanto, esses achados sugerem que o RAPL pode ser aplicado com êxito em experimentos relacionados ao consumo de energia, embora algumas preocupações persistam, como a previsibilidade do tempo de atualização dos registros e eventuais desvios de registro (DICKSON; SEBOK, 2023).

O RAPL é um conjunto de *Model-Specific Registers* (MSRs) adequados para rastrear o consumo de energia e foi introduzido na linha de processadores *Sandy Bridge*. As interfaces RAPL podem ser utilizadas para recuperar informações sobre o consumo de energia nos seguintes domínios:

- **Pacote (PKG)**: O consumo de energia da tomada;
- **Power Plane 0 (PP0)**: O consumo de energia de todos os núcleos do processador no soquete;

[7] Unity: Unity3D, uma poderosa engine de desenvolvimento de jogos e ambiente de criação para aplicações interativas em 2D e 3D.

[8] GNOME: Ambiente de desktop para sistemas operacionais Unix, conhecido por sua interface gráfica amigável e flexível.

- **Power Plane 1 (PP1)**: O consumo de energia da GPU integrada;
- **DRAM**: O consumo de energia da memória RAM anexada à CPU.

É importante observar que nem todos os domínios estão disponíveis em todos os dispositivos. PP1 só pode ser acessado em CPUs de *desktop*, enquanto a DRAM só está disponível em dispositivos de servidor. PKG e PP0 estão disponíveis em CPUs de cliente e servidor.

Os MSRs RAPL são registradores de 32 bits e são atualizados aproximadamente a cada 1 milissegundo, o que permite a coleta de informações de consumo de energia em tempo real (ou quase em tempo real). No entanto, é essencial considerar que, como observado por fontes anteriores, esses registradores podem transbordar facilmente quando acessados e lidos diretamente.

Adicionalmente, diferentes arquiteturas de processador podem armazenar os valores de energia consumida em unidades e incrementos diferentes. Por exemplo, o valor padrão para processadores Sandy Bridge é de 15,3 µJ. Os valores das unidades de potência e tempo podem ser determinados lendo o registrador MSR_RAPL_POWER_UNIT.

A interface RAPL também oferece funcionalidade para limitar o uso de energia em diferentes domínios da CPU, mas, no contexto deste projeto, focamos apenas na funcionalidade de relatório dos MSRs RAPL.

2.4.3 *Power Api* e *Joule It*

A Power API (Application Programming Interface) é uma interface de programação de aplicativos que fornece acesso a informações e controle relacionados ao consumo de energia e ao gerenciamento de energia em sistemas de computação. A Power API permite que os desenvolvedores de software interajam com componentes de hardware relacionados à energia, como processadores, unidades de processamento gráfico (GPUs), unidades de processamento central (CPUs), DRAM (Dynamic Random-Access Memory), entre outros, a fim de monitorar e otimizar o consumo de energia de um sistema.

A Power API pode fornecer informações detalhadas sobre o consumo de energia em tempo real, permitindo que os desenvolvedores monitorem o uso de energia de aplicativos específicos ou de componentes de hardware. Isso é particularmente valioso em cenários onde o gerenciamento de energia é crítico, como em dispositivos móveis, servidores e centros de dados, onde a eficiência energética é uma preocupação importante.

Algumas das funcionalidades comuns da Power API podem incluir:

1. Monitoramento de Energia: A API pode fornecer informações detalhadas sobre o consumo de energia em diferentes componentes do sistema, permitindo que os desenvolvedores acompanhem o uso de energia ao longo do tempo;

2. Controle de Energia: A Power API também pode permitir o controle do consumo de energia, permitindo que os desenvolvedores ajustem as configurações de energia em tempo real para otimizar o desempenho e a eficiência energética;

3. Limitação de Energia: Em sistemas onde é importante evitar o superaquecimento ou atingir limites de energia, a API pode ser usada para definir limites de consumo de energia;

4. Perfil de Energia: A API pode ajudar a criar perfis de consumo de energia para aplicativos ou sistemas inteiros, o que pode ser usado para otimizar o uso de energia em cenários específicos;

5. Relatórios de Diagnóstico: A API pode fornecer informações de diagnóstico sobre como a energia está sendo consumida em um sistema, o que é útil para identificar gargalos de energia e ineficiências.

Dentre as diversas ferramentas contidas na coleção da PowerAPI, a que mais se encaixou com a proposta da pesquisa foi Jouleit. Onde esta ferramenta é um script que deve ser executado em conjunto com a aplicação desejada para que possa medir o consumo energetico, consumo da CPU, memória e tempo gasto durante a execução.

3 TRABALHOS RELACIONADOS

Este capítulo tem como objetivo apresentar as referências de trabalhos relacionados que serviram como fundamentação para o desenvolvimento desta pesquisa.

Ao explorar trabalhos anteriores e estudos relevantes, buscasse estabelecer um sólido embasamento teórico e contextual para o presente trabalho. A revisão da literatura foi conduzida de maneira abrangente que considera pesquisas e publicações que abordam temas similares ou diretamente relacionados aos objetivos desta investigação.

Discutesse as principais contribuições encontradas na literatura onde destacasse conceitos, metodologias e resultados que influenciaram a formulação das questões de pesquisa, a definição das abordagens metodológicas e a interpretação dos resultados obtidos.

Modeling Power and Energy Usage of HPC Kernels (TIWARI et al., 2012) informa, a importância dos *kernels* intensivos em computação é evidente no cenário de aplicações de Computação de Alto Desempenho (HPC), já que estes *kernels* representam a maior parte do tempo de execução. Nesse contexto, muitas das características de consumo de energia e demanda de potência de aplicações HPC podem ser analisadas em termos do comportamento destes *kernels* constituintes. Dada a relevância das restrições relacionadas a potência e energia como obstáculos significativos no design de sistemas exascala, é crucial desenvolver uma compreensão mais profunda de como os *kernels* se comportam em termos de potência/energia quando submetidas a diferentes otimizações baseadas em compiladores e configurações de *hardware*.

No desenvolvimento de modelos de potência e energia para unidades centrais de processamento (CPUs) e módulos de memória (DIMMs) por meio do treinamento de Redes Neurais Artificiais para três *kernels* de HPC amplamente utilizadas. Essas redes neurais são treinadas com dados empíricos coletados na arquitetura de destino. Os modelos utilizam parâmetros de otimização específicos do *kernel* e ajustes de *hardware* como entradas para previsões de taxa de consumo de potência e consumo de energia dos componentes do sistema. Os resultados apresentam uma taxa de erro absoluto que, em média, é inferior a 5.5% para três *kernels* importantes - multiplicação de matrizes (MM), computação de stencil e fatoração LU.

Seven Pillars to Achieve Energy Efficiency in High-Performance Computing Data Centers

(HUSSAIN et al., 2019), diz que a eficiência energética em ambientes de computação intensiva, como centros de dados e Sistemas de Alto Desempenho (HPC), tornou-se uma preocupação perene devido à sua relevância crucial na atualidade. O *design* eficiente em termos de energia e as medidas de ecologia energética representam desafios centrais em ambientes HPC. No entanto, as pesquisas atuais concentram-se em métodos práticos para medir a utilização de energia que visa tomar decisões em prol da Computação Verde sem exceder recursos e sem comprometer o desempenho. Este trabalho realiza uma análise abrangente sobre as questões,

desafios e soluções relacionadas ao consumo de energia em data centers e sistemas HPC, com foco no período de 2010 a 2016.

O estudo classifica os problemas existentes na eficiência energética que os *Data Centers* enfrentam, este que proporciona uma visão abrangente dos modelos adotados por cada abordagem. A contribuição do trabalho é dupla. Primeiramente, por meio dessa categorização, busca-se oferecer uma visão fácil e concisa do modelo subjacente de eficiência energética adotado por cada abordagem. Em segundo lugar, propõe-se um *framework* de sete pilares para a eficiência energética em sistemas HPC e *data centers*, que representa uma contribuição inédita na área.

Esse trabalho, ao abordar as questões relacionadas à eficiência energética, oferece uma base sólida para compreender o estado da arte no período analisado. A proposta do *framework* de sete pilares adiciona uma perspectiva estruturada para o desenvolvimento contínuo de estratégias eficazes em eficiência energética, destacasse a necessidade de considerar múltiplos aspectos nesse contexto.

Performance assessment of OpenMP constructs and benchmarks using modern compilers and multi-core CPUs (GAWRYCH; CZARNUL,) se insere no contexto dos avanços contemporâneos na Arquitetura de Sistemas Computacionais, especialmente diante do crescente número de núcleos, expansão da memória cache e evolução das arquiteturas, bem como o constante aprimoramento de compiladores. A demanda por avaliações precisas de cargas de trabalho, frequentemente executadas e representativas, torna-se imperativa nesse cenário em constante transformação.

A principal métrica explorada neste artigo é a velocidade de execução, dado que a potência computacional dos modernos CPUs é predominantemente derivada da utilização eficiente de múltiplos núcleos. Os experimentos realizados abrangem uma variedade de códigos que incluem normalização de lotes, convolução, função linear, multiplicação de matrizes, teste de números primos e equação de onda. Essas operações foram executadas com diferentes compiladores, como GNU gcc, LLVM clang, icx e icc, em quatro sistemas distintos, compostos por 1 ou 2 sockets, nomeadamente: 1 x Intel Core i7-5960X, 1 x Intel Core i9-9940X, 2 x Intel Xeon Platinum 8280L e 2 x Intel Xeon Gold 6130.

Os resultados obtidos neste estudo fornecem valiosas sugestões relativas ao escalonamento em CPUs específicas, inclusive com configurações recomendadas de número de *threads*. Ao discutir e analisar os desempenhos alcançados em diferentes configurações de *hardware* e compiladores, este trabalho contribui para o entendimento mais aprofundado do impacto das escolhas arquiteturais e de *software* na eficiência computacional. Dessa forma, fornece uma base sólida para investigações futuras e otimizações direcionadas no desenvolvimento de aplicações que visam explorar ao máximo o potencial dos modernos sistemas computacionais.

SIMD programming using Intel vector extensions (AMIRI; SHAHBAHRAMI, 2020) o intuito da pesquisa se insere no contexto das extensões de instrução única para dados múltiplos

(SIMD), uma das capacidades mais significativas dos Processadores de Propósito Geral (GPPs) modernos, destinada a aprimorar o desempenho de aplicações com modificações mínimas no *hardware*. Cada fornecedor de GPP, como HP, Sun, Intel e AMD, apresenta sua própria Arquitetura de Conjunto de Instruções (ISA) e microarquitetura SIMD com perspectivas distintas. A Intel, em particular, tem desempenhado um papel proeminente na expansão das tecnologias SIMD tanto do ponto de vista de hardware quanto de software.

As tecnologias SIMD introduzidas pela Intel, como *MultiMedia eXtensions* (MMX), *Streaming SIMD Extensions* (SSE), *Advanced Vector eXtensions* (AVX), *Fused Multiply Add* (FMA) e AVX-512, representam avanços significativos. O desenvolvimento dos processadores também testemunhou a expansão da largura do registrador de 64 bits para 512 bits, e o aumento do número de registradores vetoriais de 8 para 32. Esses registradores mais amplos proporcionam mais caminhos de paralelismo, enquanto o aumento no número de registradores vetoriais reduz movimentos de dados extras para a memória cache.

Para aproveitar as vantagens das extensões SIMD, diversas abordagens de programação foram desenvolvidas. A Vetorização Automática do Compilador (CAV), como uma abordagem de vetorização implícita, oferece ferramentas simples e fáceis para SIMDização. No entanto, o aprimoramento de desempenho do CAV nem sempre é garantido. Por outro lado, a Modelagem de Programação Intrínseca (IPM) proporciona acesso de baixo nível aos registradores vetoriais para a SIMDização explícita. No entanto, programar com IPM exige um grande conhecimento, especialmente nas características de arquitetura de baixo nível que torna crucial a escolha de instruções adequadas e metodologias de vetorização para mapear um algoritmo específico.

O objetivo deste trabalho é triplo. Primeiramente, oferece uma revisão da tecnologia SIMD em geral e das extensões SIMD da Intel em particular. Em segundo lugar, discute comparativamente algumas características SIMD das tecnologias da Intel, como MMX, SSEs, AVX e FMA, em termos de ISA, largura do vetor e ferramentas de programação SIMD.

Em terceiro lugar, para comparar o desempenho de diferentes auto-vetorizadores e abordagens de IPM usando o compilador Intel C++ (ICC), GNU Compiler Collection (GCC) e Low Level Virtual Machine (LLVM), mapeasse e implementasse alguns kernels multimídia representativos nas extensões AVX e AVX2. Finalmente, nossos resultados experimentais demonstram que, embora a melhoria de desempenho usando a abordagem IPM seja maior do que os CAVs, o programador necessita de mais esforços de programação e conhecimento de diferentes estratégias de mapeamento. Portanto, estender as capacidades dos auto-vetorizadores para gerar códigos vetorizados mais eficientes torna-se uma questão importante em diferentes compiladores.

GCC vs. ICC comparison using PARSEC Benchmarks (ALMOMANY; ALQURAAN; BALACHANDRAN, 2014) alinha-se com a proposta de avaliar o impacto de diversas otimizações de compiladores na performance de programas, utilizasse duas renomadas suítes de compiladores: o GNU C Compiler e o C/C++ Compiler da Intel que emprega *benchmarks* do PARSEC. A otimização de compiladores constitui o processo de ajustar a saída de um compilador

para minimizar ou maximizar atributos específicos de um programa executável. A otimização é frequentemente realizada através da ativação de *flags* de otimização.

O escopo desta pesquisa explora a possibilidade de aprimorar o desempenho do programa por meio de uma melhor utilização das características arquiteturais existentes, notadamente através das otimizações de compiladores. A ativação criteriosa dessas otimizações não apenas tem o potencial de aprimorar a performance do programa, mas também de reduzir a necessidade de atualizações onerosas e, por conseguinte, os custos do sistema em desenvolvimento.

A análise da eficácia das otimizações de compiladores apresenta-se como um componente crucial na busca por programas mais eficientes e econômicos. A escolha entre o GNU C Compiler e o C/C++ Compiler da Intel, bem como a seleção específica de flags de otimização, tornam-se elementos fundamentais para alcançar resultados que não apenas beneficiem o desempenho dos programas, mas também otimizem os recursos arquiteturais existentes.

Ao investigar o impacto dessas otimizações em *benchmarks* específicos, este trabalho contribui para a compreensão mais profunda de como as escolhas de otimização de compiladores podem influenciar significativamente o desempenho de programas em ambientes práticos. Essa análise comparativa entre suítes de compiladores amplamente utilizadas e a utilização de *benchmarks* reconhecidos acrescenta uma perspectiva valiosa ao corpo de conhecimento existente, fornecendo insights relevantes para o desenvolvimento de software eficiente e economicamente viável.

mdspan in C++: A Case Study in the Integration of Performance Portable Features into International Language Standards (HOLLMAN et al., 2019) a presença ubíqua de *arrays* [1] multidimensionais em Computação de Alto Desempenho (HPC) destaca sua importância, entretanto, a ausência destes na linguagem C++ representa uma limitação reconhecida e duradoura em seu uso para HPC. Este trabalho refere-se ao projeto e implementação do *mdspan* [2], uma proposta de visão de *arrays* multidimensionais para inclusão no padrão C++23. Essa proposta é amplamente inspirada pelo trabalho realizado no projeto *Kokkos* [3], um modelo de programação C++ de alto desempenho utilizado por diversas instituições de HPC para preparar suas bases de código para sistemas de supercomputação da classe exascala.

O texto descreve o *design* final do *mdspan* após um processo de cinco anos para alcançar consenso na comunidade C++. Especificamente, explora como o *design* aborda alguns dos desafios fundamentais da programação portátil de alto desempenho, destacasse os pontos de personalização que possibilitam uma extensão perfeita para áreas não atualmente abordadas pelo padrão C++, mas de importância crítica no mundo da computação heterogênea dos sistemas

[1] Arrays: Estruturas de dados que armazenam elementos do mesmo tipo em uma sequência indexada, oferecendo acesso eficiente por meio de índices.

[2] mdspan: Multidimensional span em C++, introduzido para representar arrays multidimensionais de forma eficiente.

[3] Projeto Kokkos: Uma biblioteca de programação paralela para facilitar o desenvolvimento eficiente em arquiteturas de hardware heterogêneas.

modernos.

Um aspecto significativo ressaltado no texto é a adaptação do *design* às necessidades de programação portátil de alto desempenho, além de fornecer uma implementação de alta qualidade em sua forma atual. Inclui também diversos *benchmarks* para demonstrar a natureza de zero *overhead* do design moderno proposto.

Essa iniciativa busca superar uma lacuna conhecida na linguagem C++, oferecendo uma solução robusta e padronizada para a manipulação eficiente de *arrays* multidimensionais. A análise desta proposta e sua implementação contribuem não apenas para o avanço do padrão C++, mas também para a melhoria do suporte à programação portátil em HPC e abre caminho para a computação eficiente em sistemas heterogêneos.

Extending the LLVM/Clang Framework for OpenMP Metadirective Support (MISHRA; MALIK; CHAPMAN, 2020b) o surgimento do OpenMP 5.0 introduziu diversas novas diretrizes para atender à demanda crescente dos Sistemas de Computação de Alto Desempenho emergentes. Entre essas novas diretrizes, as metadiretivas e as diretrizes de declaração de variantes desempenham um papel crucial no controle do comportamento de execução de uma aplicação, por meio de adaptação em tempo de compilação com base no contexto do OpenMP. A metadiretiva permite a seleção de diretrizes do OpenMP com base no contexto do OpenMP envolvente, bem como em condições definidas pelo usuário. Por outro lado, a diretriz de declaração de variante declara uma variante especializada de uma função base e especifica o contexto no qual essa variante especializada é utilizada.

Este trabalho destaca a implementação da diretriz de *metadiretiva* em Clang, uma vez que o suporte para metadiretiva não está disponível nesse compilador. A implementação apresentada suporta a especificação da metadiretiva do OpenMP 5.0 que inclui uma extensão dinâmica para condições especificadas pelo usuário. A avaliação dinâmica dessas condições proporciona aos programadores maior liberdade para expressar uma variedade de algoritmos adaptativos que melhoram o desempenho global de uma aplicação.

Uma aplicação prática destacada no texto envolve a estimativa do custo de execução de um kernel com base em variáveis dinâmicas ou estáticas, como tempo ou consumo de energia. A decisão de offload do kernel para uma GPU usando a metadiretiva é, portanto, adaptativa e baseada nas condições dinâmicas do sistema.

O trabalho inclui a modificação de vários códigos de *benchmark* na suíte de *benchmarks* Rodinia, que se destina a aplicações e *kernels* voltados para plataformas *multi-core* CPU e GPU. Essa modificação possibilita aos desenvolvedores de aplicativos estudar o comportamento da *metadiretiva*. A análise revela que a principal vantagem da implementação dinâmica da *metadiretiva* é a adição de um mínimo a nenhum overhead à aplicação do usuário, ao mesmo tempo em que oferece flexibilidade aos programadores para introduzir portabilidade e adaptabilidade em seu código. O trabalho na suíte de *benchmarks* Rodinia também fornece diretrizes valiosas para

os programadores alcançarem melhor desempenho com a *metadiretiva*.

An empirical study of the effect of source-level loop transformations on compiler stability (GONG et al., 2018) diz que a otimização moderna de compiladores é um processo complexo que não oferece garantias de fornecer o código-alvo mais rápido e eficiente. Por esse motivo, os compiladores enfrentam dificuldades em produzir um desempenho estável a partir de versões de código que realizam a mesma computação e diferem apenas na ordem das operações. Essa instabilidade torna os compiladores ferramentas de otimização de programa muito menos eficazes e frequentemente força os programadores a realizar uma busca de força bruta ao ajustar o desempenho. Neste trabalho, analisasse a estabilidade do processo de compilação e a margem de desempenho de três compiladores de propósito geral amplamente utilizados: GCC, ICC e Clang.

Para o estudo, extrair mais de 1.000 ninhos de *loops* de *benchmarks* conhecidos, bibliotecas e aplicações reais. Em seguida, aplicamos sequências de transformações de *loops* em nível de origem a esses ninhos de *loops* para criar numerosas mutações semanticamente equivalentes. Finalmente, analisamos o impacto das transformações na qualidade do código em termos de localidade, contagem dinâmica de instruções e vetorização.

Os resultados mostram que ao aplicar transformações de origem para origem e buscando as melhores configurações de vetorização, a porcentagem de *loops* acelerados em pelo menos 1,15x é de 46,7% para o GCC, 35,7% para o ICC e 46,5% para o Clang. Em média, o potencial de melhoria de desempenho é estimado em pelo menos 23,7% para o GCC, 18,1% para o ICC e 26,4% para o Clang.

A análise de estabilidade mostra que, sob nossa configuração experimental, o coeficiente de variação médio do tempo de execução em todas as mutações é de 18,2% para o GCC, 19,5% para o ICC e 16,9% para o Clang. O coeficiente de variação mais alto para um único ninho de loops atinge 118,9% para o GCC, 124,3% para o ICC e 110,5% para o Clang. Concluí-se que os compiladores avaliados precisam de melhorias adicionais para afirmarem que possuem comportamento estável.

How to Benchmark Supercomputers (XIE; XIAO, 2015) informa que o emprego de *benchmarks* desempenha um papel crucial na avaliação de supercomputadores, simulasse cargas de trabalho específicas para comparar plataformas, identificar gargalos de desempenho, avaliar soluções potenciais e auxiliar usuários na decisão de adquirir ou utilizar máquinas mais adequadas às suas necessidades de aplicação. Neste contexto, este trabalho busca contextualizar e aprofundar a compreensão sobre o propósito, importância e métodos associados ao *benchmarking* de supercomputadores.

O presente estudo destaca a diversidade de *benchmarks* existentes e explora o estado da arte nesta área. Cinco *benchmarks* principais para avaliação de supercomputadores são abordados. Entre eles, o *Linpack* destaca-se como o mais popular, sendo fundamental para a elaboração do

renomado ranking TOP500. Este *benchmark* se destina a medir o desempenho de Computação de Alto Desempenho e proporciona uma métrica valiosa para comparar e classificar sistemas.

Além do Linpack, o trabalho aborda o *benchmark* HPCG, cujo objetivo é estressar um equilíbrio entre a velocidade de operações de ponto flutuante e a largura de banda e latência de comunicação. A suíte de *benchmarks* Rodinia é mencionada como uma ferramenta essencial para avaliar supercomputadores heterogêneos, compostos por GPUs e CPUs multi-core. O SPEC *benchmark*, aplicável às gerações mais recentes de computadores de alto desempenho, é explorado como outra referência importante. Por fim, o DEISA *benchmark* é destacado que incorpora uma variedade de aplicações reais de diversas disciplinas, desde astrofísica até física de partículas.

Benchmarking OpenCL, OpenACC, OpenMP, and CUDA: Programming Productivity, Performance, and Energy Consumption (MEMETI et al., 2017) a evolução dos sistemas de computação paralela modernos frequentemente se depara com a heterogeneidade em nível de nó, onde os nós podem abranger CPUs de propósito geral e aceleradores (como GPU ou Intel Xeon Phi) que proporcionam alto desempenho com características adequadas de consumo de energia. No entanto, explorar o desempenho disponível em arquiteturas heterogêneas pode apresentar desafios significativos. Diversos *frameworks* de programação paralela, como OpenMP, OpenCL, OpenACC e CUDA, estão disponíveis, e a seleção do mais adequado para um contexto específico não é uma tarefa trivial.

Este trabalho se insere no âmbito da investigação empírica das características de OpenMP, OpenACC, OpenCL e CUDA, que foca na produtividade de programação, desempenho e consumo de energia. A avaliação da produtividade de programação é realizada através da ferramenta *CodeStat*, desenvolvida internamente, que permite determinar a porcentagem de linhas de código necessárias para paralelizar o código usando um *framework* específico. As ferramentas *MeterPU* e *x-MeterPU* são empregadas para avaliar o consumo de energia e o desempenho.

Os experimentos são conduzidos com a suíte de *benchmarks* SPEC e a suíte de benchmarks Rodinia, padrões da indústria, para computação acelerada em sistemas heterogêneos que combinam Processadores Intel Xeon E5 com aceleradores GPU ou um co-processador Intel Xeon Phi.

Esta pesquisa contribui para o entendimento prático das vantagens e limitações desses *frameworks* em ambientes heterogêneos, fornecendo informações valiosas para a escolha adequada de ferramentas de programação paralela em cenários específicos. A avaliação abrangente desses *frameworks* em termos de produtividade, desempenho e consumo de energia oferece uma base sólida para a otimização de aplicações em sistemas heterogêneos, maximizando o potencial dessas arquiteturas na computação paralela moderna.

Evaluating execution time predictions on GPU kernels using an analytical model and

machine learning techniques (AMARIS et al., 2023) diz que o desafio de prever o desempenho de aplicações executadas em GPUs é crucial para otimizar o agendamento eficiente de tarefas. Diversas abordagens são empregadas para alcançar essa previsão, destaca-se a modelagem analítica e técnicas de aprendizado de máquina (ML). Enquanto a modelagem analítica, como a baseada em BSP, oferece uma abordagem estruturada, as técnicas de ML exigem grandes conjuntos de treinamento e características confiáveis, que permite, no entanto, capturar interações complexas entre arquitetura e *software* sem intervenção manual.

Neste trabalho, foi realizada uma comparação entre um modelo analítico baseado em BSP para prever o tempo de execução de *kernels* em GPUs e três diferentes técnicas de ML: Regressão Linear, Máquina de Vetores de Suporte (SVM) e Floresta Aleatória. O modelo analítico baseia-se no número de operações de computação e acessos à memória da GPU, com informações adicionais sobre o uso de cache obtidas por meio de perfil de execução.

Os experimentos foram conduzidos utilizando 20 kernels CUDA, sendo 11 deles pertencentes a 6 aplicações do mundo real da suíte de *benchmarks* Rodinia, e os demais que representa aplicações clássicas de matriz-vetor comumente utilizadas para benchmarking. Os dados foram coletados em 9 GPUs NVIDIA em máquinas diferentes.

Os resultados indicam que o modelo analítico apresenta melhor desempenho na previsão quando as aplicações têm escalabilidade regular. Além disso, destaca-se que as técnicas de ML obtiveram alta precisão quando implementado um processo de extração de características. Conjuntos de 5 e 10 características foram testados de duas maneiras distintas, para GPUs e Kernels desconhecidos. Os experimentos com ML e processo de extração de características apresentaram erros em torno de 1.54% e 2.71%, respectivamente, para GPUs e Kernels desconhecidos.

Analysis of Three Intrusion Detection System Benchmark Datasets Using Machine Learning Algorithms (KAYACIK; ZINCIR-HEYWOOD, 2005) é dito que o uso de algoritmos de aprendizado de máquina para análise de tráfego de rede representa uma abordagem significativa, e este trabalho empregou dois algoritmos específicos - um algoritmo de *clustering* e uma Rede Neural - para analisar o tráfego de rede registrado a partir de três fontes distintas. Dessas fontes, duas foram sintéticas, indicando que o tráfego foi gerado em um ambiente controlado para avaliação de detecção de intrusões. O principal objetivo da análise foi identificar as diferenças entre o tráfego sintético e o tráfego do mundo real, embora a metodologia detalhada neste trabalho possa ser aplicada para fins gerais de análise de rede. Além disso, o trabalho discute brevemente o framework utilizado para gerar uma das fontes de tráfego sintético.

Ao revisar trabalhos relacionados, é possível identificar outras pesquisas que exploraram o uso de Aprendizado de Máquina para análise de tráfego de rede. Trabalhos anteriores podem ter se concentrado em diferentes algoritmos, técnicas de pré-processamento, ou abordagens para lidar com tráfego sintético e do mundo real. Compreender como esses trabalhos abordaram questões semelhantes ou diferentes pode enriquecer a discussão sobre a escolha de algoritmos, o design de experimentos e as implicações dos resultados na detecção de intrusões e análise de

tráfego de rede em geral.

Improving one-class SVM for anomaly detection (LI et al., 2003) relata, com o crescimento extraordinário da Internet, a segurança dos Sistemas de Informação tornou-se uma questão de séria preocupação global devido à rápida conexão e acessibilidade. O desenvolvimento de métodos eficazes para detecção de intrusões é, portanto, uma tarefa urgente para garantir a segurança de computadores e sistemas de informação. Dado que a maioria dos ataques e abusos pode ser identificada por meio da análise de arquivos de log do sistema e da análise de padrões neles contidos, uma abordagem para detecção de intrusões pode ser construída com base nesses dados.

Inicialmente, realizasse uma análise profunda dos padrões de ataques e abusos nos arquivos de *log*. Em seguida, propomos uma abordagem que utiliza máquinas de vetores de suporte para detecção de anomalias. Trata-se de uma abordagem baseada em SVM de uma classe, treinada com dados abstratos de *logs* de usuários de 1999 do DARPA. Este método visa identificar comportamentos anômalos que possam indicar atividades suspeitas, contribuindo assim para fortalecer a segurança dos sistemas de informação diante das ameaças crescentes no ambiente online.

3.1 Síntese dos trabalhos relacionados

Os trabalhos relacionados revelam uma riqueza de abordagens e contribuições em vários campos da Ciência da Computação. Dentro desse cenário diversificado, destacam-se estudos que exploram profundamente desafios cruciais, desde o aprimoramento do desempenho em supercomputadores até a otimização de compiladores para alcançar melhores níveis de eficiência. A Programação Paralela, especialmente através do OpenMP, é analisada em detalhes, assim como a avaliação meticulosa de *frameworks* dedicados a esse domínio.

A predição de desempenho em CPUs, juntamente com a introdução de uma visão multidimensional em linguagem C, evidencia a constante busca por inovação e aprimoramento no âmbito da computação de alto desempenho. Além disso, a análise abrangente da eficiência energética em *data centers* e sistemas HPC reflete uma preocupação crescente com a sustentabilidade e o impacto ambiental.

Destaca-se também a modelagem avançada do consumo de potência e energia em *kernels* de HPC, que utiliza redes neurais artificiais e indica uma tendência em direção à aplicação de técnicas de aprendizado de máquina para otimizar esses aspectos críticos.

4 METODOLOGIA

Este capítulo é dedicado a descrever os componentes e os processos empregados ao longo do desenvolvimento deste estudo.

No decorrer do capítulo, serão apresentadas e detalhadas as diferentes partes constituintes, desde as ferramentas e tecnologias utilizadas até os métodos empregados para coleta e análise de dados. Essa descrição busca fornecer uma compreensão do ambiente e das etapas essenciais que compuseram o cenário de pesquisa.

Além disso, são discutidos os critérios de seleção e a justificativa por trás das escolhas feitas em relação aos componentes e processos adotados. Este capítulo não apenas esclarece as decisões tomadas, mas também fornece uma base sólida para a replicação do estudo e a compreensão dos fatores que influenciaram seu desenvolvimento.

Dessa forma, ao final deste capítulo, espera-se que o leitor tenha uma visão clara e detalhada dos elementos que contribuíram para a condução deste estudo, estabelecesse assim o contexto necessário para a compreensão plena dos resultados apresentados.

4.1 Componentes de Hardware

O sistema computacional empregado para a execução deste estudo está equipado com uma unidade central de processamento (CPU) Intel Core i7 5500U. Este processador apresenta uma frequência base de operação de 2400 MHz (2.4 GHz) e pode operar em modo turbo que atinge uma frequência de 3000 MHz (3.0 GHz). Composta por dois núcleos físicos e quatro threads, a CPU foi fabricada na litografia de 14 nm e pertence à microarquitetura Haswell [1].

O processador em questão possui três níveis distintos de memória cache que totaliza 4 MB. A distribuição desse cache abrange o nível 1 (L1) com 64 KB de uso interno, o nível 2 (L2) com 256 KB e o nível 3 (L3), que compartilha uma memória de 4 MB.

No que diz respeito à memória de acesso randômico (RAM), o sistema utiliza uma capacidade total de 10 GB. Essa memória RAM é baseada na arquitetura DDR3L, uma variante da DDR3 com ênfase na eficiência energética, indicada pelo "L"em DDR3L, que significa "Low-Voltage". Adicionalmente, a memória opera a uma frequência de 1600 MHz (1.6 GHz) que contribui para um desempenho eficiente e econômico em termos de energia.

[1] Arquitetura Haswell: Microarquitetura de processadores da Intel, lançada em 2013 como a quarta geração da família Intel Core, notável por melhorias na eficiência energética, desempenho gráfico e suporte a instruções avançadas.

4.2 Componentes de Software

4.2.1 Sistema Operacional utilizado

No âmbito acadêmico e para fins de pesquisa, o Sistema Operacional mais amplamente adotado é o Linux. Este sistema destaca-se como uma escolha altamente recomendada quando se busca desempenho em investigações científicas.

A popularidade do Linux no meio acadêmico é atribuída, em grande parte, à sua vasta gama de ferramentas disponíveis que abrange diversas áreas do conhecimento, com especial ênfase na computação. Considerada esta versatilidade que a escolha por este sistema foi realizada. Vale ressaltar que o Linux não se limita a um sistema operacional único; existem diversas distribuições, cada uma com propósitos distintos, projetadas para atender às necessidades específicas de diferentes perfis de usuários.

Para este trabalho, a distribuição escolhida foi o Linux Ubuntu. Essa decisão não foi baseada em especificidades técnicas, mas sim na familiaridade com este sistema. A escolha do Ubuntu, entre as várias opções disponíveis, foi motivada pelo conhecimento prévio da equipe de pesquisa que considera o aprendizado de uma distribuição diferente não se mostrava tão relevante para os objetivos do estudo.

O sistema operacional Linux Ubuntu empregado nesta pesquisa está na versão 22.04.1 LTS e utiliza a arquitetura de 64 bits. Esta versão foi selecionada em virtude de sua estabilidade e do suporte a longo prazo, aspectos cruciais para ambientes acadêmicos e pesquisas científicas que demandam consistência e confiabilidade em suas operações.

4.2.2 Escolha do *Benchmark*

Este estudo tem como principal objetivo realizar uma análise comparativa entre compiladores que visa obter uma compreensão mais aprofundada de suas vantagens em contextos diversos. Para alcançar esse propósito, foram conduzidas pesquisas de *benchmarks* que atendessem a requisitos essenciais para esta investigação. Estes requisitos foram orientados para garantir uma abordagem facilitada, flexibilidade nos testes e a capacidade de realizar coleta de dados em diversos campos de pesquisa, além da tradicional Álgebra Linear. Adicionalmente, era desejável que os *benchmarks* fossem predominantemente implementados na linguagem C.

Inicialmente, o suíte de *benchmarks* escolhido para a coleta de dados foi o Parsec-Ompss. No entanto, essa escolha mostrou-se inviável devido à sua natureza, uma vez que a maioria dos *benchmarks* incluídos no Parsec-Ompss utilizava inputs gerados externamente. Esses *inputs* dependiam de ferramentas exógenas que demandavam um tempo considerável para gerar um arquivo relativamente pequeno, o que impactava significativamente o desempenho do *benchmark* em questão. Após a exclusão desse suíte, uma nova pesquisa foi conduzida para encontrar um conjunto de benchmarks compatível com a proposta apresentada. Nesse contexto, surgiu a

consideração de trabalhar com o suíte *NAS Parallel Benchmark*.

O *NAS Parallel Benchmark*, inicialmente atrativo devido à diversidade de campos em sua biblioteca, revelou-se inadequado devido à falta de flexibilidade necessária para dar continuidade à pesquisa. Esses *benchmarks* foram desenvolvidos com a premissa de coleções limitadas de *inputs* pré-programadas, incorporadas diretamente em seu código-fonte. Isso dificultou a realização de modificações sem prejudicar a performance e, adicionalmente, pelo menos metade dos benchmarks estavam codificados em Fortran, o que apresenta um obstáculo adicional.

Após uma nova busca, a suíte de *benchmarks* Rodinia foi identificada como uma alternativa viável. Esta suíte abrange *benchmarks* de diversos campos, ou "Dwarves", que vão desde Álgebra Linear até Aprendizado de Máquina aplicado ao treinamento de ferramentas médicas. Além disso, esses *benchmarks* destacam-se por sua alta flexibilidade em relação às entradas que permite a geração de processos com uma variedade de valores para sobrecarregar adequadamente os componentes do sistema, que atende assim às características fundamentais para o desenvolvimento desta pesquisa.

Dentro deste suite de *benchmarks*, foram selecionados cinco, de categorias distintas para ampliar a margem de comparação em varios cenários diferentes, *benchmarks* estes que são:

- ***LU DECOMPOSITION*:** Implementação da decomposição LU, um algoritmo de álgebra linear, utilizado para fatoração de uma matriz em produto de uma matriz triangular inferior e uma matriz triangular superior;
- ***BACK PROPAGATION*:** *Benchmark* relacionado à rede neural, especificamente à retropropagação, uma técnica fundamental no treinamento de redes neurais para aprendizado supervisionado;
- ***STREAMCLUSTER*:** Algoritmo de agrupamento (*clustering*) utilizado para analisar grandes conjuntos de dados, identificando padrões e agrupando dados similares;
- ***LAVA MD*:** Implementação do método de Dinâmica Molecular, um benchmark comum em simulações computacionais para estudar o comportamento de partículas em um sistema;
- ***MYOCYTE*:** *Benchmark* voltado para simulações relacionadas à Biologia Computacional, em particular, modelando o comportamento de miócitos, células musculares cardíacas.

4.2.3 Compiladores selecionados

Com a seleção do sistema operacional e do suíte de *benchmarks* estabelecida, a próxima etapa envolve a consideração de diferentes compiladores para conduzir esta pesquisa. Naturalmente, o primeiro compilador escolhido é o compilador nativo do C, o GCC. Este compilador é amplamente utilizado em sistemas Linux e similares, uma vez que é frequentemente pré-instalado

no sistema. Caso não esteja presente, a instalação pode ser realizada através do terminal com o comando:

```
$ user:-$ sudo apt install build-essential
```

A versão utilizada para este estudo foi a 11.4.0.

Como segundo compilador, propõe-se utilizar a plataforma LLVM, desenvolvida com foco na otimização em tempo de execução durante a compilação, com o Clang que atua como "*frontend*"para acessar suas ferramentas na compilação da linguagem C. A instalação do compilador na sistema pode ser efetuada através dos comandos:

```
$ user:-$ sudo apt-get update
$ user:-$ sudo apt-get install clang
```

Para compilar utilizasse o paradigma OpenMP com o Clang, é necessário substituir a "flag- *fopenmp* pela biblioteca libiomp5. A ativação desta biblioteca requer a instalação de suas dependências com o comando:

```
$ user:-$ sudo apt-get update
$ user:-$ sudo apt-get install libiomp5
```

A versão utilizada para esta pesquisa foi a versão 14.0.0.

Por último, o compilador selecionado para este estudo foi o *Intel C++ Compiler* [2] (ICC ou ICL), desenvolvido pela Intel para otimizar o desempenho em arquiteturas Intel. Apos realizar a instalação do executável, para utiliza-lo, é necessário usar um comando via terminal para ativar o compilador:

```
$ user:-$ source /opt/intel/oneapi/setvars.sh intel64
```

A versão do compilador utilizada nesta pesquisa foi a 2021.10.0 20230609. Essa escolha se alinha com a proposta de explorar diferentes compiladores para avaliar seu impacto no desempenho dos *benchmarks* selecionados e promover uma análise abrangente neste estudo comparativo.

4.2.4 Compilação dos *benchmarks*

Para efetuar a compilação dos *benchmarks* com diferentes compiladores, foi imperativo adaptar os seus*Makefiles*, que originalmente empregavam o GCC como compilador padrão, para possibilitar a compilação também com o Clang e o ICC. Nesse contexto, houve a necessidade não apenas de substituir o GCC pelos compiladores pertinentes, mas também de ajustar as flags de compilação associadas ao paradigma paralelo, como exemplificado a seguir:

[2] A instalação deste compilador exige o download de um executável no site oficial da Intel ou no seguinte link: Download do ICC.

```
PREFIX=${PARSECDIR}/pkgs/kernels/streamcluster/inst/${PARSECPLAT}
TARGET_C = sc_cpu
TARGET_O = sc_omp

ifdef version
  ifeq "$(version)" "parallel"
    CXXFLAGS := $(CXXFLAGS) -DENABLE_THREADS -pthread
  endif
endif

all: cpu omp

cpu:
#$(CXX)
clang++ $(CXXFLAGS) $(LDFLAGS) streamcluster_original.cpp
    -o $(TARGET_C) $(LIBS) -DENABLE_THREADS -pthread

omp:
clang++ -O3 -libiomp -o $(TARGET_O) streamcluster_omp.cpp

clean:
rm -f *.o *~ *.txt sc_cpu sc_omp
```

Para cada *benchmark* selecionado dentre os cinco anteriormente mencionados, realizou-se o processo de compilação com os *Makefiles* adaptados para interação com os compiladores escolhidos. Esse procedimento resultou em quinze binários distintos, que foram preparados para avaliação. Cabe destacar que foram necessárias modificações sutis no código-fonte dos *benchmarks*, as quais serão detalhadas em uma próxima subseção.

4.2.5 Dando Permissão ao Intel RAPL

Para a realização da aquisição do consumo energético, tornou-se imperativo conceder permissões para a utilização do Intel RAPL (Running Average Power Limit) por meio de um script externo denominado "Joule it", o qual será discutido na seção subsequente. A viabilização desse procedimento demanda a instalação de um pacote denominado "sysfsutils", conforme ilustrado a seguir:

```
$ user:-$ sudo apt-get update
$ user:-$ sudo apt install sysfsutils
```

Após a instalação deste pacote, é igualmente crucial realizar modificações em um arquivo interno, localizado em:

```
$ user:-$ nano /etc/sysfs.conf
```

E, em seguida, é necessário adicionar esta linha de comando ao arquivo do sistema:

```
mode class/powercap/intel-rapl:0/energy\_uj = 0444.
```

Ao executar esses procedimentos, torna-se imperativo reiniciar o sistema para consolidar as modificações. Após a reinicialização, para, enfim, ativar essa funcionalidade do sistema, é necessário inserir o seguinte comando no terminal:

```
$ user:-$ sudo chmod -R a+r /sys/class/powercap/intel-rapl
```

Cada etapa necessária para conceder permissões ao uso do Intel RAPL está detalhadamente documentada no repositório GitHub mantido por Benoit Courty (COURTY, 2021). O referido repositório fornece um guia passo a passo, abordando procedimentos específicos para garantir a configuração adequada e a concessão de permissões necessárias ao Intel RAPL. A consulta a esse recurso se revela essencial para uma implementação eficaz e compreensão abrangente do processo envolvido na ativação do Intel RAPL em ambientes de pesquisa e desenvolvimento.

4.2.6 *Script* de Coleta e Joule It

Para a condução da coleta de dados neste estudo, elaboraram-se *scripts* em *Bash*, emprega a API de monitoramento de consumo conhecida como Joule It da Power API. Este script é configurável para a coleta de parâmetros como *CORE, CPU, DRAM, DURATION, UNCORE* e *EXIT CODE*, expressos em unidades de microjoules e microssegundos, respectivamente.

Tabela 1 – Exemplificação do arquivo resultante da coleta padrão do Joule It

CORE	CPU	DRAM	DURATION	UNCORE	EXIT CODE
122109002	168557613	41240983	18690949	5677	0
122237969	168562862	41457048	18663982	5493	0
121223689	167233703	40950091	18554820	4028	0
123840015	170802175	42154372	18922905	5310	0
122941458	169784966	41711075	18906358	5371	0
...	...	...	...	...	...

Autoria própria.

É crucial destacar que, para o pleno funcionamento do Joule It, é imperativo conceder permissões no terminal por meio dos seguintes comandos:

```
$ user:-$ chmod +x jouleit.sh
```

O script para a coleta de dados foi elaborado em *Bash*, executa os binários do *benchmark* em conjunto com o *script "jouleit.sh"*, responsável pelo monitoramento do consumo energético e pela gravação dos resultados em um arquivo CSV. Além disso, foi necessário excluir linhas que exibiam qualquer tipo de texto no terminal no código fonte, já que isso acaba por desencadear uma série de erros durante a coleta de informações.

O método de coleta foi estabelecido da seguinte maneira: foi determinado um valor máximo de entrada, garante assim que o sistema não seja sobrecarregado a ponto de travar, mas permanecesse próximo desse limite, servindo como uma carga máxima para o sistema, respeitando valores múltiplos de dois. Após a identificação empírica desse valor, uma abordagem foi adotada dividindo-o pela metade para obter o valor inicial. Subsequentemente, foram realizados incrementos de cento e vinte e oito passos até alcançar o valor máximo desejado, resultando em 128 amostras para a análise deste estudo. Este método pode ser descrito da seguinte maneira:

$$\frac{V_{\text{max}} - V_{\text{min}}}{128} = \text{Value} \tag{4.1}$$

Este método revelou-se eficaz na coleta de dados para quatro dos cinco *benchmarks* selecionados. No entanto, para o *benchmark* LavaMD, lamentavelmente, não foi possível reunir 128 amostras devido à sua natureza peculiar. Nessa circunstância, foi viável obter apenas 90 amostras, as quais, ainda assim, se mostraram suficientes para análise. A seguir, representa-se o *script* em *bash* utilizado para realizar a coleta:

```
cc = "Compilador" % Esta parte foi utilizada
para diferenciar qual binário está sendo
utilizado no momento de gerar o arquivo.csv

output=$(./jouleit.sh -l)
output="$output;INPUT"
output=${output//$'\n'/,}
echo $output >> bp_${cc}.csv

echo "Run BACKPROP"
for i in {16777216..33554432..131072}
do
output=$(./jouleit.sh -c ./backprop ${i})
output="$output;$i"
output=${output//$'\n'/,}
echo $output >> bp_${cc}.csv
echo $output
done
```

Configurando a coleta da seguinte forma:

- Back Propagation varia de 16.777.216 até 33.554.432, incrementando 131.072 em cada iteração;
- LavaMD varia de 1 até 90, incrementando 1 em cada iteração;
- Lud Decomposition varia de 16.384 até 32.768, incrementando 128 em cada iteração;
- Myocyte varia de 512 até 1.024, incrementando 4 em cada iteração;
- Stream Cluster varia de 2.097.152 até 4.194.304, incrementando 16.384 em cada iteração.

Isso resulta em quinze conjuntos de dados CSV, cada um com a inclusão de uma coluna adicional denominada *INPUT*, que proporciona controle durante a análise desses dados.

Tabela 2 – Exemplificação do arquivo CSV gerado no Script de coleta

CORE	CPU	DRAM	DURATION	UNCORE	EXIT CODE	INPUT
73647760	91968454	10552097	9812865	2503	0	512
27366324	34201390	3997121	3636782	1587	0	516
74480400	92856879	10615390	9855222	2990	0	520
31772136	39662802	4590259	4200268	1648	0	524
32970985	41178301	4775988	4366487	1953	0	528
21445746	26863213	3161369	2860645	1343	0	532
20357247	25481930	3005851	2706413	1221	0	536
46353581	57772862	6602156	6073955	1953	0	540
...	...	...	...	...	...	...

Autoria própria.

4.2.7 *Script* de Coleta do tamanho dos Binários

Além dos *scripts* desenvolvidos em *Bash* para efetuar a coleta das informações de consumo, conforme detalhado na seção 4.2.6, foi elaborado de maneira complementar um *script* em Python. Esse script tem como finalidade realizar a coleta do tamanho dos arquivos, proporcionando assim a criação de um conjunto de dados destinado à análise subsequente.

Tabela 3 – Exemplificação dos Dados de Tamanho dos Arquivos

Nome do Arquivo	Tamanho (KB)
lud_omp	21.828125
coleta.py	1.2353515625
backprop	49.8359375
lavaMD	20.734375
sc_omp	40.671875
myocyte.out	64.8984375

Autoria própria.

4.2.8 Pré-Processamento dos dados Coletados

Ao realizar a coleta descrita no tópico anterior (4.2.6), foram gerados quinze conjuntos de dados CSV distintos destinados à análise deste estudo. Com os valores coletados em unidades de microjoules e microssegundos, implementou-se uma função para converter esses dados para a unidade de joules e segundos, visasse uma melhor visualização das informações. Adicionalmente, foi necessário criar uma nova métrica no conjunto de dados denominada *ENERGY*, cujo valor corresponde à soma das métricas *CPU, DRAM* e *UNCORE*, tornasse assim uma métrica que representa o consumo energético total dos principais componentes do sistema. Para uma visualização preliminar, a métrica *EXIT_CODE*, que se refere a possíveis erros que podem ocorrer durante a coleta de dados, foi excluída. No entanto, esta métrica será empregada posteriormente para análise comparativa.

Tabela 4 – Exemplificando Dataset resultante após tratamento com Pandas

Index	CORE	CPU	DRAM	DURATION	UNCORE	INPUT	ENERGY
0	126.44	166.85	44.05	18.76	0.07	16777216	210.97
1	127.59	167.97	44.2	18.84	0.07	16908288	212.24
2	128.62	169.32	44.69	19.06	0.07	17039360	214.08
3	128.71	169.67	44.82	19.15	0.07	17170432	214.56
4	129.95	171.25	45.34	19.33	0.07	17301504	216.66
...	...	...	...	...	...	...	...

Autoria própria.

4.2.9 Análise Exploratória

Para a análise, utilizou-se a ferramenta *Google Colaboratory*, baseada em *Jupyter* para a criação dos *Notebooks*. A linguagem empregada na codificação dos scripts de análise é o Python. Para auxiliar neste estudo, foram utilizadas as bibliotecas Pandas, NumPy, Scikit-Learn, Seaborn e Matplotlib respectivamente.

A biblioteca **Pandas** é empregada para carregar e ajustar o conjunto de dados, visando uma posterior utilização otimizada, assim como para manipular e limpar eventuais dados indesejados. O **Numpy** será utilizado para realizar refatorações matemáticas nos dados, proporcionando estatísticas mais precisas sobre as informações coletadas. **Scikit-Learn** foi utilizado para treinamento de aprendizado de máquina afim de viabilizar uma melhor visualização de possiveis anomalias nos dados.

Em seguida, **Seaborn**, construído sobre as bases do **Matplotlib**, expande as capacidades de visualização de informações. Ao utilizar o **Seaborn**, o **Matplotlib** entra em cena para representar visualmente os dados, que proporciona uma análise mais aprofundada por meio de gráficos significativos e esteticamente agradáveis.

4.2.9.1 Consumo Total de Execução

Ao analisar o consumo energético total, obtém-se uma avaliação abrangente dos impactos energéticos decorrentes da execução dos *benchmarks*. Além disso, é possível observar o perfil matemático associado a esse consumo e entender como ele se comporta em cada iteração realizada. Essa abordagem científica permite uma compreensão mais aprofundada não apenas dos aspectos quantitativos do consumo energético, mas também da dinâmica matemática subjacente.

4.2.9.2 Tempo Total de Execução

O tempo dedicado a cada execução apresenta uma correlação direta com o desempenho e o consumo energético. Esta relação é fundamentada na premissa de que, para cada problema abordado, um maior tempo de execução implica em uma demanda energética proporcionalmente superior do sistema para a resolução do algoritmo em questão. Portanto, é de suma importância analisar de maneira conjunta o comportamento temporal e o consumo energético total. Essa abordagem integrada permite uma compreensão mais profunda dos aspectos temporais e energéticos.

4.2.9.3 Uma Comparação detalhada

Ao conduzirmos uma comparação entre os *benchmarks* compilados por diferentes compiladores, é possível identificar tendências marcantes nas relações de consumo e nas plataformas correspondentes. A análise comparativa desses dados foi realizada principalmente por meio do estudo de gráficos, permitindo uma avaliação visual e mais aprofundada das nuances presentes nos resultados.

Através da observação das tendências, torna-se possível discernir padrões de desempenho e eficiência de cada compilador em relação aos *benchmarks*d utilizados. Os gráficos fornecem uma representação visual que facilita a identificação de variações significativas e padrões consistentes, oferecendo informações sobre o impacto das escolhas de compiladores nas métricas de consumo.

4.2.9.4 Utilizando *Outliers* para identificar Anomalias

Enquanto a comparação dos *benchmarks* foi conduzida inicialmente utilizando gráficos como fonte primária de investigação, identificaram-se inconsistências nos dados, o que motivou uma pesquisa mais aprofundada sobre essas anomalias. Diante dessa situação, propôs-se a aplicação de técnicas de visualização de *outliers*, que primeiramente emprega o *Boxplot* como uma ferramenta concisa para avaliar quais *benchmarks* e/ou compiladores apresentavam essas anomalias estatísticas de maneira clara.

A utilização do *Boxplot* proporcionou uma visão intuitiva das distribuições estatísticas dos dados, permitindo identificar de forma destacada os pontos fora do padrão e as variações

significativas. Este método prévio foi crucial para uma avaliação inicial, delineando as áreas de interesse que demandavam uma análise mais aprofundada.

Posteriormente, para uma análise mais robusta e uma visualização mais refinada das anomalias, optou-se pela aplicação da técnica *One-Class SVM*. Esta abordagem permitiu uma compreensão mais detalhada das discrepâncias nos dados, destacando de forma mais precisa os padrões anômalos e fornecendo informações sobre os *benchmarks* e compiladores específicos que apresentavam comportamentos divergentes.

4.2.9.5 Avaliação Residual dos Binários

Uma abordagem adicional que pode fornecer informações substanciais para a análise comparativa empreendida neste estudo, é o impacto do resíduo gerado durante a compilação por diferentes compiladores sobre o desempenho final de cada *benchmark*. Além disso, uma avaliação direta do tamanho dos arquivos resultantes da compilação, correlacionando-os com o desempenho final das simulações, pode revelar dados cruciais sobre a influência da etapa de compilação no cenário global de execução. Essa abordagem científica ampliada permite uma compreensão mais aprofundada das relações entre as características do processo de compilação e o desempenho subsequente, que contribui significativamente para a análise comparativa abrangente realizada neste estudo.

5 RESULTADOS

Este capítulo dedica-se à análise dos resultados derivados das coletas realizadas, empregasse os métodos e métricas de avaliação previamente delineados na seção anterior. A abordagem metodológica adotada proporcionou uma estrutura rigorosa para a coleta de dados, enquanto as métricas de avaliação estabeleceram critérios objetivos para a análise e interpretação dos resultados obtidos.

As informações extraídas durante essas coletas não apenas oferecem informações sobre os fenômenos observados, mas também permitem uma avaliação crítica da eficácia dos métodos empregados. Este capítulo busca apresentar uma análise detalhada desses resultados, que destaca padrões emergentes, eventuais desafios enfrentados durante a coleta e a interpretação desses dados à luz das métricas estabelecidas.

Ao abordar os resultados, será possível contextualizar as descobertas dentro do quadro conceitual delineado no início da pesquisa. Este processo não apenas valida a robustez da metodologia adotada, mas também contribui para a construção de uma narrativa coerente e fundamentada cientificamente em torno dos resultados obtidos, ampliando, assim, a compreensão do fenômeno investigado.

5.1 Avaliando individualmente cada *Benchmark*

Nesta seção, promovosse uma análise minuciosa do comportamento individual de cada *benchmark*, considerasse os dados coletados a partir de três compiladores distintos. Este enfoque proporcionará uma compreensão aprofundada das nuances do desempenho de cada aplicação, que permite uma avaliação criteriosa das variações introduzidas pelos diferentes compiladores.

Os resultados obtidos desta investigação serão meticulosamente dissecados, com o objetivo de identificar padrões, tendências e possíveis disparidades entre os compiladores empregados. Esta abordagem detalhada visa não apenas destacar as características intrínsecas de cada *benchmark*, mas também discernir o impacto específico de cada compilador sobre o desempenho das aplicações estudadas.

Ao examinar o comportamento individual dos *benchmarks*, almejamos não apenas evidenciar diferenças quantitativas, mas também compreender qualitativamente as razões por trás dessas variações. Dessa forma, esta seção visa contribuir significativamente para uma análise aprofundada da influência dos compiladores no desempenho das aplicações.

5.1.1 Resultado: *Stream Cluster*

Ao analisar o gráfico que representa o consumo total do *Stream Cluster* para cada compilador, destaca-se imediatamente a distinção entre as compilações, mantendo, contudo,

uma tendência semelhante em ambos os casos. É evidente que, embora sigam uma trajetória comparável, há uma discrepância significativa nos níveis de consumo entre elas.

De maneira notável, o compilador ICC demonstra uma vantagem clara em relação aos outros dois compiladores, conforme ilustrado na Figura 1.

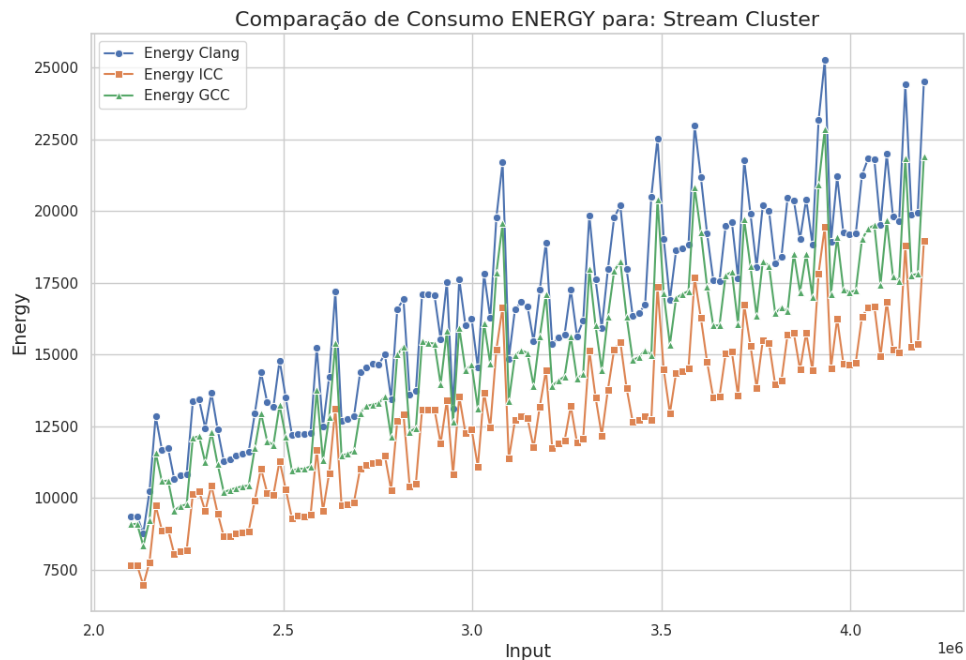

Figura 1 – Comparação de Consumo de Energia do Stream Cluster
Autoria própria.

Ao estabelecermos uma correlação entre o tempo de execução e o consumo total de cada compilador empregado no Stream Cluster, torna-se evidente uma relação direta entre essas variáveis, conforme destacado na Figura 2. Essa representação gráfica aponta para uma conexão intrínseca entre o tempo de execução e o consumo total.

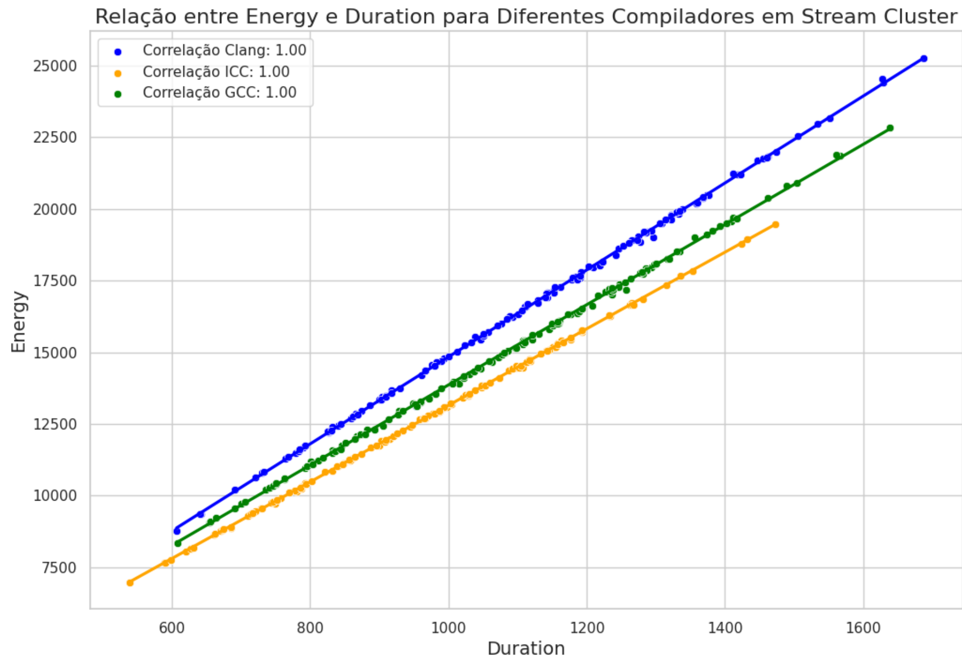

Figura 2 – Correlação entre duração e consumo energético total do Stream Cluster
Autoria própria.

Ao examinarmos as percentagens médias dos componentes individuais em relação ao consumo total, torna-se evidente que o Stream Cluster, de acordo com o propósito inicial do benchmark, é predominantemente consumidor de recursos da CPU em ambos os compiladores, conforme ilustrado na Figura 3. Este comportamento ratifica a concepção original do benchmark.

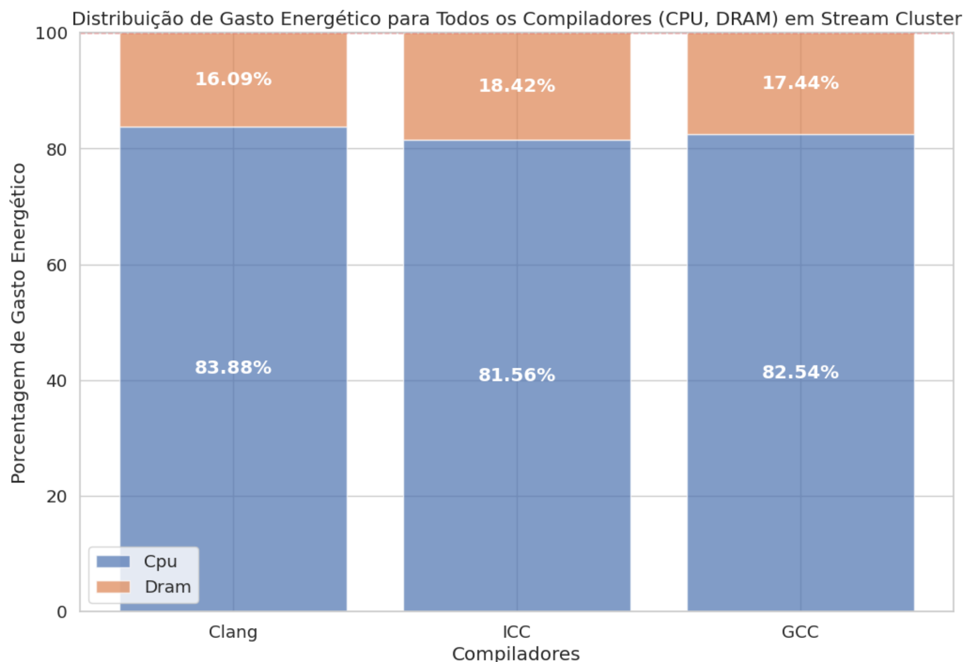

Figura 3 – Distribuição Média Percentual entre CPU e DRAM em relaçao ao consumo energético total do Stream Cluster
Autoria própria.

Adicionalmente, ao analisarmos as médias percentuais, notamos uma redução média de 15.13% no consumo pelo ICC em relação ao GCC e um aumento médio de 10.87% pelo Clang, conforme destacado na Figura 4. Estes dados oferecem uma perspectiva clara das discrepâncias nos padrões de consumo entre os compiladores, fornecendo uma base sólida para compreender as implicações dessas diferenças no desempenho global do Stream Cluster.

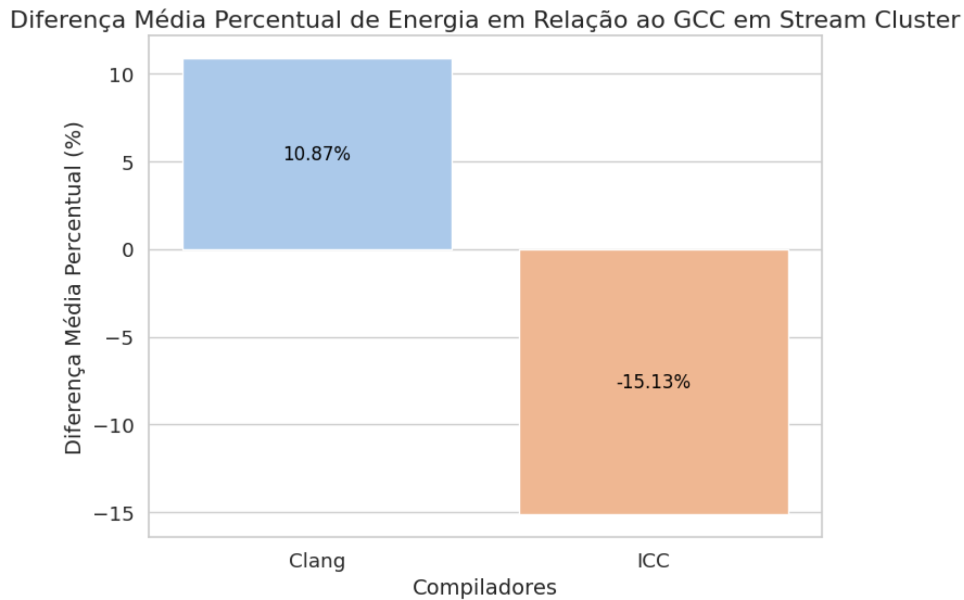

Figura 4 – Média Percentual de ganho de consumo em relação ao compilador GCC do Stream Cluster

Autoria própria.

5.1.2 Resultado: LU Decomposition

Ao analisar o gráfico comparativo do consumo total de energia para o *Benchmark Lu Decomposition*, emerge uma tendência exponencial. Entretanto, é pertinente ressaltar antecipadamente uma distinta diferença em seu fator de crescimento, expresso por b na função exponencial:

$$f(x) = a \cdot b^x$$

Observa-se claramente uma propensão a um maior crescimento de consumo no *benchmark* compilado em Clang em comparação aos demais, conforme evidenciado na Figura 5. Essa disparidade no fator de crescimento não só destaca a variação no padrão de consumo entre os compiladores, mas também fornece informações valiosas sobre o desempenho diferenciado do *Benchmark Lu Decomposition* sob diferentes ambientes de compilação.

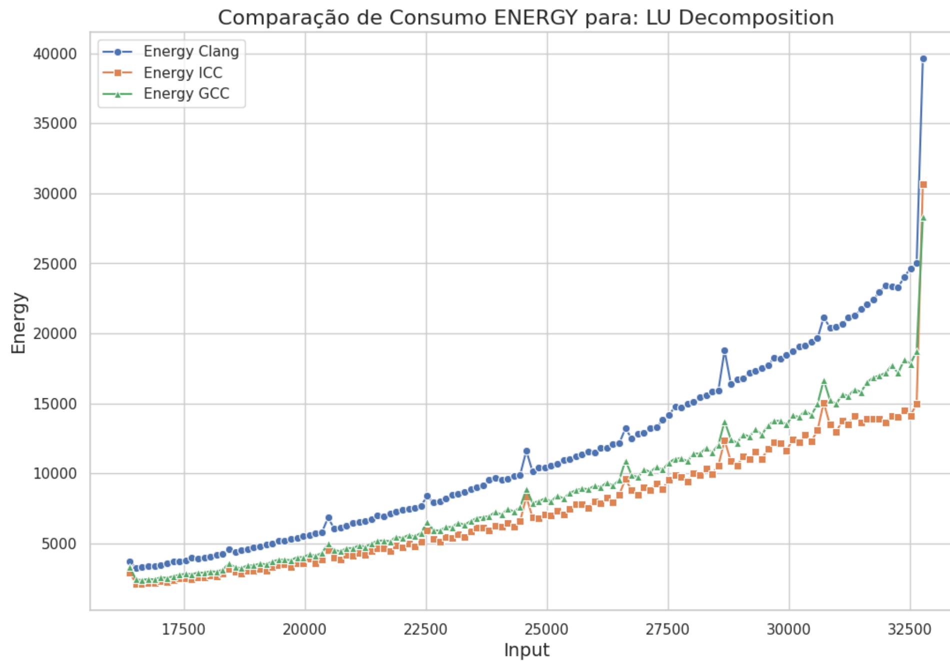

Figura 5 – Comparação de Consumo de Energia do LU Decomposition
Autoria própria.

Ao correlacionar o tempo de duração das execuções com seus consumos totais, observamos uma relação de 1 para 1 em suas compilações correspondentes, conforme destacado na Figura 6. Essa proporção direta sugere que, à medida que o tempo de execução aumenta ou diminui, o consumo total acompanha essa variação de maneira proporcional, indicando uma estreita correspondência entre essas duas métricas cruciais.

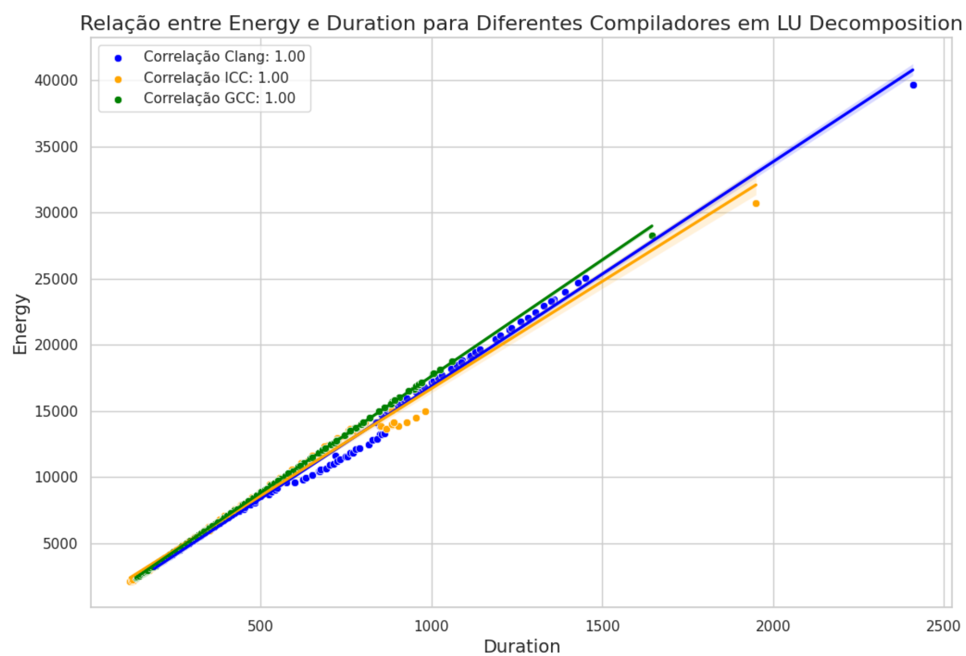

Figura 6 – Correlação entre duração e consumo energético total do LU Decomposition
Autoria própria.

Ao analisar a média percentual de consumo, percebemos que este *benchmark* consome predominantemente mais de 80% de CPU em todos os casos, conforme proposto de maneira evidente na Figura 7. Essa distribuição percentual enfatiza a carga significativa imposta à unidade de processamento central durante a execução do Benchmark Lu Decomposition.

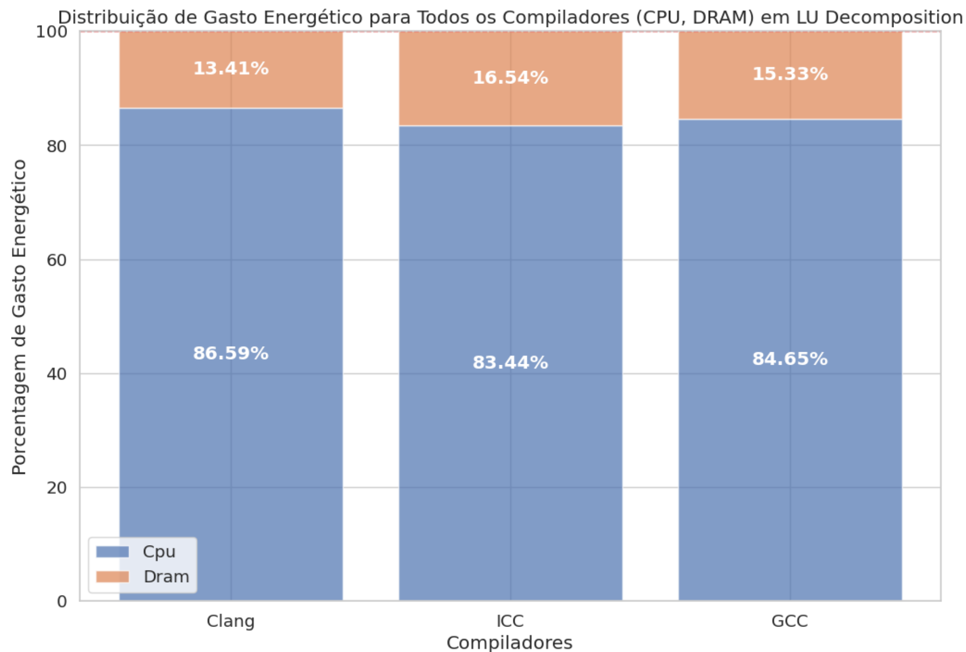

Figura 7 – Distribuição Média Percentual entre CPU e DRAM em relaçao ao consumo energético total do LU Decomposition

Autoria própria.

Ao concluir a comparação das médias percentuais globais de consumo em relação ao *benchmark* compilado em GCC, notamos que o Clang revela-se energeticamente 33.73% menos eficiente do que o ICC, enquanto este último apresenta um ganho de eficiência de 12.16%, conforme evidenciado na Figura 8. Essa análise ressalta claramente as disparidades substanciais no desempenho energético entre os compiladores avaliados, oferecendo uma perspectiva quantificada sobre a eficiência relativa dessas implementações no contexto específico do *Benchmark Lu Decomposition*. Essa compreensão detalhada das médias percentuais contribui para embasar decisões informadas sobre a escolha do compilador, considerando tanto o desempenho quanto o consumo energético.

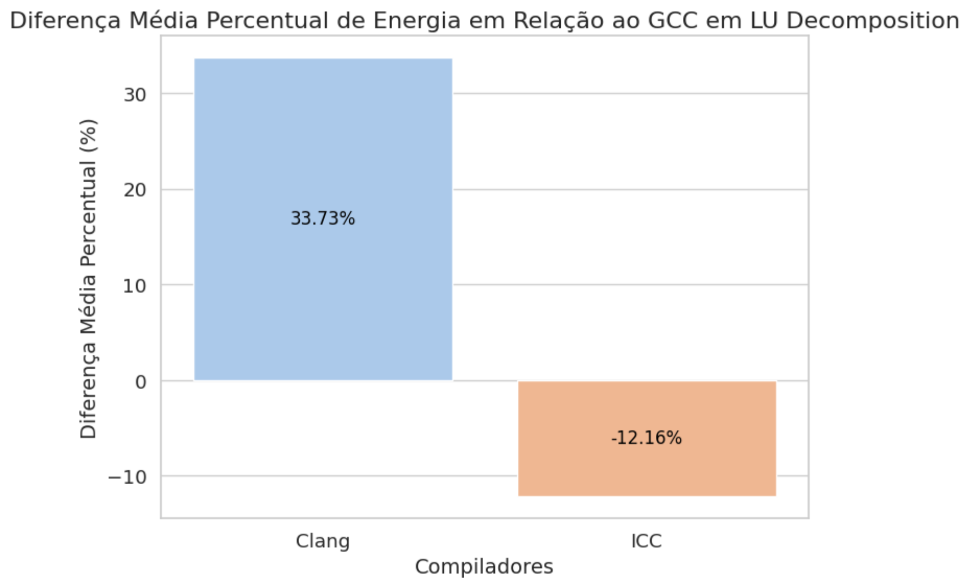

Figura 8 – Média Percentual de ganho de consumo em relação ao compilador GCC do LU Decomposition

Autoria própria.

5.1.3 Resultado: LavaMD

Ao observar a tendência do *Benchmark LavaMD*, percebesse um comportamento exponencial, como evidenciado pela função descrita na subseção anterior 5.1.2. No entanto, neste cenário específico, as compilações revelam-se notavelmente equivalentes, conforme ilustrado na Figura 9.

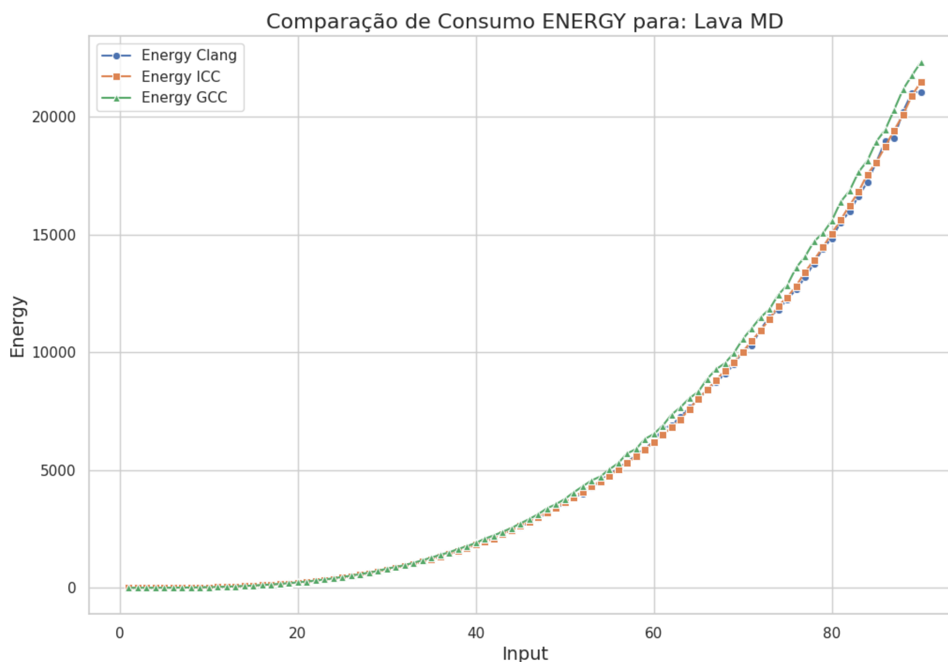

Figura 9 – Comparação de Consumo de Energia do LavaMD

Autoria própria.

Ao correlacionar as métricas de consumo total e duração de cada execução, torna-se evidente uma relação de equivalência no gráfico representado na Figura 10. Essa equivalência nas compilações do *Benchmark LavaMD* sugere que, apesar da natureza exponencial do comportamento, as implementações geradas pelos compiladores avaliados resultam em desempenhos e consumos energéticos comparáveis.

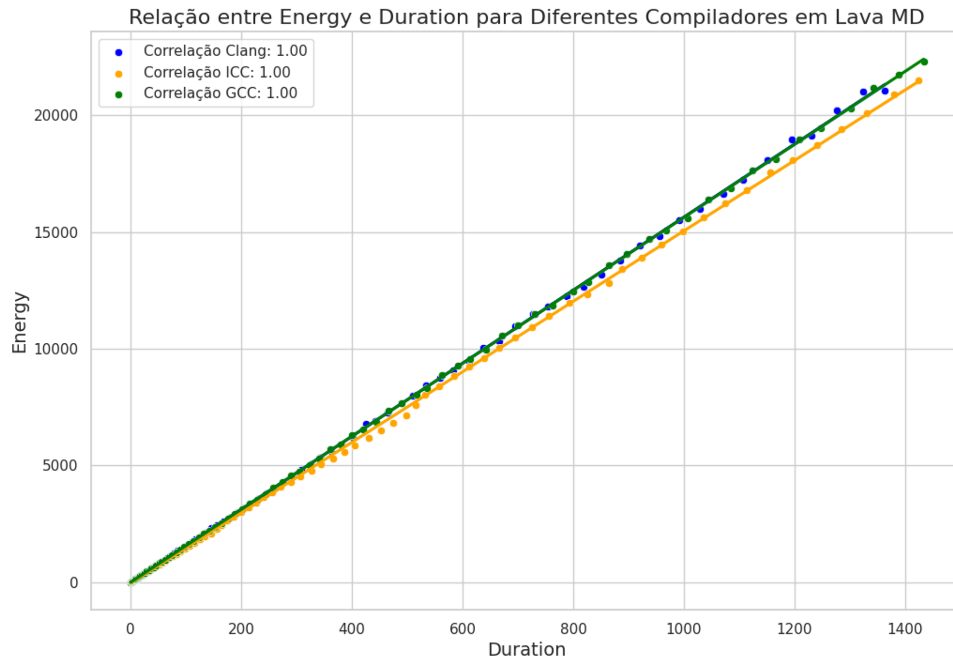

Figura 10 – Correlação entre duração e consumo energético total do LavaMD
Autoria própria.

A análise da distribuição média entre CPU e DRAM neste *benchmark* revela uma predominância significativa no uso de CPU, ultrapassando os 90%. Essa ênfase na carga computacional ressoa de maneira altamente positiva, alinhando-se à premissa central desses *benchmarks*, conforme ilustrado na Figura 11. Essa distribuição evidencia a natureza intensiva de CPU característica desse cenário de teste.

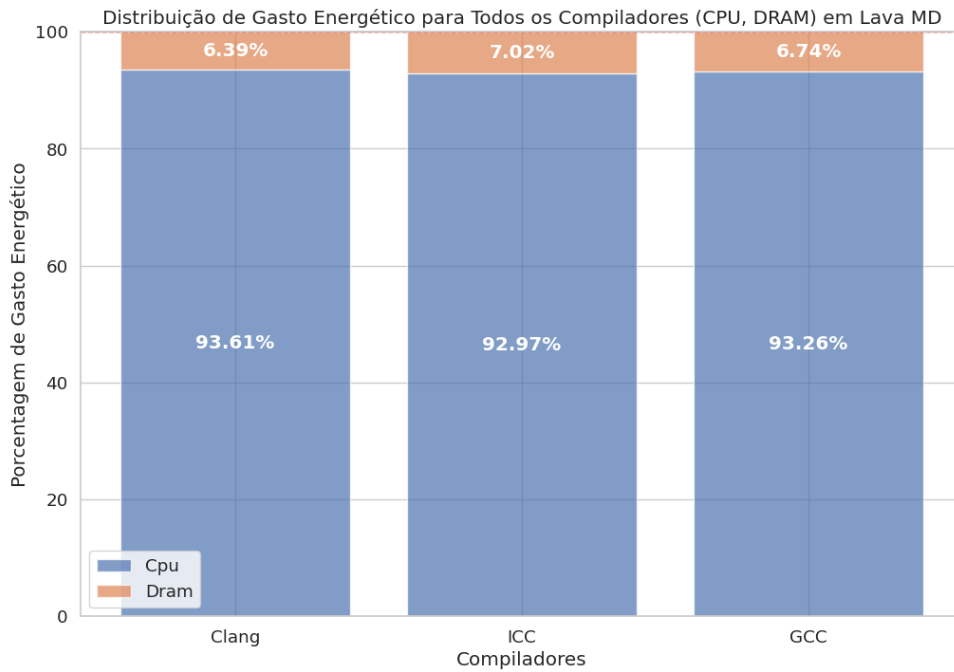

Figura 11 – Distribuição Média Percentual entre CPU e DRAM em relaçao ao consumo energético total do LavaMD

Autoria própria.

Ao direcionar nossa atenção para a média percentual do consumo total em relação à aplicação compilada em GCC, observamos melhorias notáveis na eficiência tanto para o Clang quanto para o ICC. O Clang exibe uma melhoria de eficiência de 3.41%, enquanto o ICC registra um aumento de 3.18%, conforme demonstrado de maneira elucidativa na Figura 12. Esses resultados sublinham otimizações eficazes nos compiladores, indicando uma utilização mais eficiente dos recursos durante a execução do *Benchmark LavaMD*.

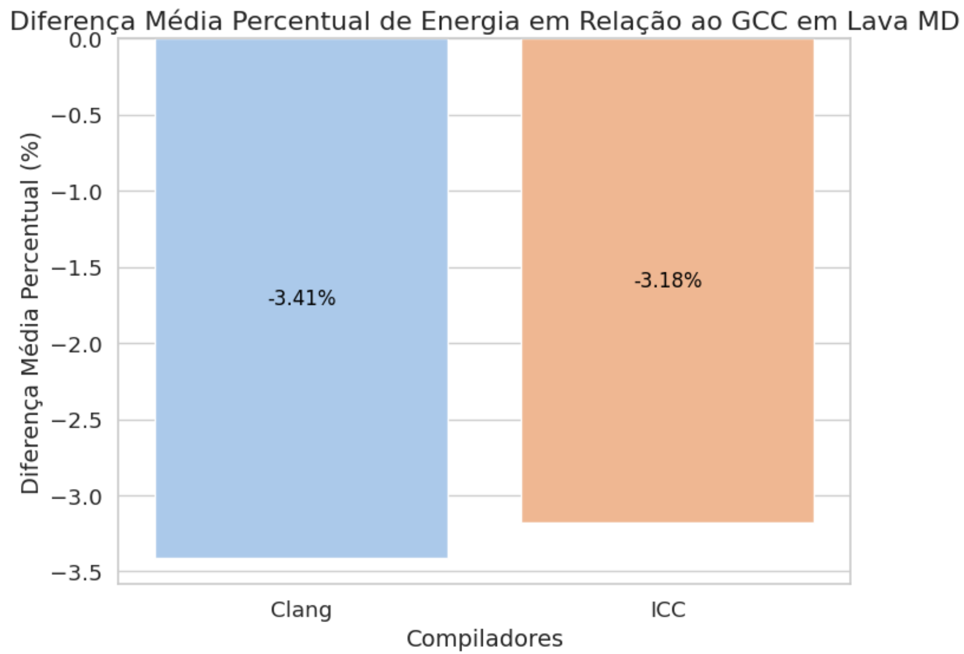

Figura 12 – Média Percentual de ganho de consumo em relação ao compilador GCC do LavaMD
Autoria própria.

5.1.4 Resultado: Back Propagation

Diferente dos resultados anteriormente abordados nas subseções 5.1.1, 5.1.2 e 5.1.3, os dados agora delineiam um perfil mais linear para o *Benchmark Back Propagation*. A Figura 13 destaca uma vantagem clara em eficiência para a compilação em Clang.

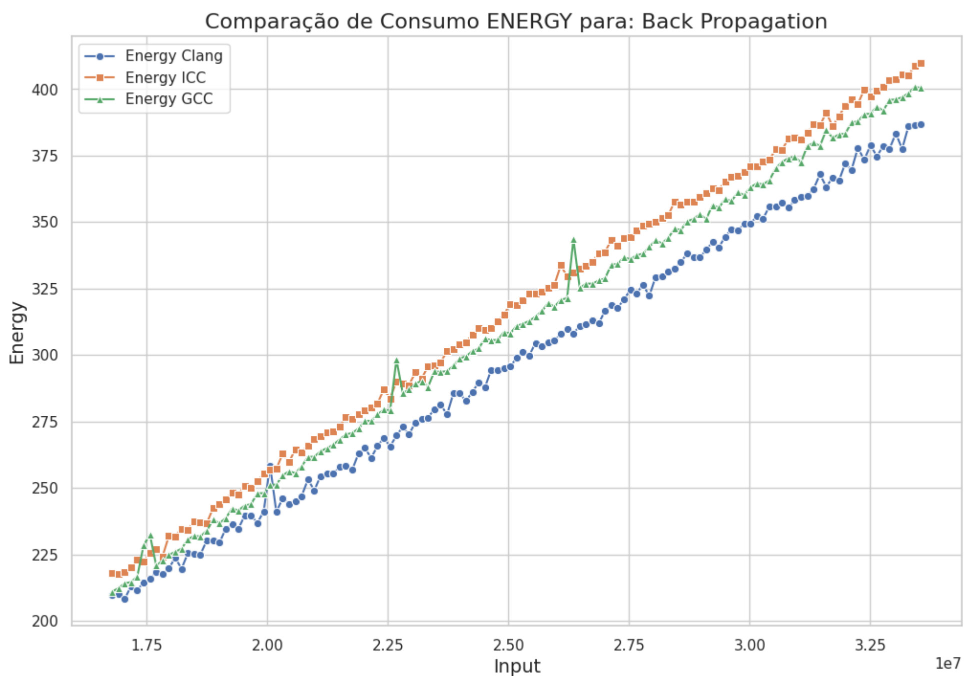

Figura 13 – Comparação de Consumo de Energia do Back Propagation
Autoria própria.

Notavelmente, é possível identificar uma correlação perfeita entre o tempo de execução e a métrica de consumo, conforme exemplificado no gráfico da Figura 14.

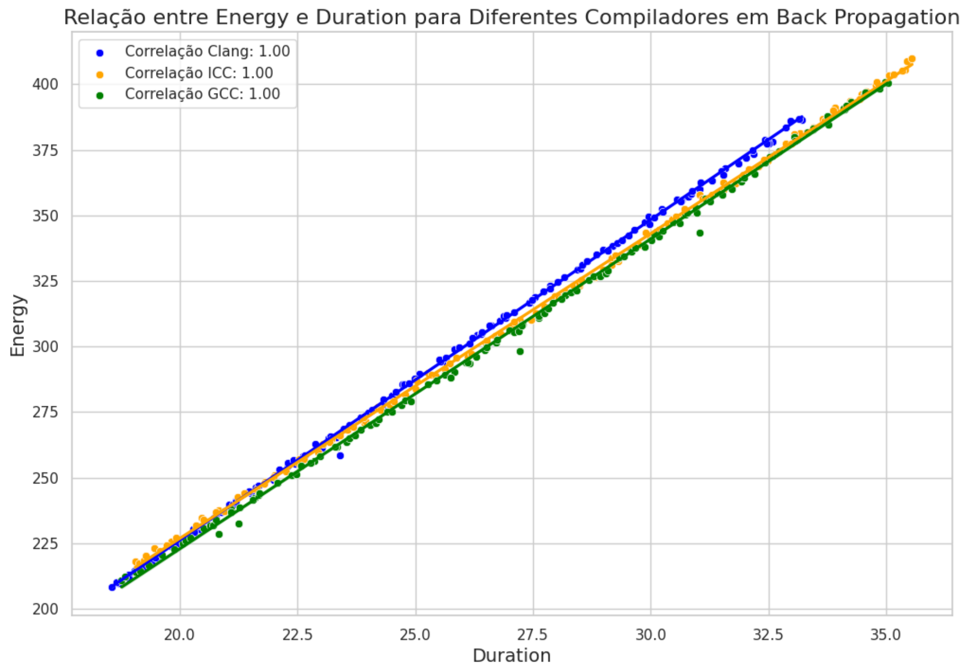

Figura 14 – Correlação entre duração e consumo energético total do Back Propagation
Autoria própria.

Apesar de operar ligeiramente abaixo da margem de 80% no uso de CPU em relação ao consumo total, conforme evidenciado na Figura 15, o *Benchmark Back Propagation* ainda se mantém dentro das expectativas típicas para um benchmark "*cpu bound*".

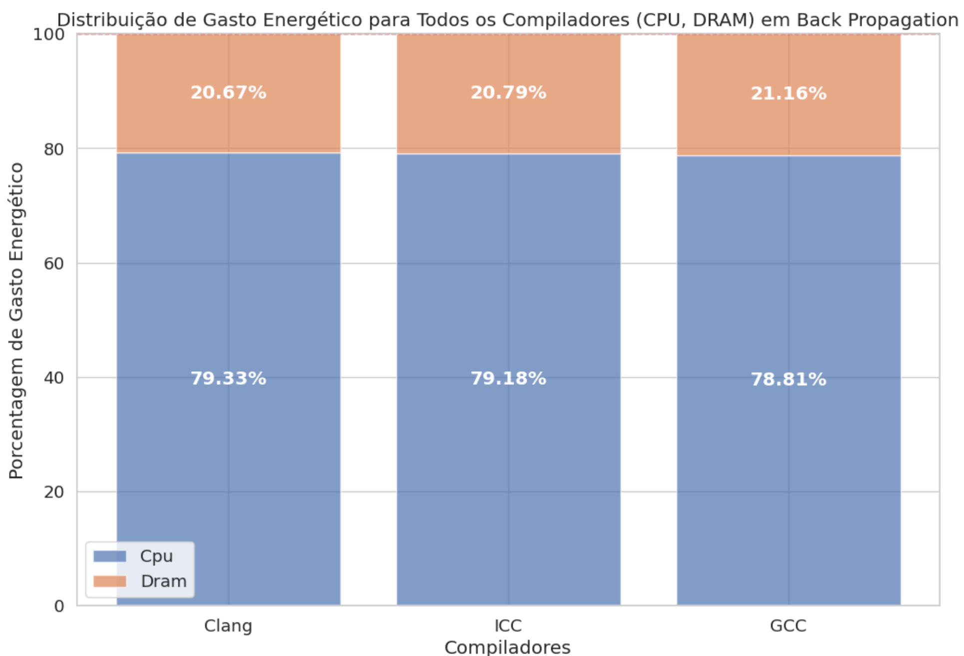

Figura 15 – Distribuição Média Percentual entre CPU e DRAM em relaçao ao consumo energético total do Back Propagation
Autoria própria.

Em análises prévias, destacou-se que o Clang demonstra uma eficiência superior neste contexto específico, apresentando uma eficiência 3.84% superior em comparação com o *benchmark* compilado em GCC. Em contraste, o ICC revelou-se 2.04% menos eficiente, conforme ilustrado de maneira elucidativa na Figura 16. Essa análise aprofundada enriquece a compreensão do comportamento singular do *Benchmark Back Propagation*.

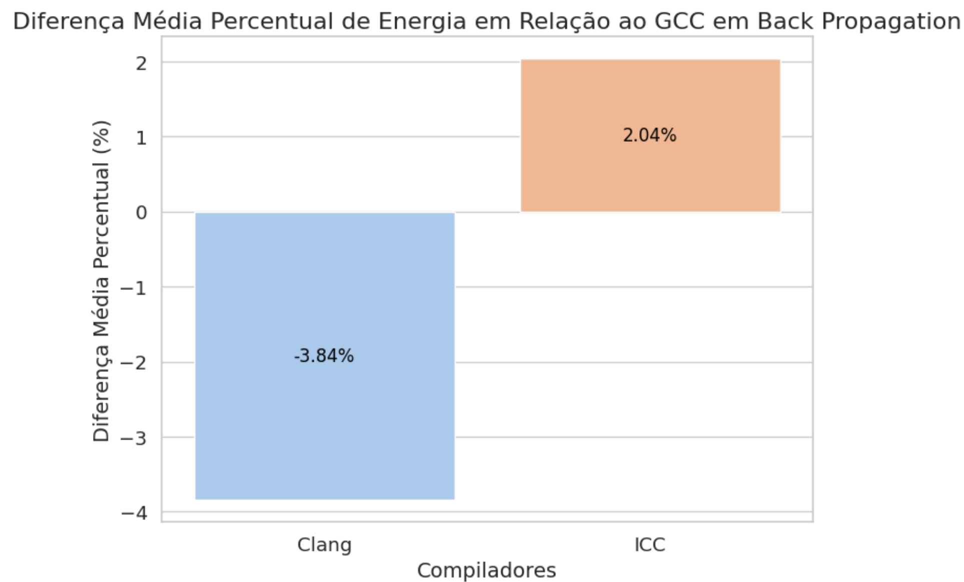

Figura 16 – Média Percentual de ganho de consumo em relação ao compilador GCC do Back Propagation

Autoria própria.

5.1.5 Resultado: Myocyte

Numa análise inicial, os resultados deste benchmark apresentam peculiaridades que despertam surpresa. Uma irregularidade estatística torna-se evidente, sobretudo nos inputs situados entre aproximadamente 660 e 760, tanto na compilação com Clang quanto na compilada com ICC. A Figura 17 ilustra de forma marcante essa discrepância, destacasse especialmente na compilação com ICC. Essa disparidade levanta suspeitas de incoerências que merecem uma investigação que será realizada de forma mais aprofundada na seção 5.2 para compreender os motivos por trás desse comportamento atípico nesse intervalo específico de inputs.

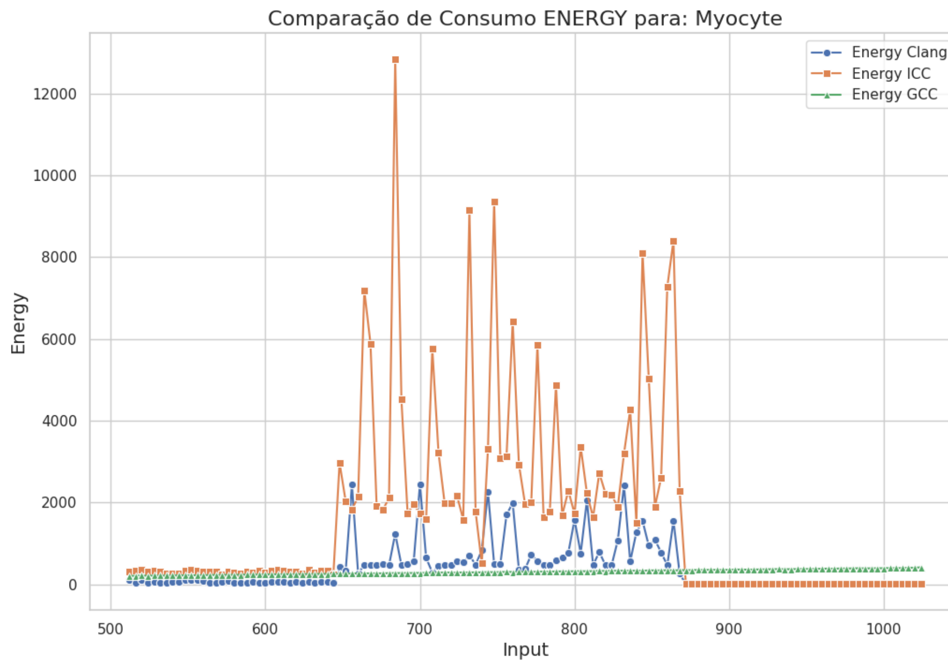

Figura 17 – Comparação de Consumo de Energia do Myocyte
Autoria própria.

Apesar da anormalidade observada no gráfico geral de consumo, é notável que a relação entre a duração da execução de cada etapa da coleta e a métrica de consumo total permanece perfeita. Essa consistência é evidenciada de maneira esclarecedora na Figura 18. Apesar das variações inesperadas no consumo total, a correspondência impecável entre o tempo de execução de cada etapa e o consumo total sugere uma interação estável e proporcional entre essas métricas específicas.

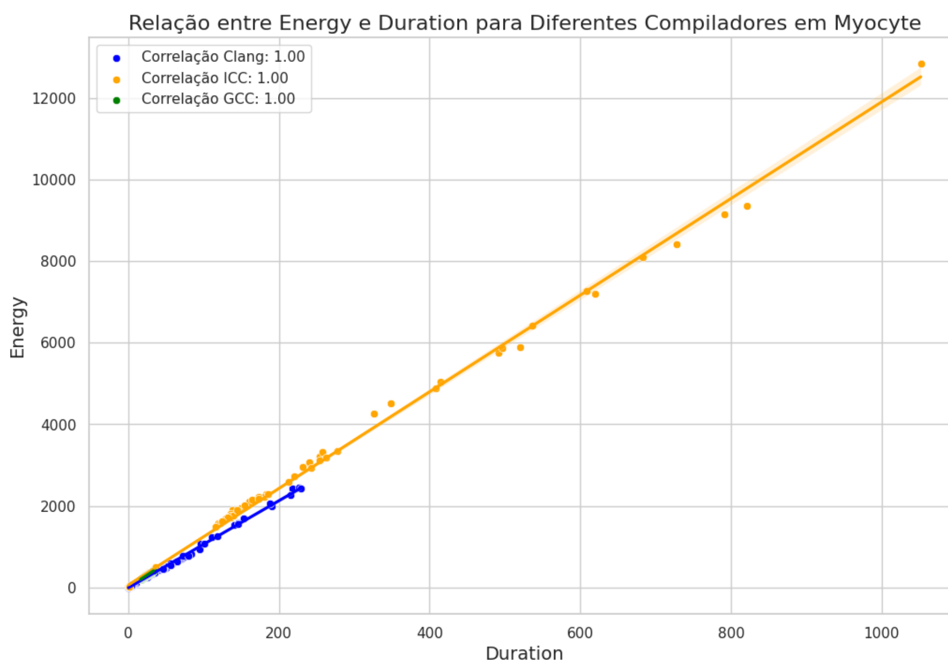

Figura 18 – Correlação entre duração e consumo energético total do Myocyte
Autoria própria.

Assim como os resultados do LavaMD, conforme descrito na subseção 5.1.3, o benchmark Myocyte permaneceu consistentemente acima dos 90% de uso da CPU, como evidenciado na Figura 19.

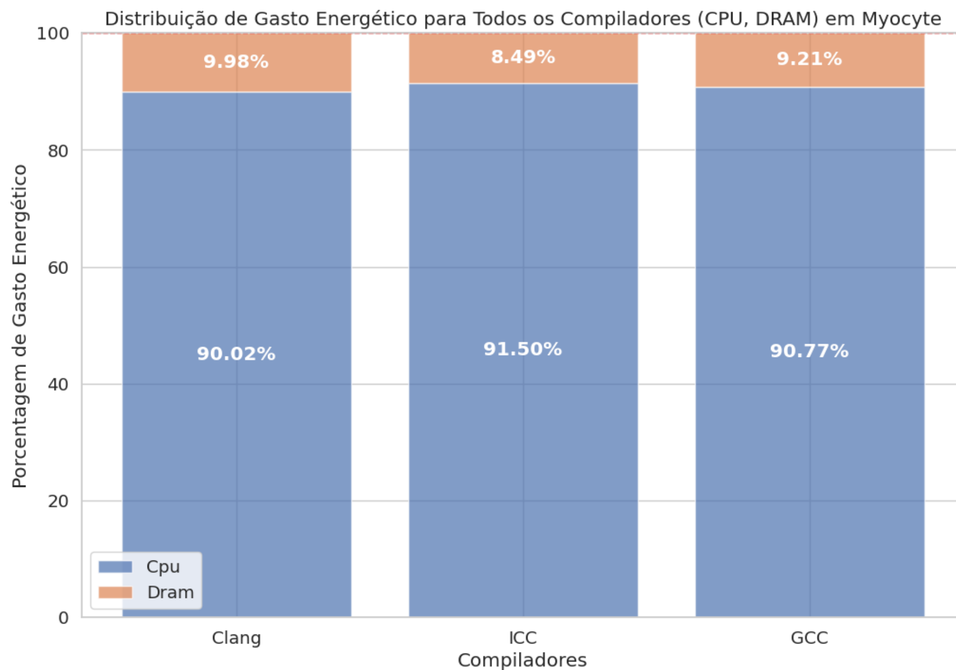

Figura 19 – Distribuição Média Percentual entre CPU e DRAM em relaçao ao consumo energético total do Myocyte

Autoria própria.

Essa coerência persiste, apesar dos resultados equívocos até então apresentados pelo benchmark. Além disso, é notável que, devido à aberração estatística previamente discutida, houve um impacto direto na média de eficiência em relação à aplicação compilada em GCC. Nesse cenário, o Clang mostrou-se 28.69% menos eficiente. Apesar desse revés, permanece substancialmente mais eficiente do que o ICC, que apresenta uma eficiência 442.95% inferior, conforme demonstrado de maneira imperativa na Figura 20.

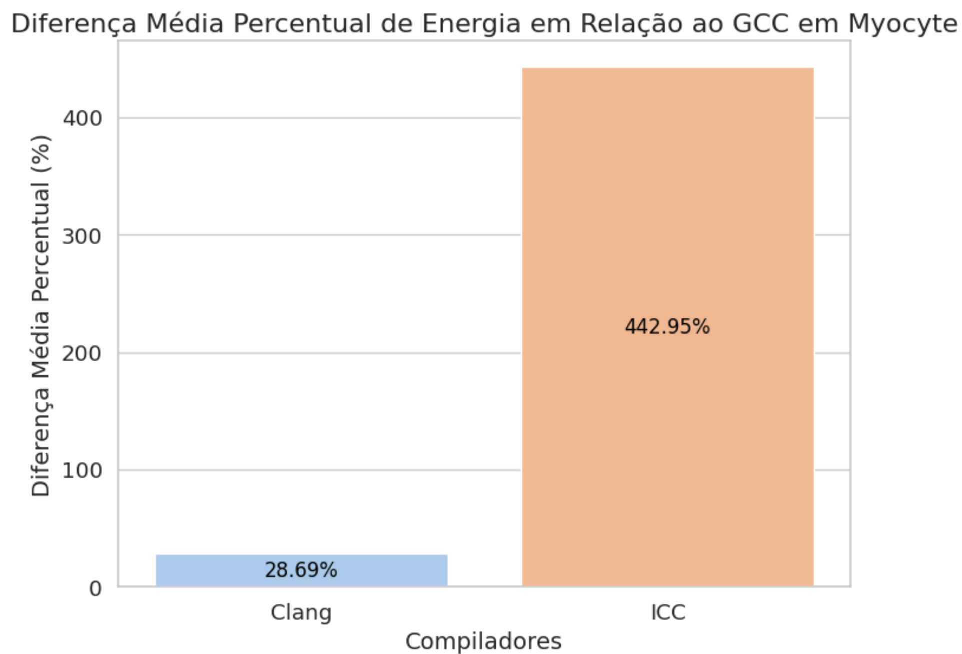

Figura 20 – Média Percentual de ganho de consumo em relação ao compilador GCC do Myocyte
Autoria própria.

5.2 Análise de *Outliers*

Nesta seção, serão examinadas métricas específicas para identificar e investigar possíveis anomalias nos dados coletados. Com técnicas de detecção de *outliers*, nossa análise visa destacar padrões ou discrepâncias significativas que possam influenciar a integridade e interpretação dos resultados.

5.2.1 Pré Visualização com *BoxPlots*

Nesta observação visual, fica evidente que, à exceção do *benchmark Myocyte*, conforme representado na Figura 25, nenhum dos outros *benchmarks*, como ilustrado nas Figuras 21, 22, 23 e 24, apresenta qualquer anomalia aparente em seus dados. Diante desse cenário, optaremos por realizar uma análise mais detalhada exclusivamente nos dados referentes ao *benchmark Myocyte*, onde se destaca a possível presença de *outliers* que requerem uma investigação minuciosa.

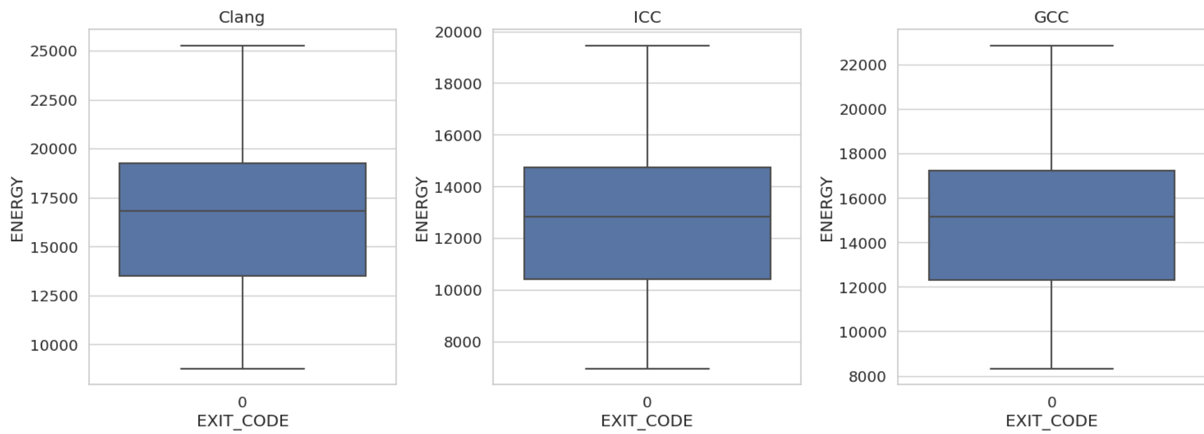

Figura 21 – Verificação e Analise de Outlier do Stream Cluster
Autoria própria.

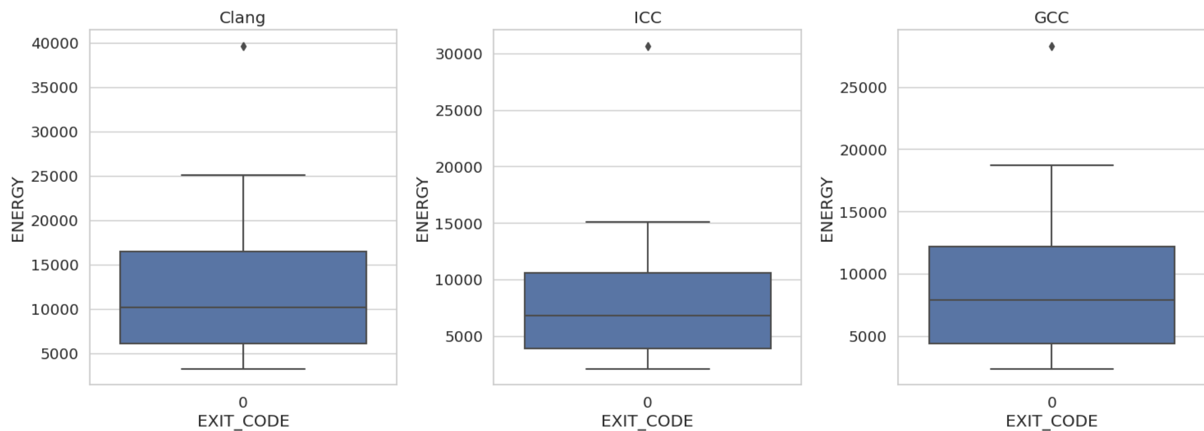

Figura 22 – Verificação e Analise de Outlier do LU Decomposition
Autoria própria.

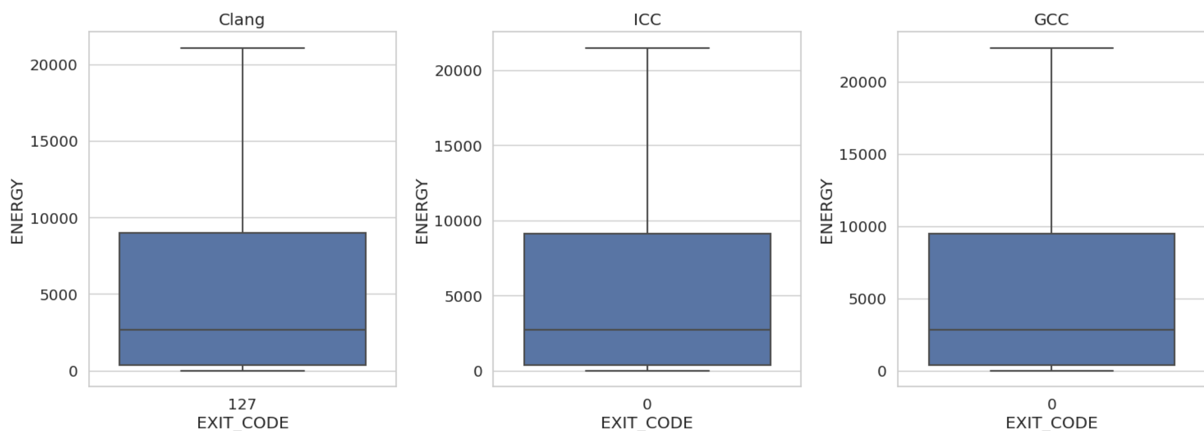

Figura 23 – Verificação e Analise de Outlier do LavaMD
Autoria própria.

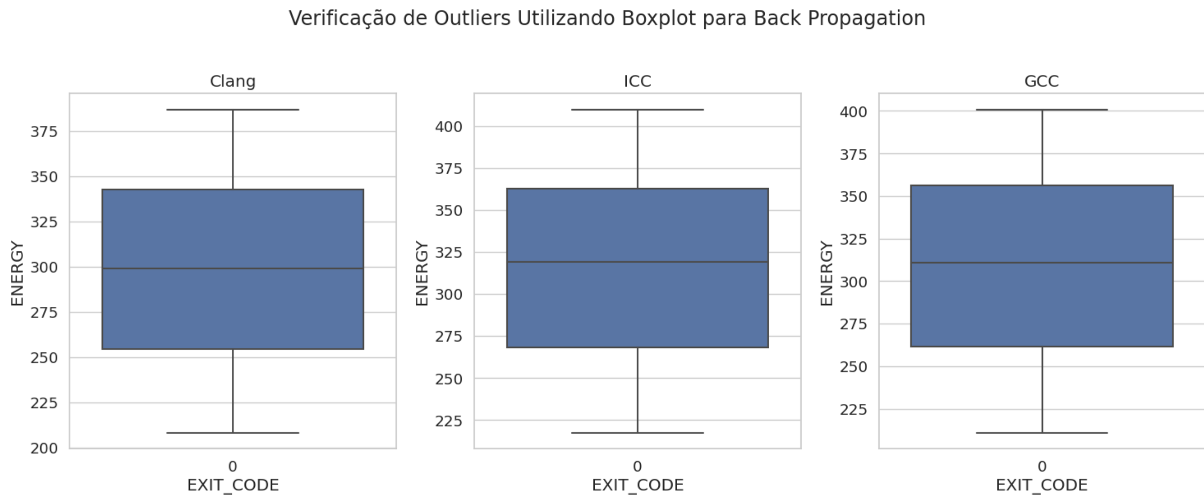

Figura 24 – Verificação e Analise de Outlier do Back Propagation
Autoria própria.

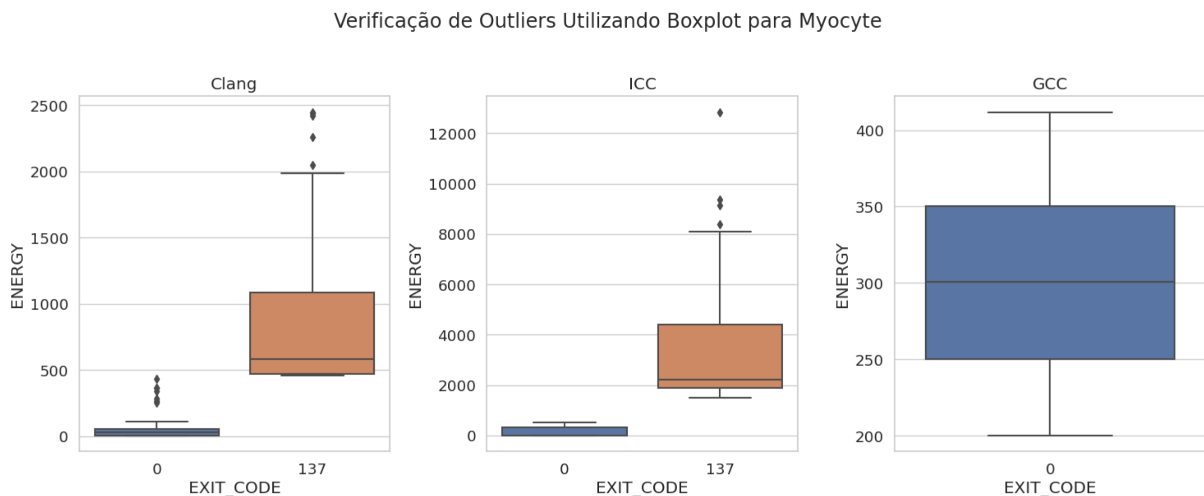

Figura 25 – Verificação e Analise de Outlier do Myocyte
Autoria própria.

É relevante observar que o ajuste foi realizado considerando um *EXIT_CODE* diferente de zero, conforme evidenciado na Figura 25. O *EXIT_CODE* destacado possui o valor *137*, que corresponde a *SIGKILL*, indicando que o sistema interrompeu o processo de maneira inesperada para evitar o travamento do mesmo.

5.2.2 Uso de *One Class SVM* para detectar anomalia em *Myocyte*

Com base no *Boxplot* apresentado na Figura 25, implementou-se uma adaptação do modelo de aprendizado com o GCC como referência, uma vez que este não apresentava quaisquer anomalias, conforme constatado na Figura 26.

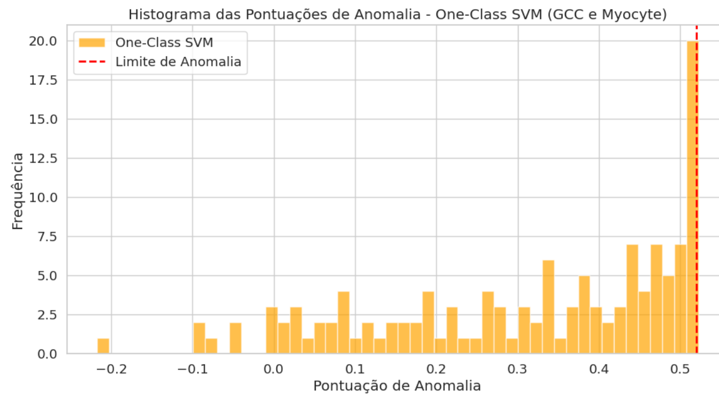

Figura 26 – Detecção de anomalia em Myocyte compilado em GCC utilizando One Class SVM
Autoria própria.

Após essa calibração, prosseguiu-se com a análise dos dados pós-treinamento, evidenciando a frequência de anomalias nos dados das compilações do Clang, conforme ilustrado na Figura 27, e do ICC, como representado na Figura 28. Essa abordagem possibilitou destacar e quantificar a presença de padrões anômalos específicos nas compilações em Clang e ICC em relação ao *benchmark Myocyte*.

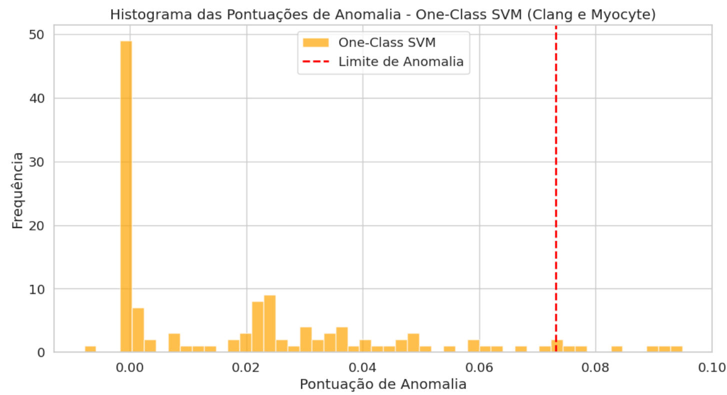

Figura 27 – Detecção de anomalia em Myocyte compilado em Clang utilizando One Class SVM
Autoria própria.

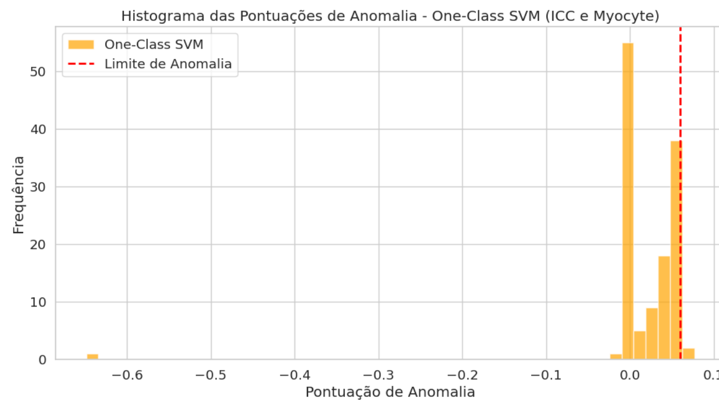

Figura 28 – Detecção de anomalia em Myocyte compilado em ICC utilizando One Class SVM
Autoria própria.

5.3 Comparação dos Compiladores em Relação aos *Benchmarks*

Nesta seção, será conduzida uma análise comparativa abrangente entre os *benchmarks* e seus respectivos compiladores, buscando proporcionar uma visão abrangente dos resultados obtidos nesta pesquisa.

5.3.1 Distribuição do consumo dos Compiladores por *Benchmark*

É evidente uma notória tendência na distribuição percentual apresentada na Figura 29, na qual o ICC lidera em eficiência em quatro dos *benchmarks*, com exceção do *Myocyte*. Essa observação destaca o desempenho proeminente do ICC em relação aos demais compiladores na maioria dos casos analisados.

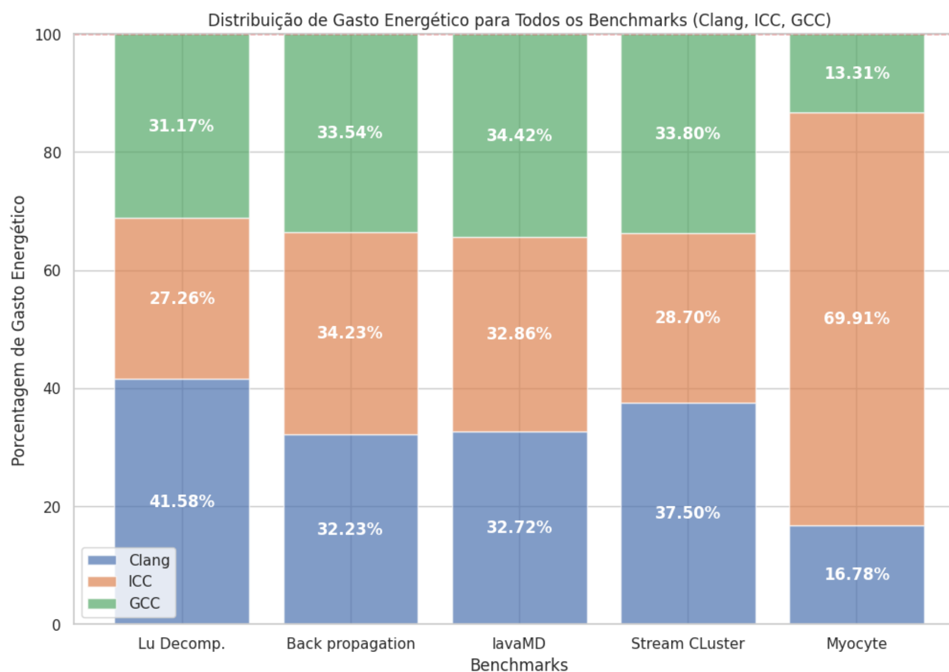

Figura 29 – Distribuição de Consumo dos Compiladores para cada Benchmark
Autoria própria.

5.3.2 Média Percentual do Consumo Total

Ao analisar o consumo geral entre os compiladores, inicialmente observasse, conforme demonstrado no gráfico da Figura 30, que o compilador ICC lidera o ranking como o menos eficiente entre os três. Ele apresenta 38.59% a mais de consumo em comparação com os 32.16% do Clang e os 29.25% do GCC. Entretanto, como destacado na seção 5.2.

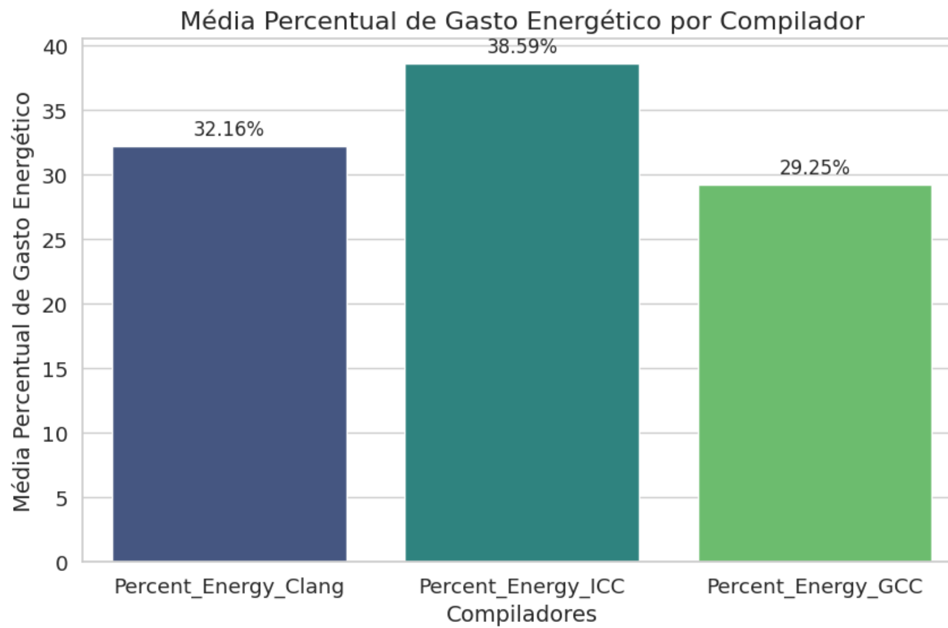

Figura 30 – Média Percentual de consumo total dos compiladores
Autoria própria.

Devido às anomalias geradas durante a coleta de dados relacionados ao *benchmark Myocyte*, ao removê-las da estimativa, percebesse que, na verdade, o ICC é o compilador mais eficiente. Ele se posiciona com 30.76%, em comparação com os 36.01% e 33.23% do Clang e GCC, respectivamente, como ilustrado no gráfico da Figura 31. Essa retificação na análise destaca a importância de considerar e tratar anomalias para uma avaliação precisa do desempenho dos compiladores.

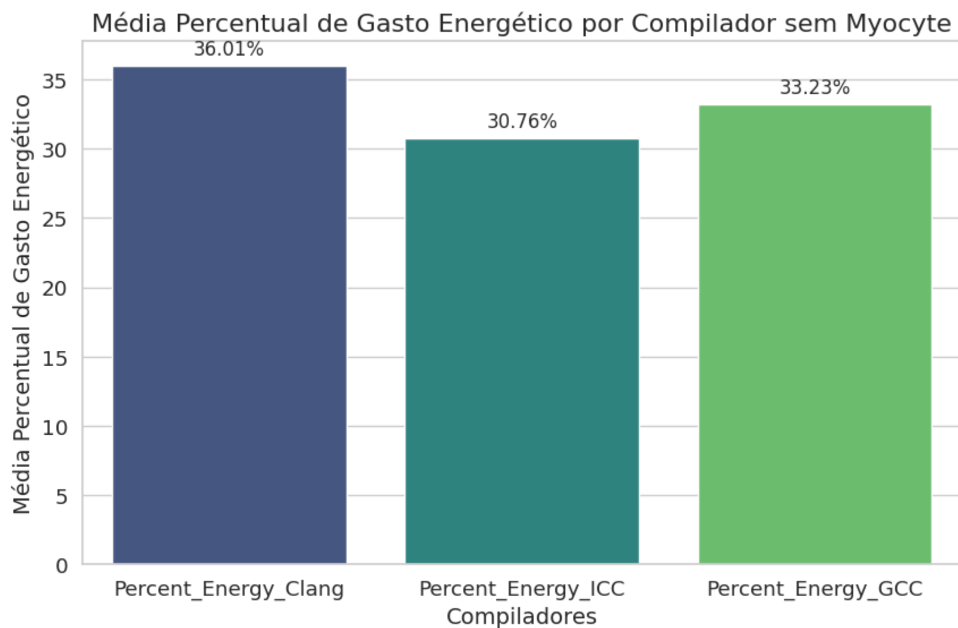

Figura 31 – Média Percentual de consumo total dos compiladores excluindo Myocyte
Autoria própria.

5.4 Comparando os Binários Gerados para cada *Benchmark*

Nesta seção, procederemos com uma comparação do tamanho dos arquivos gerados pelos *benchmarks* para cada compilador. Essa análise visa oferecer informações sobre as características de saída dos compiladores em diferentes contextos de *benchmarking*.

5.4.1 Tamanho dos arquivos gerados

A Figura 32 destaca as disparidades no tamanho dos arquivos gerados por cada compilador. Essa visualização proporciona informações valiosas sobre a eficiência de cada compilador em termos de otimização de código e utilização de recursos, uma vez que o tamanho do arquivo pode ser indicativo do desempenho e da eficácia das otimizações realizadas durante o processo de compilação.

Ao observar essas diferenças, é possível realizar uma análise mais aprofundada para compreender como cada compilador aborda a geração de código e como isso impacta o resultado final em termos de consumo de recursos e desempenho da aplicação. Essa comparação do tamanho dos arquivos é uma métrica significativa na avaliação da eficiência dos compiladores em cenários diversos e fornece uma visão abrangente de seu desempenho relativo.

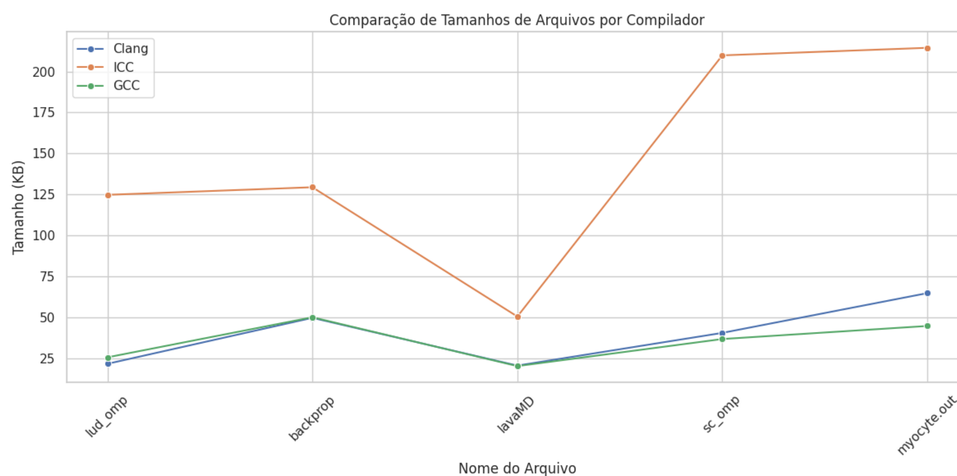

Figura 32 – Comparação dos tamanhos dos binários gerados
Autoria própria.

5.4.2 Aumento Percentual

Na Figura 33, evidenciamos a diferença percentual no tamanho dos arquivos gerados em relação aos arquivos compilados em GCC, que foi utilizado como referência para esta comparação. Esta análise proporciona uma perspectiva clara das variações proporcionadas por diferentes compiladores em termos de tamanho de arquivo, revelando dados cruciais sobre suas estratégias de otimização.

Ao examinar a LU Decomposition, percebemos que o Clang gerou um arquivo aproximadamente 15% menor em comparação ao GCC, enquanto o ICC produziu um arquivo significativamente maior, cerca de 384% maior. No caso do Backprop, o Clang gerou um arquivo ligeiramente menor, aproximadamente 0.73%, enquanto o ICC resultou em um arquivo 157% maior. Essa tendência persiste, destacando-se que o ICC pode gerar arquivos até 468% maiores em comparação aos arquivos compilados com o GCC.

Essa análise percentual do aumento no tamanho dos arquivos oferece uma visão mais precisa das discrepâncias entre os compiladores, permitindo uma compreensão mais profunda de como suas abordagens individuais impactam o consumo de espaço e recursos durante a compilação.

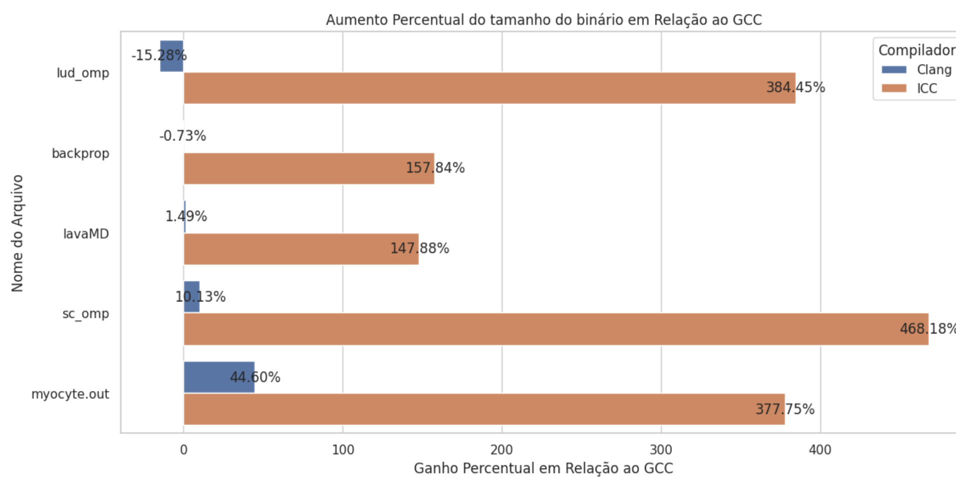

Figura 33 – Comparação do aumento Percentual do tamanho dos arquivos em relação ao GCC
Autoria própria.

6 CONCLUSÕES E TRABALHOS FUTUROS

Neste estudo, realizasse uma análise abrangente do consumo energético de *benchmarks* utilizando diferentes compiladores, todos integrados ao conjunto Rodinia. A variedade de simulações incorporadas permitiu uma avaliação detalhada do desempenho de cada compilador em face dessas simulações específicas.

Ao explorar a ampla gama de simulações presentes no suíte Rodinia, obtivemos informações cruciais sobre como cada compilador enfrenta diversas cargas de trabalho e cenários de aplicação. A diversidade nas simulações ofereceu uma compreensão mais profunda do comportamento de cada compilador em termos de eficiência energética.

Durante a análise comparativa, levamos em consideração diversos fatores, como a otimização do código gerado pelos compiladores e a eficácia na gestão de recursos. O objetivo não era apenas identificar diferenças no consumo energético, mas também compreender as razões subjacentes a essas disparidades, proporcionando, assim, uma visão mais completa do desempenho relativo de cada compilador em contextos específicos.

Ao examinarmos o ganho percentual de cada compilador em vários cenários e aplicarmos uma análise aprofundada com a identificação de outliers, torna-se evidente que, embora compiladores como ICC e Clang sejam alternativas viáveis para determinados tipos de problemas, ainda apresentam irregularidades devido a anomalias identificadas na observação dos dados obtidos.

É relevante destacar um aspecto frequentemente negligenciado na atualidade, dada a abundância de memória disponível para armazenamento nos dias de hoje. A nossa análise revela uma tendência positiva quando os arquivos executáveis possuem tamanhos maiores. Isso sugere que os compiladores capazes de obter melhor desempenho são provavelmente aqueles que geram binários maiores, indicando possivelmente um uso mais cauteloso de abreviações em sua escrita e proporcionando um arquivo mais detalhado para interpretação pelo processador.

Esta observação destaca a importância de considerar não apenas métricas tradicionais de desempenho, mas também características relacionadas ao tamanho do binário gerado pelos compiladores. Compreender a relação entre o tamanho do executável e o desempenho pode guiar escolhas mais informadas na seleção de compiladores, especialmente em cenários onde a eficiência de armazenamento e interpretação do código pelo processador desempenham um papel crucial.

Dessa forma, essa pesquisa não apenas amplia o entendimento do impacto energético de diferentes compiladores, mas também fornece informações valiosas para otimizar a escolha do compilador em cenários específicos. Isso contribui para promover a eficiência energética e aprimorar o desenvolvimento de software em ambientes que requerem especial atenção à gestão de recursos.

6.1 Trabalhos Futuros

Em estudos futuros, para a continuação da pesquisa realizada, seria de extrema importância empregar máquinas mais modernas. Além disso, a exploração de diferentes configurações de hardware torna-se crucial, uma vez que, da mesma forma que os benchmarks escolhidos abrangem diversas áreas, diferentes configurações podem gerar dados relevantes para a avaliação conduzida neste trabalho. Adicionalmente, para uma análise mais aprofundada dos binários gerados pelos compiladores, a aplicação de engenharia reversa nos binários para a linguagem assembly poderia revelar informações adicionais sobre como o tamanho gerado influencia no menor consumo energético dessas aplicações.

REFERÊNCIAS

ALMOMANY, A.; ALQURAAN, A.; BALACHANDRAN, L. Gcc vs. icc comparison using parsec benchmarks. **IJITEE**, Citeseer, v. 4, n. 7, 2014.

AMARIS, M. et al. Evaluating execution time predictions on gpu kernels using an analytical model and machine learning techniques. **Journal of Parallel and Distributed Computing**, Elsevier, v. 171, p. 66–78, 2023.

AMIRI, H.; SHAHBAHRAMI, A. Simd programming using intel vector extensions. **Journal of Parallel and Distributed Computing**, Elsevier, v. 135, p. 83–100, 2020.

ANTAO, S. F. et al. Offloading support for openmp in clang and llvm. In: IEEE. **2016 Third Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)**. [S.l.], 2016. p. 1–11.

ASANOVIC, K. et al. The landscape of parallel computing research: A view from berkeley. eScholarship, University of California, 2006.

BACKUS, J. W.; HEISING, W. P. Fortran. **IEEE Transactions on Electronic Computers**, EC-13, n. 4, p. 382–385, 1964.

BISHOP, C. M.; NASRABADI, N. M. **Pattern recognition and machine learning**. [S.l.]: Springer, 2006. v. 4.

CHAVAN, A. S. et al. Experimental analysis of dedicated gpu in virtual framework using vgpu. In: **2021 International Conference on Emerging Smart Computing and Informatics (ESCI)**. [S.l.: s.n.], 2021. p. 483–489.

CHE, S. et al. A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads. In: **IEEE International Symposium on Workload Characterization (IISWC'10)**. [S.l.: s.n.], 2010. p. 1–11.

COURTY, B. Unable to read intel rapl files for cpu power : Permission denied. 2021. Acesso em: 06/05/2023. Disponível em: <https://github.com/mlco2/codecarbon/issues/244>.

DANIEL. Compilers. **Practical Introduction to Computer Architecture**, Springer, p. 451–493, 2009.

DASH, M. B.; SAHU, A. Characterization and implications for architecture of the parsec benchmark suite.

DE, R. Learning representations by back-propagation errors. **nature**, v. 323, p. 533–536, 1986.

DICKSON, V. L.; SEBOK, P. T. Quantifying the power consumption of processes on linux using intel rapl. 2023.

DONGARRA, J. Trends in high performance computing: a historical overview and examination of future developments. **IEEE Circuits and Devices Magazine**, v. 22, n. 1, p. 22–27, 2006.

DONGARRA, J.; LUSZCZEK, P. Hpc challenge: design, history, and implementation highlights. In: **Contemporary High Performance Computing**. [S.l.]: Chapman and Hall/CRC, 2017. p. 13–30.

EMBREE, P. M.; KIMBLE, B.; BARTRAM, J. F. **C language algorithms for digital signal processing**. [S.l.]: Acoustical Society of America, 1991.

FILHO, J. E. M. **Descobrindo o Linux-3ª Edição: Entenda o sistema operacional GNU/Linux**. [S.l.]: Novatec Editora, 2012.

GAWRYCH, B.; CZARNUL, P. Performance assessment of openmp constructs and benchmarks using modern compilers and multi-core cpus.

GOLUB, G. H.; LOAN, C. F. V. Matrix computations, 4th. **Johns Hopkins**, 2013.

GONG, Z. et al. An empirical study of the effect of source-level loop transformations on compiler stability. **Proceedings of the ACM on Programming Languages**, ACM New York, NY, USA, v. 2, n. OOPSLA, p. 1–29, 2018.

GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. **Deep learning**. [S.l.]: MIT press, 2016.

GOUGH, B. J.; STALLMAN, R. **An Introduction to GCC.** [S.l.]: Network Theory Limited, 2004.

HATCHER, P. J. et al. Dataparallel c: A simd programming language for multicomputers. In: IEEE COMPUTER SOCIETY. **The Sixth Distributed Memory Computing Conference, 1991. Proceedings**. [S.l.], 1991. p. 91–92.

HOLLMAN, D. S. et al. mdspan in c++: A case study in the integration of performance portable features into international language standards. In: IEEE. **2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)**. [S.l.], 2019. p. 60–70.

HUSSAIN, S. M. et al. Seven pillars to achieve energy efficiency in high-performance computing data centers. **Recent Trends and Advances in Wireless and IoT-enabled Networks**, Springer, p. 93–105, 2019.

HUYNH, A. et al. Tp-parsec: A task parallel parsec benchmark suite. **Journal of Information Processing**, Information Processing Society of Japan, v. 27, p. 211–220, 2019.

JR, G. L. S. An overview of common lisp. In: **Proceedings of the 1982 ACM Symposium on LISP and Functional Programming**. [S.l.: s.n.], 1982. p. 98–107.

KAYACIK, H. G.; ZINCIR-HEYWOOD, N. Analysis of three intrusion detection system benchmark datasets using machine learning algorithms. In: SPRINGER. **International Conference on Intelligence and Security Informatics**. [S.l.], 2005. p. 362–367.

KHAN, K. N. et al. Rapl in action: Experiences in using rapl for power measurements. **ACM Trans. Model. Perform. Eval. Comput. Syst.**, Association for Computing Machinery, New York, NY, USA, v. 3, n. 2, mar 2018. ISSN 2376-3639. Disponível em: <https://doi.org/10.1145/3177754>.

KIESSLING, A. An introduction to parallel programming with openmp. In: **The University of Edinburgh, A Pedagogical Seminar (accessed 24 September 2020), URL: https://www. roe. ac. uk/ifa/postgrad/pedagogy/2009_kiessling. pdf**. [S.l.: s.n.], 2009. v. 76.

LATTNER, C.; ADVE, V. Llvm: a compilation framework for lifelong program analysis & transformation. In: **International Symposium on Code Generation and Optimization, 2004. CGO 2004.** [S.l.: s.n.], 2004. p. 75–86.

LI, K.-L. et al. Improving one-class svm for anomaly detection. In: **Proceedings of the 2003 International Conference on Machine Learning and Cybernetics (IEEE Cat. No.03EX693)**. [S.l.: s.n.], 2003. v. 5, p. 3077–3081 Vol.5.

MACHADO, R. S. et al. Comparing performance of c compilers optimizations on different multicore architectures. In: **2017 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)**. [S.l.: s.n.], 2017. p. 25–30.

MEMETI, S. et al. Benchmarking opencl, openacc, openmp, and cuda: programming productivity, performance, and energy consumption. In: **Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing**. [S.l.: s.n.], 2017. p. 1–6.

MISHRA, A.; MALIK, A. M.; CHAPMAN, B. Extending the llvm/clang framework for openmp metadirective support. In: **2020 IEEE/ACM 6th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC) and Workshop on Hierarchical Parallelism for Exascale Computing (HiPar)**. [S.l.: s.n.], 2020. p. 33–44.

MISHRA, A.; MALIK, A. M.; CHAPMAN, B. Extending the llvm/clang framework for openmp metadirective support. In: IEEE. **2020 IEEE/ACM 6th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC) and Workshop on Hierarchical Parallelism for Exascale Computing (HiPar)**. [S.l.], 2020. p. 33–44.

NOVILLO, D. Gcc an architectural overview, current status, and future directions. In: **Proceedings of the linux symposium**. [S.l.: s.n.], 2006. v. 2, p. 185.

REN, X. G. Optimize openfoam from the compiler perspective. **Applied Mechanics and Materials**, Trans Tech Publ, v. 687, p. 3183–3186, 2014.

RITCHIE, D. M. The development of the c language. **ACM Sigplan Notices**, v. 28, n. 3, p. 201–208, 1993.

SAMMET, J. E. The early history of cobol. In: **History of Programming Languages**. [S.l.: s.n.], 1978. p. 199–243.

STREITZ, F. H. et al. 100+ tflop solidification simulations on bluegene/l. In: **Proceedings of IEEE/ACM Supercomputing**. [S.l.: s.n.], 2005. v. 5.

SZAFARYN, L. G. et al. Experiences with achieving portability across heterogeneous architectures. **Proceedings of WOLFHPC, in Conjunction with ICS, Tucson**, Citeseer, 2011.

TABASSUM, M.; MATHEW, K. Software evolution analysis of linux (ubuntu) os. In: **2014 International Conference on Computational Science and Technology (ICCST)**. [S.l.: s.n.], 2014. p. 1–7.

THORNTON, J. E. The cdc 6600 project. **Annals of the History of Computing**, IEEE, v. 2, n. 4, p. 338–348, 1980.

TIWARI, A. et al. Modeling power and energy usage of hpc kernels. In: **2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum**. [S.l.: s.n.], 2012. p. 990–998.

WILLIAMS, J.; CURTIS, L. Green: The new computing coat of arms? **IT Professional Magazine**, IEEE Computer Society, v. 10, n. 1, p. 12, 2008.

WONG, P.; WIJNGAART, R. D. Nas parallel benchmarks i/o version 2.4. **NASA Ames Research Center, Moffet Field, CA, Tech. Rep. NAS-03-002**, p. 91, 2003.

XIE, G.; XIAO, Y.-H. How to benchmark supercomputers. In: **2015 14th International Symposium on Distributed Computing and Applications for Business Engineering and Science (DCABES)**. [S.l.: s.n.], 2015. p. 364–367.

ZWAKENBERG, R. G. Cdc 6600/7600 optimization. **ACM SIGPLAN Notices**, ACM New York, NY, USA, v. 5, n. 7, p. 130, 1970.