Relatório: Implementação do Algoritmo mppSort em GPU

Disciplina: CI1009 - Programação Paralela com GPUs
Autores: Cristiano Mendieta e Thiago Ruiz
Semestre: 2º Semestre de 2025

---

1. Introdução

Este relatório descreve a implementação do algoritmo de ordenação paralela **mppSort** para GPUs utilizando CUDA. O algoritmo foi baseado na especificação do Trabalho 2 da disciplina (versão 1.1) e implementa técnicas de programação paralela como histogramas, soma de prefixos (scan), particionamento de dados e ordenação.

1.1 Contexto

O algoritmo mppSort (multicore parallel partitioning sort) foi originalmente proposto para CPUs multicore por Cordeiro, Blanco e Zola (SBBD 2025). Este trabalho adapta as ideias centrais para a arquitetura de GPUs CUDA, aproveitando características específicas como:

- Milhares de threads executando concorrentemente
- Hierarquia de memória (global, shared, registradores)
- Operações atômicas em shared memory
- Modelo de programação SIMT (Single Instruction, Multiple Threads)

1.2 Objetivos

Os objetivos principais deste trabalho são:

1. **Implementar** uma versão paralela eficiente do mppSort em GPU seguindo a especificação fornecida
2. **Utilizar** técnicas de otimização como shared memory, operações atômicas e acessos coalescidos
3. **Validar** a corretude da implementação comparando com std::sort
4. **Avaliar** a performance comparando com thrust::sort
5. **Reportar** vazão em GElements/s para diferentes tamanhos de entrada (1M, 2M, 4M, 8M elementos)

1.3 Algoritmo mppSort - Visão Geral

O algoritmo divide a ordenação em etapas:

1. **Histograma:** Dividir o intervalo [nMin, nMax] em h bins e contar quantos elementos pertencem a cada bin
2. **Prefix Sum:** Calcular posições iniciais de cada bin no vetor de saída
3. **Particionamento:** Mover elementos para seus bins corretos no vetor de saída
4. **Ordenação Local:** Ordenar elementos dentro de cada bin

Esta estratégia permite paralelismo massivo nas fases de histograma e particionamento, com ordenação final independente por bin.

2. Descrição do Algoritmo

O algoritmo mppSort divide o problema de ordenação em várias etapas, cada uma implementada como um kernel CUDA:

2.1 Kernel 1: blockAndGlobalHisto

Assinatura: `blockAndGlobalHisto<<<nb, nt>>>(HH, Hg, h, Input, nElements, nMin, nMax)`

Este kernel é responsável por:

- Calcular histogramas locais por bloco de threads, armazenados na matriz HH (nb linhas × h colunas)
- Calcular o histograma global (vetor Hg com h elementos)
- Dividir o intervalo [nMin, nMax] em h bins de largura L = (nMax - nMin) / h

Implementação:

1. Cada bloco aloca um histograma local em shared memory (h elementos)
2. Threads processam elementos em grid-stride loop para garantir que todos os nElements sejam processados
3. Para cada elemento lido:
    - Calcula o bin correspondente: `bin = (valor - nMin) / L`
    - Incrementa atomicamente o contador do bin em shared memory
4. Ao final, cada thread escreve seu bin correspondente:
    - Na matriz HH (histograma por bloco)
    - No vetor Hg usando atomicAdd (histograma global)

Otimizações implementadas:

- Uso de shared memory para reduzir acessos à memória global (acesso a shared memory é ~100x mais rápido)
- Grid-stride loop para processar todos os elementos independentemente do número de threads
- Coalescência de acessos à memória no vetor Input (leituras sequenciais)
- Operações atômicas apenas em shared memory (mais rápido) e uma atualização final em global memory

2.2 Kernel 2: globalHistoScan

Assinatura: `globalHistoScan<<<1, nt>>>(Hg, SHg, h)`

Este kernel implementa uma soma de prefixos exclusiva (exclusive scan) sobre o histograma global Hg, produzindo o vetor SHg.

Implementação: Devido a questões de robustez e simplicidade, optou-se por uma implementação sequencial dentro de um único bloco:

1. Copia Hg para shared memory
2. Thread 0 executa o scan exclusivo sequencialmente:

```
SHg[0] = 0
for i = 1 to h-1:
    SHg[i] = SHg[i-1] + Hg[i-1]
```

3. Escreve o resultado de volta para global memory

Justificativa da abordagem sequencial:

- O histograma tem apenas h=256 elementos, um tamanho muito pequeno
- Implementações paralelas (como Blelloch scan) são mais complexas e propensas a erros
- O tempo deste kernel é desprezível comparado aos outros kernels
- Para h pequeno, o overhead de sincronização de um scan paralelo não compensa

Resultado: O vetor SHg contém as posições iniciais de cada bin no vetor de saída ordenado. Por exemplo, se Hg = [100, 200, 150], então SHg = [0, 100, 300].

2.3 Kernel 3: verticalScanHH

Assinatura: `verticalScanHH<<<h, nt3>>>(HH, PSv, h, nb)`

Este kernel calcula a soma de prefixos exclusiva por coluna (vertical) da matriz HH, produzindo a matriz PSv.

Implementação:

1. Lança h blocos, um para cada coluna do histograma
2. Cada bloco processa sua coluna (que tem nb linhas):
    - Copia a coluna de HH para shared memory
    - Executa scan exclusivo sequencial na coluna
    - Escreve o resultado na coluna correspondente de PSv
3. O número de threads por bloco (nt3) é ajustado para ser suficiente para nb elementos

Paralelização:

- Paralelismo entre colunas: cada coluna é processada independentemente em um bloco diferente
- Dentro de cada coluna: implementação sequencial por simplicidade (nb=10 é muito pequeno)

Resultado: PSv[b][c] contém o número total de elementos que os blocos 0 até b-1 colocaram no bin c. Isso permite que o bloco b saiba onde começar a colocar seus elementos no bin c.

2.4 Kernel 4: PartitionKernel

Assinatura: `PartitionKernel<<<nb, nt>>>(HH, SHg, PSv, h, Input, Output, nElements, nMin, nMax, nb)`

Este é o kernel mais complexo e mais importante do algoritmo, responsável pelo particionamento dos dados de entrada no vetor de saída, organizando-os por bins.

Implementação (seguindo especificação v1.1):

1. **Preparação em shared memory:**
    - Cada bloco b aloca vetor HLsh (Histograma por Linha em shared) com h elementos
    - Carrega PSv[b] (linha b de PSv) para HLsh
    - Carrega SHg para shared memory
2. **Cálculo de posições iniciais:**
    - Cada thread soma: HLsh[c] = PSv[b][c] + SHg[c]
    - Resultado: HLsh[c] contém a posição inicial onde o bloco b começará a inserir elementos no bin c
3. **Particionamento paralelo:**
    - Cada thread em grid-stride loop:
        - Lê elemento e da entrada: `e = Input[i]`
        - Calcula o bin: `f = (e - nMin) / L`
        - Obtém posição atomicamente: `p = atomicAdd(&HLsh[f], 1)`
        - Escreve na saída: `Output[p] = e`

Técnica chave:

- Uso de **atomicAdd em shared memory** (não global!) para alocação de posições
- Atomics em shared memory são ~10-20x mais rápidos que em global memory
- Bounds checking (`if (p < nElements)`) para evitar acessos inválidos
- A técnica de "privatização" com PSv permite que cada bloco trabalhe em regiões disjuntas de cada bin, reduzindo contenção

Resultado: Após este kernel, Output está particionado por bins (elementos do bin 0, depois bin 1, etc), mas ainda não ordenado dentro de cada bin.

2.5 Kernel 5: Ordenação por Bins

Após o particionamento, cada bin precisa ser ordenado internamente. A implementação adotada utiliza exclusivamente a biblioteca Thrust por questões de robustez

e corretude.

Implementação:

1. Copia Hg e SHg de volta para o host
2. Para cada bin i (de 0 a h-1):
    - Calcula início: `start = SHg[i]`
    - Calcula tamanho: `size = Hg[i]`
    - Se size > 0:
        - Cria device_ptr apontando para Output[start]
        - Chama `thrust::sort(ptr, ptr + size)`
        - Ordenação in-place na memória global

Justificativa para uso exclusivo do Thrust:

- Implementação do bitonicSort dos CUDA samples requer bins com tamanho potência de 2
- Bins reais têm tamanhos arbitrários (não são potências de 2)
- Thrust::sort é altamente otimizado e testado
- Confiabilidade: todos os testes passaram na verificação de corretude
- Simplicidade: menos código propenso a erros

Nota sobre performance: Embora bitonicSort em shared memory pudesse ser mais rápido para bins pequenos, a distribuição real dos dados resulta em bins de tamanhos variados. O overhead de verificar tamanhos, fazer padding, etc., não justificaria a complexidade adicional para este trabalho.

3. Detalhes de Implementação

3.0 Estrutura Geral do Programa

O programa principal (`mppSort.cu`) segue a seguinte estrutura:

```
main():
  1. Parse argumentos (nElements, h, nR)
  2. Gera dados no host e calcula nMin, nMax
  3. Aloca memória na GPU (Input, Output, HH, Hg, SHg, PSv)
  4. Copia Input para GPU

  5. Loop de nR repetições (medindo tempo com cudaEvent):
     a. Kernel 1: blockAndGlobalHisto
     b. Kernel 2: globalHistoScan
     c. Kernel 3: verticalScanHH
     d. Kernel 4: PartitionKernel
     e. Kernel 5: Loop ordenando cada bin com thrust::sort

  6. Calcula e reporta vazão do mppSort

  7. Benchmark do Thrust (loop nR repetições):
     - thrust::sort no vetor completo

  8. Calcula e reporta vazão do Thrust

  9. verifySort(): Compara resultado com std::sort

  10. Libera memória
```

Medição de tempo:

- Uso de `cudaEvent_t` para timing preciso na GPU
- `cudaEventRecord(start)` antes do loop
- `cudaEventRecord(stop)` após o loop
- `cudaEventSynchronize(stop)` para garantir conclusão
- `cudaEventElapsedTime(&time, start, stop)` para obter tempo em ms

3.1 Configuração de Execução

- **GPU:** NVIDIA GeForce GTX 750 Ti
    - Número de SMs: 5
    - Compute Capability: 5.0
- **Número de blocos (nb):** NP × 2 = 5 × 2 = **10 blocos**
- **Threads por bloco (nt): 1024 threads** (máximo permitido pela GPU conforme especificação)
- **Número de bins (h): 256** (conforme especificação, cabe em shared memory)
- **Shared memory:** Alocada dinamicamente para cada kernel:
    - Kernel 1: `h * sizeof(unsigned int)` = 1024 bytes
    - Kernel 2: `h * sizeof(unsigned int)` = 1024 bytes
    - Kernel 3: `nb * sizeof(unsigned int)` = 40 bytes por bloco

- Kernel 4: `2 * h * sizeof(unsigned int)` = 2048 bytes

3.2 Geração de Dados

Os dados de entrada são gerados conforme especificação:

```
unsigned int v = rand() * 100 + rand();
```

Exemplo de intervalo gerado (teste com 1M elementos):

- Intervalo de dados [nMin, nMax]: [10731, 4294956968]
- Largura do bin (L): 16777133

Os valores mínimo (nMin) e máximo (nMax) são calculados durante a geração no host, e a largura dos bins é:

```
L = (nMax - nMin) / h
```

Características dos dados gerados:

- Distribuição pseudo-aleatória cobrindo quase todo o range de unsigned int (32 bits)
- Cada elemento tem 32 bits, range de 0 a 4.294.967.295
- Seeds aleatórias garantem distribuição diferente a cada execução
- Bins tendem a ter tamanhos similares (distribuição aproximadamente uniforme)

3.3 Verificação de Corretude

A função `verifySort()` compara o resultado do mppSort com uma ordenação de referência usando `std::sort` do C++ STL:

1. Cria cópia do vetor de entrada no host
2. Ordena com std::sort (algoritmo QuickSort/IntroSort otimizado)
3. Copia o vetor Output da GPU para o host
4. Compara elemento por elemento
5. Retorna true se todos os elementos estão na mesma ordem

Resultado: Todos os testes (1M, 2M, 4M, 8M elementos) indicaram **"Ordenação correta!"**

4. Experimentos e Resultados

4.1 Ambiente de Testes

- **GPU:** NVIDIA GeForce GTX 750 Ti
- **CUDA Version:** 12.x (compatível com sm_50)
- **Número de SMs:** 5 Streaming Multiprocessors
- **Compute Capability:** 5.0
- **Compilador:** nvcc com flags `-arch=sm_50 -O3`
- **Data dos experimentos:** 10/11/2025

4.2 Parâmetros dos Experimentos

- **h (número de bins):** 256
- **nR (repetições):** 10
- **nb (blocos):** 10
- **nt (threads/bloco):** 1024
- **Elementos testados:** 1M, 2M, 4M, 8M onde M = 10^6 (não potências de 2, conforme especificação)
- **Medição:** cudaEvent para timing preciso de cada kernel
- **Repetições:** Cada teste executado 10 vezes, reportando tempo médio

4.3 Resultados de Performance

Nro Elementos	Vazão mppSort (GElements/s)	Vazão Thrust (GElements/s)	Speedup (Thrust/mppSort)	Tempo Médio mppSort (ms)
1.000.000	0,447	0,270	0,60x	2,24
2.000.000	0,307	0,285	0,93x	6,51
4.000.000	0,432	0,276	0,64x	9,27
8.000.000	0,539	0,305	0,56x	14,84

Observações importantes:

- **M = 10^6** conforme especificação (não são potências de 2)

- Speedup < 1 significa que **mppSort é mais rápido que Thrust**
- Todos os testes passaram na verificação de corretude
- Tempo inclui todos os 5 kernels: histogram, scans (2 e 3), partition, e sorting
- Vazão calculada como: `nElements / (tempo_médio_ms / 1000) / 1e9`

5. Análise dos Resultados

5.1 Performance Geral

Os resultados demonstram que **mppSort superou Thrust em todos os tamanhos testados**, com speedups variando de 0,56x a 0,93x (lembrando que speedup < 1 significa que mppSort é mais rápido):

- **1M elementos:** mppSort foi 1,66x mais rápido (speedup 0,60x = 0,270/0,447)
- **2M elementos:** Performance similar (speedup 0,93x), quase empate
- **4M elementos:** mppSort foi 1,56x mais rápido (speedup 0,64x)
- **8M elementos:** mppSort foi 1,77x mais rápido (speedup 0,56x = 0,305/0,539)

Interpretação: A estratégia de particionamento em bins com histogramas e uso intensivo de shared memory mostrou-se eficaz. O mppSort consegue melhor desempenho especialmente quando há mais dados para processar (8M elementos), sugerindo que o algoritmo escala melhor que o Thrust para esta GPU específica.

5.2 Escalabilidade

Analisando a vazão (throughput) em função do tamanho da entrada:

Tamanho	Vazão mppSort	Crescimento
1M	0,447 GE/s	baseline
2M	0,307 GE/s	-31% (queda)
4M	0,432 GE/s	+41% vs 2M
8M	0,539 GE/s	+25% vs 4M

Observação interessante: A queda de vazão em 2M elementos pode indicar um "sweet spot" negativo onde:

- O problema ainda não é grande o suficiente para esconder latências
- Há possível contenção nos atomics do particionamento

Para 4M e 8M, a vazão aumenta consistentemente, indicando melhor utilização da GPU com problemas maiores.

5.3 Eficiência dos Kernels

Baseado na implementação, a contribuição estimada de cada kernel para o tempo total:

1. **Kernel 1 (blockAndGlobalHisto):** ~25-30% do tempo
    - Lê todos os nElements da global memory
    - Grid-stride loop, bem paralelizado
    - Atomics em shared memory são rápidos
2. **Kernel 2 (globalHistoScan):** <1% do tempo
    - Apenas 256 elementos para processar
    - Execução sequencial é adequada
    - Tempo desprezível
3. **Kernel 3 (verticalScanHH):** <5% do tempo
    - Apenas h colunas com nb=10 linhas cada
    - Paralelizado por coluna
    - Trabalho total muito pequeno
4. **Kernel 4 (PartitionKernel):** ~30-40% do tempo
    - Lê e escreve todos os nElements
    - Atomics em shared memory para cada elemento
    - Potencial gargalo devido aos atomics
5. **Kernel 5 (Thrust sorting):** ~30-40% do tempo
    - Ordena cada bin individualmente
    - Múltiplas chamadas ao thrust::sort
    - Overhead de launches múltiplos

Gargalos identificados:

- **Partition kernel:** Atomics podem causar serialização dentro de cada bin
- **Multiple thrust::sort calls:** Overhead de lançar h kernels separados

5.4 Impacto do Número de Bins

Com h=256 bins fixo (conforme especificação), cada bin deve conter em média:

- 1M elementos: ~3.906 elementos/bin
- 8M elementos: ~31.250 elementos/bin

Bins maiores:

- Mais contenção nos atomics do Kernel 4
- Mais trabalho para o thrust::sort de cada bin

Bins menores (se testássemos h maior):

- Menos contenção, atomics mais eficientes
- Mais chamadas ao thrust::sort (overhead)
- Mais espaço para histogramas

O valor h=256 parece um bom equilíbrio para esta GPU e tamanhos testados.

6. Conclusões

Este trabalho implementou com sucesso o algoritmo **mppSort** para GPUs utilizando CUDA, alcançando os objetivos propostos na especificação:

6.1 Objetivos Alcançados

Implementação completa dos 5 kernels especificados:

- Kernel 1: blockAndGlobalHisto (histogramas por bloco e global)
- Kernel 2: globalHistoScan (scan do histograma global)
- Kernel 3: verticalScanHH (scan vertical por coluna)
- Kernel 4: PartitionKernel (particionamento com atomics em shared memory)
- Kernel 5: Ordenação por bins com thrust::sort

Verificação de corretude: Todos os testes (1M, 2M, 4M, 8M) passaram na verificação contra std::sort

Performance superior ao Thrust: mppSort foi 1,56x a 1,77x mais rápido em 3 dos 4 testes

Experimentos conforme especificação:

- Tamanhos: 1M, 2M, 4M, 8M elementos (M=10^6, não potências de 2)
- Parâmetros: h=256, nb=10, nt=1024, nR=10 repetições
- Vazão reportada em GElements/s

6.2 Técnicas de Programação Paralela Aplicadas

Este trabalho demonstrou aplicação prática de conceitos fundamentais:

1. **Shared Memory:** Uso intensivo para reduzir acessos à global memory
    - Histogramas locais (Kernel 1)
    - Buffers para scan (Kernels 2 e 3)
    - Contadores privatizados (Kernel 4)
2. **Operações Atômicas:** atomicAdd em shared memory para alocação eficiente de posições
3. **Grid-Stride Loops:** Processamento de todos os elementos independentemente do número de threads
4. **Prefix Sum (Scan):** Algoritmo fundamental para calcular posições de destino
5. **Coalescência:** Acessos sequenciais ao vetor Input para maximizar bandwidth
6. **Privatização:** Técnica de PSv para reduzir contenção entre blocos

6.3 Contribuições e Aprendizados

Principais insights obtidos:

- Atomics em shared memory são significativamente mais rápidos que em global memory
- Para problemas pequenos (h=256, nb=10), implementações sequenciais podem ser mais práticas
- Trade-off entre complexidade e corretude: simplicidade pode ser preferível
- Importance de bounds checking em operações com atomics

Limitações identificadas:

- Implementação sequencial dos scans não escala para h ou nb muito grandes
- Uso exclusivo de thrust::sort pode não ser ótimo para todos os tamanhos de bins

6.4 Resultado Final

O algoritmo mppSort implementado demonstrou ser **competitivo e eficiente** para a GPU testada, superando thrust::sort na maioria dos casos. A implementação segue fielmente a especificação v1.1 do trabalho, utilizando técnicas modernas de programação paralela em GPU.