Análise de Desempenho e Consumo Energético de Aplicações Recursivas em Ambientes OpenMP, CUDA e CUDA DP

Angelo Gaspar Diniz Nogueira[1], **Arthur Francisco Lorenzon**[2], **Claudio Schepke**[1], **Diego Kreutz**[1]

[1]Laboratório de Estudos Avançados em Computação (LEA)
Programa de Pós-Graduação em Engenharia de Software (PPGES)
Universidade Federal do Pampa (UNIPAMPA), Alegrete, Brazil

{angelonogueira.aluno,claudioschepke,diegokreutz}@unipampa.edu.br

[2]Grupo de Processamento Paralelo e Distribuído (GPPD)
Universidade Federal do Rio Grande do Sul (UFRGS), Porto Alegre, Brazil

aflorenzon@inf.ufrgs.br

Resumo. *Interfaces de programação paralela como CUDA possibilitam explorar a concorrência em GPUs. Embora o poder de processamento seja significativo neste tipo de arquitetura, a eficiência pode ser limitada em aplicações recursivas, devido à necessidade de comunicação entre GPU e CPU. Uma alternativa é utilizar a extensão Dynamic Parallelism, também conhecida como CUDA DP. Com o objetivo de investigar em maior profundidade esta extensão de paralelismo, neste artigo avaliamos o desempenho e o consumo energético de aplicações recursivas usando OpenMP, CUDA e CUDA DP. Os resultados obtidos indicam que CUDA DP viabiliza uma economia significativa na execução de aplicações com características semelhantes ao Mergesort, chegando a uma redução de até* $23\times$ *no tempo de execução e* $7\times$ *no consumo de energia quando comparado com as variantes CUDA e OpenMP, respectivamente. Na implementação do BFS com CUDA DP, observa-se um ganho aproximado de* $5\times$ *no consumo de energia e no tempo de execução em comparação com o OpenMP. Contudo, em comparação com CUDA, há uma leve perda de* $1,6\%$ *no consumo de energia e de* 5% *no tempo de execução.*

1. Introdução

A demanda por desempenho vem crescendo significativamente em diferentes contextos e domínios, como *big data*, ciência de dados, computação gráfica e inteligência artificial [Navaux et al., 2023, Palencia and Rutten, 2024]. Para atender a esta demanda, empresas frequentemente tendem a atualizar incrementalmente seus recursos computacionais, incluindo diferentes tipos de máquinas e arquiteturas modernas. No entanto, sem investigar o efeito de um novo em relação às aplicações, esta atualização pode impactar negativamente o consumo de energia e a pegada de carbono conforme O'brien et al., 2017 e Nana et al., 2023.

Uma das alternativas para a melhoria do desempenho de aplicações em relação ao consumo de energia de sistemas computacionais é a exploração do paralelismo

em arquiteturas modernas [Jin et al., 2017]. Atualmente existem inúmeras interfaces de programação paralela e linguagens que permitem a exploração do paralelismo. Entre as mais populares estão: (*i*) OpenMP [OpenMP Architecture Review Board, 2022] para arquiteturas homogêneas e heterogêneas, compostas por unidades computacionais similares, e; (*ii*) CUDA [NVIDIA, 2022] para arquiteturas GPUs da NVIDIA. No entanto, a eficiência de CUDA é limitada em aplicações recursivas em comparação a alternativas como OpenMP devido a necessidade de uma comunicação entre GPU e CPU a cada iteração da recursão, na forma de uma chamada de *kernel* (fluxo de execução de uma aplicação na GPU).

Para mitigar o efeito negativo da comunicação entre GPU e CPU em aplicações recursivas que usam a interface CUDA, foi desenvolvida a extensão *Dynamic Parallelism* (DP) [Adinetz, 2014], que permite que a GPU faça chamadas recursivas de *kernel*. Contudo, existem diversos relatos recorrentes relacionados aos custos inerentes ao uso do paralelismo dinâmico, associados as chamadas recursivas de *kernel*, a criação e destruição de *streams* e a divergência de execução entre *threads* [Wang and Yalamanchili, 2014, Plauth et al., 2015, Jarząbek and Czarnul, 2017, Yi et al., 2021, Park et al., 2022] . Entre os desafios frequentemente identificados para obter ganhos significativos em arquiteturas paralelas específicas estão o grande volume de dados e o balanceamento de *workloads*. Portanto, é necessário avaliar o impacto de CUDA DP para cada tipo de aplicação, procurando identificar os benefícios e o custo da extensão para os respectivos algoritmos.

Com o objetivo de contribuir para a análise de algoritmos recursivos específicos no contexto de CUDA DP, como Mergesort, Quicksort, BFS e SSP, nós apresentamos neste trabalho uma análise comparativa utilizando implementações OpenMP, CUDA e CUDA DP. Para a avaliação, nós consideramos as métricas clássicas de consumo de energia e tempo de execução, que são as mais frequentemente utilizadas pelos demais trabalhos da área. A nossa análise procura avaliar os custos e benefícios do paralelismo dinâmico, nos algoritmos recursivos, utilizando diferentes quantidades de dados. Conforme apontado na literatura, o volume de dados é um dos fatores que pode impactar o perfil de desempenho e consumo de energia dos algoritmos. Este fato é exemplificado na implementação CUDA DP do algoritmo *Mergesort* que obteve a melhor performance (23× menos tempo de execução e 7× menos consumo de energia em relação as alternativas), onde o DP é invocado apenas quando há um volume suficientemente grande de dados para serem computados.

O restante do trabalho está estruturado da seguinte forma. Na Seção 2 e na Seção 3 apresentamos os trabalhos relacionados e a metodologia utilizada. Na Seção 5 discutimos as métricas adotadas e os resultados dos experimentos. Por fim, na Seção 6, apontamos as considerações finais e trabalhos futuros.

2. Trabalhos relacionados

Na Tabela 1 apresentamos os principais trabalhos relacionados ao contexto de análise de CUDA e da extensão DP. A tabela mostra as métricas, *benchmarks* e as APIs de paralelismo consideradas nas avaliações. Como podemos observar, todos os trabalhos consideram o tempo de execução como métrica de avaliação, porém, apenas em torno de 50% deles avalia o consumo de energia.

Tabela 1. Trabalhos relacionados

Referência	Métricas	Benchmarks	Comparativos
[Khalilov and Timoveev, 2021]	largura de banda da memória GPU, eficiência de paralelismo e tempo de execução	Matrix multiplication, BabelStream, Cloverleaf e LULESH	CUDA, OpenACC OpenMP
[Memeti et al., 2017]	Consumo de energia, tempo de execução e produtividade de programação	SPEC Accel e Rodinia	OpenCL, OpenACC, OpenMP, e CUDA
[Bozorgmehr et al., 2021]	Erro máximo de convergência e tempo de execuções de iterações	3D red-black successive	CPU, CUDA e CUDA DP
[Araujo et al., 2023]	Tempo de execução, consumo de memória e speedup	NAS Parallel Benchmark	CUDA, OpenACC, OpenMP
[Quezada et al., 2023]	Eficiência energetica e tempo de execução	Mandelbrot set	CUDA e CUDA DP
Este trabalho	Tempo de execução e consumo de energia	BFS, SSP, Quicksort e Mergesort	OpenMP, CUDA, CUDA DP

Como pode ser observado na Tabela 1, há análises que consideram APIs homogêneas como OpenMP e também outras ferramentas para arquiteturas heterogêneas, como OpenACC e OpenCL [Khalilov and Timoveev, 2021, Memeti et al., 2017, Araujo et al., 2023]. Entretanto, nenhum desses trabalhos incluí a extensão DP na avaliação. Além disso, nenhum dos trabalhos relacionados avalia o conjunto específico de algoritmos recursivos Mergesort, Quicksort, BFS e SSP.

Apesar de existirem implementações e análises focadas na extensão DP [Bozorgmehr et al., 2021, Quezada et al., 2023], estas não incluem APIs de arquiteturas homogêneas e abordam apenas um tipo de aplicação recursiva. Diferentemente, o nosso trabalho inclui a API OpenMP e aplicações recursivas distintas, que cobrem ordenação vetorial e busca em grafos, por exemplo. Adicionalmente, Existem autores [Guo et al., 2021, Gupta et al., 2023, El Hajj et al., 2016] que buscam otimizar a performance de algoritmos para CUDA ou CUDA DP. No entanto, estes não consideram ambas, nem outras alternativas para a comparação de performance.

3. Metodologia e Ambiente

Para o desenvolvimento dos experimentos nós consideramos APIs de programação paralela para OpenMP, CUDA e CUDA DP. Nosso objetivo foi criar *benchmarks* de aplicações recursivas distintas, que cobrem:

- Ordenação vetorial: *Mergesort* e *Quicksort* e;
- Busca em grafos: *Bread First Search* e *Single-Source Shortest Path*.

Apesar de existirem implementações especializadas, que levam em conta métodos que auxiliam às implementações DP e CUDA, nós trazemos implementações generalizadas para possibilitar uma comparação justa, cientes das perdas providas por otimizações específicas.

Os experimentos foram realizados em uma máquina heterogênea com arquitetura x86_64 e as seguintes especificações de hardware e software:

- Memória: 48 GB.
- Processador: *AMD Ryzen 9 3900X 12-Core* com 2 *threads* por *core*

- GPU: GeForce GTX 1050 com 869 CUDA *cores*.
- Sistema operacional: Ubuntu 18.04 do Linux

Para a coleta das métricas de consumo de energia e tempo de execução, utilizamos as seguintes ferramentas:

1. `nvdia-smi` para o consumo de energia na GPU, através do monitoramento em intervalos de 1000 ms;
2. `perf` para o consumo de energia na CPU e;
3. a função nativa `gettimeofday()` do Linux para o tempo de execução.

Foram realizadas 10 (dez) iterações de 10.000 (dez mil) execuções de cada *benchmark*[1] e para cada experimento. Para os experimentos variamos o volume de dados. Nos testes para o algoritmo de Quicksort e MergeSort foram utilizados vetores de 100, 10.000 e 100.000 entradas. Os algoritmos *Bread First Search* (BFS) e *Single-Source Shortest Path* (SSSP) consideram o número de nodos e de arestas. Para BFS os valores de Nodos × Arestas foram definidos em 10.000 × 30.000, 100.000 × 300.000 e 500.000 × 1.000.000. Já para SSSP utilizamos os valores de 1.000 × 4.000, 10.000 × 30.000, 100.000 × 300.000 e 200.000 × 400.000.

4. Implementação dos Algoritmos

Esta seção apresenta os detalhes de implementação dos algoritmos Mergesort, Quicksort, BFS e SSSP para as APIs OpenMP, CUDA e CUDA DP. Devido a limitação de espaço, mostramos apenas as versões CUDA DP dos algoritmos. As versões dos algoritmos em OpenMP e CUDA podem ser encontradas em [Nogueira et al., 2024].

4.1. Mergesort

Para a paralelização do algoritmo *Megersort* dividimos o vetor em subconjuntos de posições para serem iterados por *threads*. No entanto, há algumas especificidades para cada uma das implementações: (i) em OpenMP são criadas *tasks* para as chamadas recursivas contendo os subconjuntos, utilizando `selection_sort` como algoritmo para ordenar estes subconjuntos; (ii) em CUDA[1] é aplicada uma abordagem iterativa, com base em um *kernel* composto por um laço de repetição, onde são atribuídos conjuntos de posições com base no identificador das *threads* e para a ordenação é utilizado o `insertion_sort`; (iii) em CUDA DP[2], é usado um laço de repetição que atribui posições às *threads* assim como na implementação CUDA. Porém, nos casos onde há uma expressiva carga de trabalho atribuída a uma *thread*, é realizada um chamada recursiva na GPU para subdividir novamente a carga.

O Algoritmo 1 representa a implementação CUDA DP, onde são utilizados laços de repetições para determinar o número de posições que devem ser processados em uma iteração (linhas 1-3) na CPU. As atribuições às *threads* na GPU, com base no seu identificador, ocorre nas linhas 6 a 10. Nos casos onde o tamanho do subproblema é suficientemente grande, é realizada uma sub-divisão, através da chamada de um novo *kernel*. Nestes casos, a ordenação é realizada via busca binária (linhas 11–13). Caso contrário é utilizado o algoritmo de ordenação `insertion_sort`.

[1]Neste trabalho, um *benchmark* é representado por uma combinação de API e algoritmo. Considerando que temos 4 algoritmos e 3 APIs, temos um total de 12 *benchmarks*.

[1]Baseado no código do repositório `https://github.com/kevin-albert/cuda\-mergesort`

[2]Baseada na implementação do repositório `https://github.com/JoeyOhman/GPUMergeSort`

Algoritmo 1 Mergesort CUDA DP

- `size` (tamanho do vetor)
- `arr` (vetor)
- `aux` (vetor auxiliar)

```
1: Loop na CPU
2: for tamanho_atual = 1 to size step tamanho_atual * 2 do
3:     largura = tamanho_atual * 2
4:     num_sorts = (size + largura - 1) / largura
5:     Kernel na GPU
6:     idx = Id da thread
7:     inicio = largura * idx
8:     meio = inicio + tamanho_atual - 1
9:     fim = min( inicio + largura - 1, size - 1)
10:    num_threads = fim - inicio + 1
11:    if num_threads > 16384 and num_threads < 1024 * 1024 then
12:        numthreadsPerBlock = 1024
13:        Kernel_binary_search()
14:    else
15:        insertion_sort()
16:    end if
17: end for
```

4.2. Quicksort

A implementação do algoritmo *Quicksort* paraleliza as iterações sobre os subconjuntos criados pelo pivô. Todavia, há diferenças entre as implementações: (i) em OpenMP utilizamos uma abordagem *top-down*, onde são utilizadas *tasks* para iterar sobre chamadas recursivas com os subconjuntos; (ii) em CUDA[3] adotamos uma solução iterativa, utilizando *threads* para percorrer as posições de cada subconjunto dos pivôs, com base no seu ID; (iii) em CUDA DP[4] usamos uma abordagem *top-down* recursiva. No entanto, devido à limitação de profundidade máxima (24), realizamos uma ordenação iterativa com o algoritmo `selection_sort`, nos casos onde esse limite seria ultrapassado na próxima iteração.

O Algoritmo 2 representa a implementação CUDA DP, onde utilizamos uma abordagem *top-down*, com o uso de chamadas recursivas (linhas 5-11). Como pode ser observado, é necessária a inclusão de um algoritmo de ordenação *selection sort* iterativo (linhas 1-3), para casos onde a profundidade ultrapassar o limite máximo da extensão DP. Adicionalmente, a nossa implementação difere da original através do uso de memória unificada e uso de múltiplas *streams* para as chamadas de *kernel*.

4.3. BFS

A implementação do algoritmo BFS[5] utiliza dois laços de repetições aninhados para atribuir vértices adjacentes a cada *thread*, de forma a possibilitar à busca simultânea por múltiplos nodos. Ademais, as diferenças entre as implementações são: (i) OpenMP faz uso da cláusula `pragma omp parallel for` no laço de repetição interno, compartilhando informações sobres os nodos e a variável de parada do laço, para evitar conflitos; (ii) na implementação CUDA o laço de repetição externo na CPU é responsável pela chamada de um *kernel* iterativo, que realiza a atribuição de posições com base no id da *thread*. Contudo, é necessário o uso de uma fila para

[3]Baseado no repositório `https://github.com/gongminaaa/GPU_Parallel_Computing`

[4]Baseada na implementação do repositório `https://github.com/NVIDIA/cuda-samples/tree/master/Samples/3_CUDA_Features/cdpSimpleQuicksort`

Algoritmo 2 Quicksort CUDA DP

- `inicio_local` (posição vetorial inicial desta iteração);
- `fim_local` (posição final desta iteração);
- `data` (vetor);
- `Profundidade` (Profundidade atual da recursão)

```
1: if Profundidade ≥ 23 ou inicio_local - fim_local ≤ 32 then
2:     selection_sort(data, left, right)
3:     return
4: else
5:     Processo de ordenação
6:     if inicio_local < rptr - data then
7:         quicksort(data, inicio_local, nptr - data, profundidade + 1)
8:     end if
9:     if lptr - data < fim_local then
10:        quicksort(data, lptr - data, fim_local, profundidade + 1)
11:    end if
12: end if
```

delimitar os elementos que cada *thread* deve processar a cada iteração, de forma a evitar conflitos; (iii) similarmente a CUDA, CUDA DP[6] atribui posições utilizando *kernel* iterativo chamado por um laço externo na CPU. Porém, nos casos em que ocorre uma mudança na estimativa da distância entre nodos durante a execução do *kernel*, é invocada um *kernel* secundário para processar seus nodos adjacentes e então adicionar as adjacências destes a fila.

O Algoritmo 3 representa a implementação CUDA DP, que faz o uso de um laço de repetição na CPU (linhas 1-3) para realizar a chamada do *kernel* de solução. Este *kernel* é responsável pela atribuição de posições com base no identificador das *threads* (linha 5). Com bases nas posições atribuídas, a *thread* realiza as estimativas de distância das suas adjacências e as compara à estimativa atual, que é inicialmente infinita (linhas 6-13). Ademais, quando ocorre uma alteração em uma destas estimativas, é realizada a chamada de um *kernel* secundário através da *thread* 0 (linha 14 – 16). Este *kernel* secundário é responsável pela iteração das adjacências do nodo cuja estimativa foi modificada, dessa forma processando dois níveis da árvore.

4.4. SSSP

A implementação do algoritmo SSSP utiliza diversos conjuntos de vetores para representar a estrutura entre nodos. As diferenças na exploração do paralelismo em cada implementação ocorrem através da maneira de atribuir as posições destes conjuntos para as *threads*: (i) OpenMP utiliza um laço de repetição para atribuir as posições. Neste é invocada a cláusula `pragma omp parallel`. Com base nos identificadores de cada *thread* é atribuído um conjunto de posições; (ii) CUDA utiliza um laço de repetição na CPU, o qual é responsável pela chama de um *kernel* iterativo, que atribui conjuntos de posições com base no identificador da *thread*; (iii) CUDA DP utiliza um laço de repetição na CPU, responsável por chamar um *kernel* principal para o processamento dos nodos. Este invoca um *kernel* secundário, que executa uma nova iteração sobre as adjacências dos nós processados na iteração do *kernel* principal, permitindo que várias iterações sejam realizadas simultaneamente.

[5]Baseada na implementação do repositório `https://github.com/rafalk342/bfs-cuda`

[6]Baseada na implementação do repositório `https://github.com/NPSLab/show-me-dynamic-parallelism/tree/master/apps/bfs-rec`

Algoritmo 3 BFS CUDA DP

- `distancia` (vetor utilizado para armazenar a distância)
- `nivel` (altura atual da árvore)
- `lista_adj` (lista de adjacências)
- `edgesOffset` (vetor que guarda a posição inicial das adjacências dos nós no vetor `lista_adj`)
- `edgesSize` (vetor utilizado para armazenar o número de adjacências de cada nó)

```
1: Loop na CPU
2: while *changed do
3:     *changed = 0
4:     Kernel da GPU
5:     thid = Id da thread
6:     valueChange = 0
7:     if thid < N and d_distance[thid] <= nivel then
8:         int count = 0
9:         for i  = posição inicial das adjacências de thid no vetor de adjacências; i < posição final das adjacências
   de thid no vetor de adjacências; i++ do
10:            v = lista_adj[i]
11:            if d_distance[thid] + 1 < d_distance[v] then
12:                d_distance[v] = d_distance[thid] + 1
13:                d_parent[v] = thid
14:                if  0 <  Número de adjacências de v then
15:                    kernel2«<....»>(....)
16:                end if
17:                valueChange = 1
18:            end if
19:        end for
20:    end if
21:    if valueChange then
22:        *changed = valueChange
23:    end if
24: end while
```

O Algoritmo 4 ilustra a implementação do algoritmo de Programação Dinâmica (DP) utilizando CUDA. Nesta implementação, a CPU gerencia um laço de repetição, que invoca o *kernel* de solução. Esse *kernel* atribui posições com base nos IDs das *threads* (linhas 1 e 3). A partir das posições designadas, cada *thread* estima as distâncias para suas adjacências e as compara com a estimativa atual, inicialmente configurada como infinita (linhas 5-12). Além disso, sempre que há uma alteração em alguma dessas estimativas, a *thread* 0 invoca um *kernel* secundário (linhas 16-17). Esse *kernel* secundário é responsável por iterar sobre as adjacências do nó, cuja estimativa foi modificada, porém, não realiza chamadas recursivas.

5. Resultados

Nesta seção apresentamos os resultados dos 4 algoritmos considerados. A Figura 1 apresenta os resultados da soma da métrica de consumo de energia em joules da CPU e GPU, por algoritmo e valores de entradas utilizado, isto é, tamanho do vetor ou número de nós e arrestas. Um maior detalhamento desses dados é apresentado na Tabela 2.

Como pode ser observado, as implementações CUDA possuem um maior consumo de energia na CPU em comparação as implementações DP. Porém, o contrário ocorre na GPU, onde as implementações CUDA DP possuem um maior consumo. A explicação para esse comportamento está relacionada a maior necessidade de sincronização entre GPU e CPU, o que leva a um maior consumo de energia na CPU na implementação CUDA. No caso da implementação CUDA DP, o maior consumo de energia na GPU é devido aos *kernels* recursivos. A única exceção deste compor-

Algoritmo 4 SSSP CUDA DP

- `numEdges` (número de arestas)
- `dist` (vetor que contém a distância mínima dos nós)
- `finished` (variável utilizada para sinalizar o fim do *loop* de chamadas de *kernel* na CPU)
- `preNode` (vetor utilizado para determinar o predecessor dos nós)

1: `ThreadId` = Id da *thread* atual
2: `StartId` = `ThreadId` * `numEdges`
3: `EndId` = (`ThreadId` + 1) * `numEdges`
4: Verificação para ver se os valores estão fora das dimensões do vetor
5: **for** `i` = `StartId`; `i` < `EndId`; `i` = `i` + 1 **do**
6: `Origem` = Origem da aresta[`i`]
7: `Destino` = Destino da aresta[`i`]
8: `Custo` = Custo da aresta[`i`]
9: **if** `dist[Origem] + Custo < dist[Destino]` **then**
10: `dist[Destino]` = `dist[Origem] + Custo`
11: `preNode[Destino]` = `Origem`
12: `finished` = `false`
13: **end if**
14: **end for**
15: Sincronização entre *threads*
16: **if** `finished == false` **and** `ThreadId == 0` **then**
17: Realizar a chamada do kernel secundário que possui a mesma função, mas não é recursivo
18: **end if**

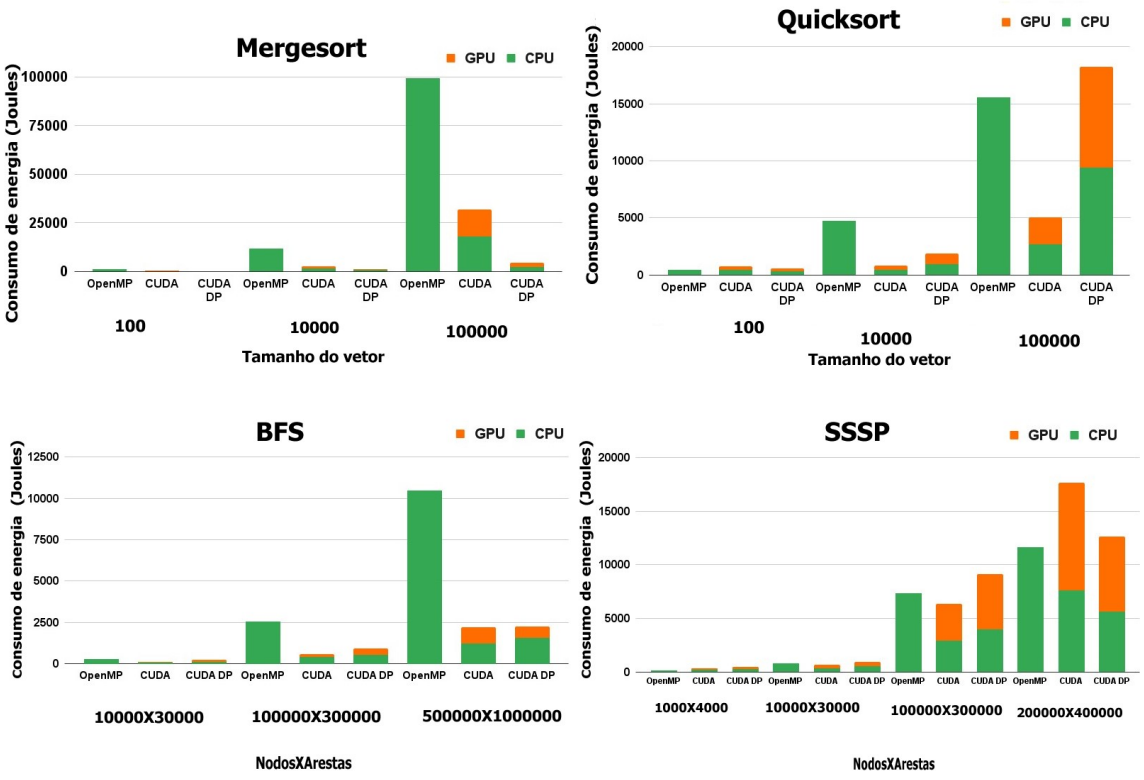

Figura 1. Consumo de energia

Tabela 2. Consumo de energia dos algoritmos

Algoritmo	Tamanho do vetor	OpenMP CPU	CUDA		CUDA DP	
			CPU	GPU	CPU	GPU
Mergesort	**100**	286,0	459,3	339,2	311,9	260,4
	10000	4781,3	475,3	361,1	981,7	909,8
	100000	15557,4	2691,6	2398,5	9409,0	8834,2
Quicksort	**100**	1156,5	143,9	157,0	94,3	76,5
	10000	11709,0	1388,1	1050,0	717,0	520,7
	100000	99459,5	17833,0	13887,5	2327,3	2132,0
BFS	**10000×30000**	286,01	50,72	49,62	127,58	102,57
	100000×300000	2551,94	393,17	202,27	554,30	386,57
	500000×1000000	10494,54	1232,53	970,26	1554,63	684,03
SSSP	**1000×4000**	114,42	190,46	145,97	260,76	202,17
	10000×30000	791,22	354,56	293,61	510,83	428,26
	100000×300000	7323,13	2903,54	3425,65	3996,69	5109,38
	200000×400000	11644,71	7633,95	10014,49	5661,36	6997,52

tamento é a implementação *Quicksort* no tamanho máximo, pois o tamanho maior leva a múltiplas chamadas do *kernel* iterativo, que consequentemente significa mais comunicações com a CPU.

A Figura 2 apresenta o tempo de execução em segundos dos experimentos realizados, separados por algoritmo e tamanho das entradas utilizadas. Com bases nessa figura é possível observar uma tendência entre o tempo de execução e consumo de energia. Quanto maior o tempo de execução maior o consumo. Podemos tirar pelo menos quatro *insights* principais com base nos dados apresentados nos gráficos:

- A implementação CUDA DP do *Quicksort* apresenta a pior performance. Há uma limitação de profundidade de recursão. Devido à maior necessidade de utilizar a função iterativa (`selection_sort`) esse limite é frequentemente atingido em tamanhos maiores. Especificamente há um aumento de $4,8\%$ e $3,89\times$ no tempo de execução e $15,41\%$ e $3,58\times$ no consumo de energia em comparação as alternativas OpenMP e CUDA. Isso sugere que uma abordagem puramente recursiva para problemas grandes, como a utilizada na implementação OpenMP, não é eficiente para CUDA DP.
- Embora a implementação CUDA DP apresente uma performance inicial inferior no algoritmo BFS, a diferença em relação à implementação CUDA tende a diminuir conforme o tamanho do problema aumenta. No maior tamanho testado a diferença foi de apenas 1,60% no consumo de energia e 5% no tempo de execução. A nossa hipótese para este comportamento está relacionada ao aumento de carga computacional e a consequente melhor eficiência na utilização de recursos por parte da implementação CUDA DP. Além disso, a implementação CUDA oferece uma aceleração de 5% em relação à alternativa CUDA DP e 5× em relação ao OpenMP, com um consumo de energia 1.6% menor em relação ao CUDA DP e aproximadamente 5× vezes menor em relação ao OpenMP.
- O uso do segundo *kernel* na implementação SSSP CUDA DP aparenta ter degradado a performance em 23.92% e 26.89% no tempo de execução e 8.01%

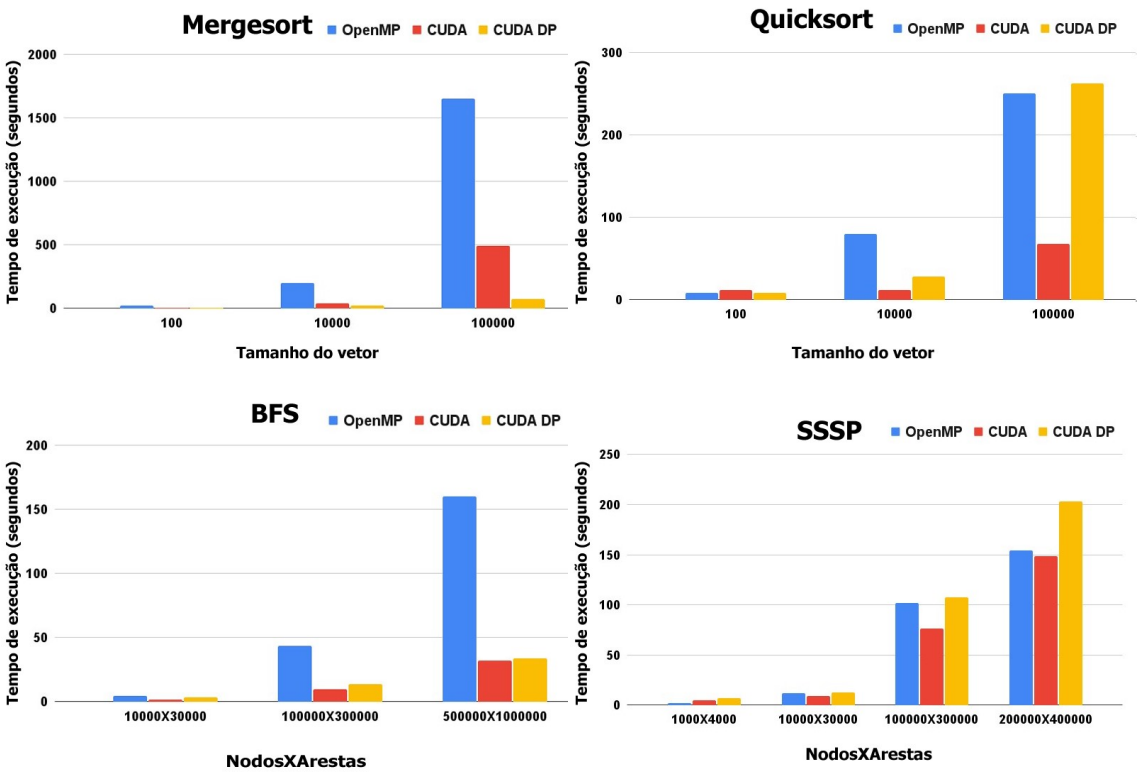

Figura 2. Tempo de execução

em relação as alternativas OpenMP e CUDA, o que ocorre provavelmente devido a falta de uma carga de trabalho suficientemente grande e balanceada, já que são feitas chamadas recursivas independente do número de adjacências. Além disso, destaca-se que não é possível verificar o melhor caminho local até o final da execução devido à natureza da consistência da memória (memória contígua), o que impossibilita algumas abordagens.

- O algoritmo *Mergesort* apresenta um comportamento distinto em comparação aos resultados anteriores, com a implementação CUDA DP sendo 23× mais rápida que a implementação OpenMP e 7× mais rápida que a implementação CUDA, além de consumir 22× e 7× menos energia, respectivamente. Este ganho é atribuído ao uso eficiente de DP, realizando as chamadas recursivas apenas quando há uma carga de trabalho suficientemente grande e balanceada.

Além disso, com bases nesses gráficos, é possível observar uma tendência direta, esperada, entre o tempo de execução e consumo de energia. Resumidamente, quanto maior o tempo de execução maior será o consumo. Para situações onde a carga de trabalho é expressiva, vale a pena gerar novos *kernels* sem a necessidade da intervenção da CPU, como é o caso da aplicação BFS, que mostrou menor consumo, para o caso mais significativo (200.000 × 400.000).

6. Considerações Finais

Neste trabalho realizamos uma avaliação de quatro algoritmos recursivos com diferentes características em relação ao padrão de acesso aos dados. Desta forma, cobri-

mos ordenação vetorial (acesso sequencial) e busca em grafos (acesso esparso), considerando diferentes volumes de dados de entrada e três interfaces distintas, OpenMP, CUDA e CUDA DP. Os resultados mostram que CUDA DP é uma extensão útil para a redução de tempo de execução e consumo de energia para aplicações recursivas como Mergesort. Como pode ser observado nos resultados, a implementação Mergesort CUDA DP supera as implementações CUDA e OpenMP em até 23× no tempo de execução e 7× no consumo de energia, respectivamente. Já para casos como o algoritmo BFS, apesar de haver um grande ganho de desempenho em relação à implementação OpenMP, não há uma diferença significativa entre as implementações CUDA e CUDA DP. Resumidamente, o desempenho da CUDA DP supera as implementações OpenMP e CUDA quando há cargas de trabalho balanceadas e suficientemente grandes, em aplicações que utilizam dados semi-estruturados ou organizados hierarquicamente.

Como trabalhos futuros, pretendemos incluir: (*i*) a avaliação de outras métricas como uso de memória na CPU e GPU, comportamento de cache e uso de GPU; (*ii*) inclusão de outras ferramentas de paralelismo como OpenACC e OpenCl; e (*iii*) adição de uma seção de estratégias optimizadas para cada implementação.

Agradecimentos. O trabalho foi apoiado pela CAPES (Código de Financiamento 001), RNP, CNPq (Universal 2023 - Projeto Nº 407827/2023-4) e FAPERGS (Edital 07/2021 PqG - Projeto Nº 21/2551-0002055-5).

Referências

Adinetz, A. (2014). Adaptive parallel computation with CUDA dynamic parallelism. *NVIDIA Corporation) Retrieved January*, 4:2016.

Araujo, G., Griebler, D., Rockenbach, D. A., Danelutto, M., and Fernandes, L. G. (2023). NAS Parallel Benchmarks with CUDA and beyond. *Software: Practice and Experience*, 53(1):53–80.

Bozorgmehr, B., Willemsen, P., Gibbs, J. A., Stoll, R., Kim, J.-J., and Pardyjak, E. R. (2021). Utilizing dynamic parallelism in CUDA to accelerate a 3D red-black successive over relaxation wind-field solver. *Environmental Modelling & Software*, 137:104958.

El Hajj, I., Gómez-Luna, J., Li, C., Chang, L.-W., Milojicic, D., and Hwu, W.-m. (2016). KLAP: Kernel launch aggregation and promotion for optimizing dynamic parallelism. In *49th Annual IEEE/ACM MICRO*, pages 1–12. IEEE.

Guo, G., Huang, T.-W., Lin, Y., and Wong, M. (2021). GPU-accelerated path-based timing analysis. In *58th ACM/IEEE DAC*, pages 721–726. IEEE.

Gupta, S. K., Singh, D. P., and Choudhary, J. (2023). New GPU Sorting Algorithm Using Sorted Matrix. *Procedia Computer Science*, 218:1682–1691.

Jarząbek, Ł. and Czarnul, P. (2017). Performance evaluation of unified memory and dynamic parallelism for selected parallel CUDA applications. *The Journal of Supercomputing*, 73:5378–5401.

Jin, C., de Supinski, B. R., Abramson, D., Poxon, H., DeRose, L., Dinh, M. N., Endrei, M., and Jessup, E. R. (2017). A survey on software methods to improve

the energy efficiency of parallel computing. *The International Journal of High Performance Computing Applications*, 31(6):517–549.

Khalilov, M. and Timoveev, A. (2021). Performance analysis of CUDA, OpenACC and OpenMP programming models on TESLA V100 GPU. In *Journal of Physics: Conference Series*, volume 1740, page 012056. IOP Publishing.

Memeti, S., Li, L., Pllana, S., Kołodziej, J., and Kessler, C. (2017). Benchmarking OpenCL, OpenACC, OpenMP, and CUDA: Programming Productivity, Performance, and Energy Consumption. In *Proceedings of the Workshop on Adaptive Resource Management and Scheduling for Cloud Computing*, pages 1–6.

Nana, R., Tadonki, C., Dokládal, P., and Mesri, Y. (2023). Energy Concerns with HPC Systems and Applications. *arXiv preprint arXiv:2309.08615*.

Navaux, P. O. A., Lorenzon, A. F., and da Silva Serpa, M. (2023). Challenges in High-Performance Computing. *Journal of the Brazilian Computer Society*, 29(1):51–62.

Nogueira, A. G. D., Lorenzon, A. F., Schepke, C., and Kreutz, D. (2024). Apêndice de Algoritmos do Trabalho: Análise de Desempenho e Consumo Energético de Aplicações Recursivas em Ambientes OpenMP, CUDA e CUDA DP. `https://github.com/kreutzd/arxiv/blob/main/sscad2024/sscad2024_trilhaprincipal_apendice1.pdf`.

NVIDIA (2022). *CUDA C++ Programming Guide*. Nvidia Corporation.

OpenMP Architecture Review Board (2022). OpenMP Application Program Interface.

O'brien, K., Pietri, I., Reddy, R., Lastovetsky, A., and Sakellariou, R. (2017). A survey of power and energy predictive models in HPC systems and applications. *ACM CSUR*, 50(3):1–38.

Palencia, J. and Rutten, P. (2024). Worldwide High-Performance Computing Server Forecast, 2023–2027: Enterprise Will Overtake HPC Labs. IDC. `https://www.idc.com/getdoc.jsp?containerId=US50525123`.

Park, S., Kim, H., Ahmad, T., Ahmed, N., Al-Ars, Z., Hofstee, H. P., Kim, Y., and Lee, J. (2022). SALoBa: Maximizing Data Locality and Workload Balance for Fast Sequence Alignment on GPUs. In *IEEE IPDPS*, pages 728–738. IEEE.

Plauth, M., Feinbube, F., Schlegel, F., and Polze, A. (2015). Using dynamic parallelism for fine-grained, irregular workloads: a case study of the n-queens problem. In *CANDAR*, pages 404–407. IEEE.

Quezada, F. A., Navarro, C. A., Romero, M., and Aguilera, C. (2023). Modeling GPU Dynamic Parallelism for self similar density workloads. *Future Generation Computer Systems*, 145.

Wang, J. and Yalamanchili, S. (2014). Characterization and analysis of dynamic parallelism in unstructured GPU applications. In *IEEE IISWC*, pages 51–60. IEEE.

Yi, X., Stokes, D., Yan, Y., and Liao, C. (2021). CUDAMicroBench: Microbenchmarks to Assist CUDA Performance Programming. In *IEEE IPDPSW*, pages 397–406. IEEE.