Relatório Técnico: CUDA-mppSort

Gabriel Pucci Bizio

16 de novembro de 2025

1 Objetivos

Os objetivos deste trabalho são:

- Implementar um algoritmo eficiente de ordenação paralela usando particionamento múltiplo;
- Medir o tempo e a vazão do algoritmo completo;
- Medir o tempo de execução dos diversos kernels utilizados para obter o vetor ordenado;
- Utilizar o algoritmo de ordenação da biblioteca *thrust* como referência para comparação de desempenho.

2 Funcionamento

O algoritmo *CUDA-mppSort* funciona dividindo o vetor de entrada em segmentos (ou partições) de amplitudes iguais. Cada chave do vetor é mapeada para um desses segmentos, de forma que:

1. as partições são ordenadas globalmente;
2. os elementos internos de cada segmento são ordenados paralelamente;
3. múltiplas partições são ordenadas simultaneamente, explorando o paralelismo da GPU.

2.1 Detalhes dos kernels

O algoritmo requer a implementação de cinco kernels, que cumprem as seguintes funções, de forma resumida:

1. **Kernel 1**: constrói um histograma que armazena a quantidade de elementos que cada partição irá comportar. Também obtém uma matriz que mapeia a quantidade de elementos em cada partição que cada bloco haverá de ordenar na fase de partição.

2. **Kernel 2**: realiza uma operação de redução (*scan*) no histograma obtido pelo Kernel 1, a fim de obter os índices correspondentes ao começo de cada partição.

3. **Kernel 3**: realiza o *scan* vertical da matriz do mapa de blocos–partições, obtendo o índice interno a cada partição no qual cada bloco deverá colocar os seus elementos.

4. **Kernel 4**: utiliza as estruturas previamente obtidas para organizar as partições de forma ordenada no vetor. Depois desse kernel, todas as partições têm seus elementos internos juntos, com as partições ordenadas em sequência, mas os elementos internos dos segmentos ainda estão desordenados.

5. **Kernel 5**: percorre os segmentos de forma paralela, ordenando internamente `NBLOCOS` partições por passo.

3 Implementação

Na implementação do *CUDA-mppSort*, foi observado que entradas muito grandes ou um número de partições excessivo podem gerar problemas e resultar em vetores ordenados incorretamente.

Quando as partições são muito grandes e estouram o limite de memória compartilhada dos blocos, o *thrust sort* é chamado para ordená-las, adicionando um *overhead* significativo. Esse overhead faz com que a implementação perca desempenho em relação ao algoritmo de referência em algumas configurações.

Mesmo quando o resultado final está incorreto em função desses erros, os passos da ordenação ainda são realizados. Tanto essas saídas incorretas quanto as entradas menores, que funcionam corretamente, apresentam desempenho competitivo e podem superar o tempo de execução do algoritmo *thrust*, desde que o tamanho das partições não seja maior que o limite da memória compartilhada.

Nesse relatório são mostrado o resultado de um teste para `10000` elementos e `16` partições, executado com `100` repetições.

4 Ambiente de teste

Os testes foram feitos utilizando a máquina *Orval* do DInf, equipada com uma GPU GTX 750 Ti.

5 Resultados

Nesta seção são apresentados os tempos dos cinco kernels que compõem o algoritmo completo, bem como a comparação entre o tempo total e a vazão do *mppSort* e do *thrust*.

5.1 Tempos dos kernels

A Tabela 1 apresenta os tempos médios dos cinco kernels do algoritmo, considerando a média sobre 100 execuções.

Tabela 1: Tempos médios dos kernels do *CUDA-mppSort* (média de 100 execuções).

Kernel	**Tempo médio**
Kernel 1 (*blockAndGlobalHistogram*)	0,015 s
Kernel 2 (*globalHistogramScan*)	0,009 s
Kernel 3 (*verticalScanHH*)	0,009 s
Kernel 4 (*partitionKernel*)	0,015 s
Kernel 5 (*bitonicSort + merge*)	0,083 s
Tempo total do mppSort	0,131 s

5.2 Comparação entre mppSort e thrust

A Tabela 2 apresenta a comparação entre o *mppSort* e o algoritmo de referência da biblioteca *thrust*, em termos de tempo de execução e vazão.

Tabela 2: Comparação entre o *mppSort* e o *thrust* em tempo e vazão.

Algoritmo	**Tempo**	**Vazão**	**Aceleração**
Thrust sort	0,201 ms	0,050 GElements/s	
mppSort	0,131 s	0,076 GElements/s	1.54x