Relatório: Implementação do CUDA-MPPSort

1 Objetivos

Implementar um algoritmo eficiente de ordenação paralela usando particionamento múltiplo. Medir o tempo e a vazão do algoritmo completo, bem como dos diferentes kernels utilizados para obter o vetor ordenado, tomando o algoritmo de ordenação da biblioteca `thrust` como referência.

2 Funcionamento

O algoritmo funciona dividindo o vetor em segmentos (partições) de amplitudes iguais, mapeando as chaves do vetor para esses segmentos, ordenando as partições globalmente e, em seguida, ordenando os elementos internos de cada partição paralelamente. Várias partições podem ser ordenadas simultaneamente.

Detalhes

O algoritmo requer a implementação de cinco kernels, com as seguintes funções resumidas:

1. Construir um histograma com a quantidade de elementos que cada partição comporta. Também gera uma matriz que mapeia quantos elementos em cada partição cada bloco deverá ordenar na fase de particionamento.

2. Realizar uma operação de redução (scan) no histograma produzido pelo kernel 1, obtendo os índices iniciais de cada partição.

3. Realizar o scan vertical da matriz bloco–partição, obtendo o índice interno de cada partição em que cada bloco deve inserir seus elementos.

4. Utilizar as estruturas anteriores para organizar as partições de forma ordenada no vetor. Após este passo, todas as partições têm seus elementos reunidos e estão em ordem, mas os elementos internos ainda não se encontram ordenados.

5. Percorrer os segmentos de forma paralela, ordenando internamente `NBLOCOS` partições por passo.

3 Implementação do Novo Bitonic Sort

O novo *Bitonic Sort* consegue ordenar vetores de tamanho arbitrário, sem depender da quantidade de threads por bloco. Cada thread recebe a responsabilidade de comparar um

número X de pares de elementos, onde X corresponde à metade do tamanho do vetor dividido pela quantidade de threads (assumindo que o número de threads seja menor que metade do tamanho do vetor).

Para vetores que não são múltiplos de 2, valores de *padding* são adicionados na memória compartilhada, e o algoritmo executa usando o próximo múltiplo de 2.

Devido a restrições dos kernels utilizados no trabalho anterior, o número máximo de segmentos é igual ao número máximo de threads por bloco, isto é, 1024 threads.

4 Ambiente de Teste

Os testes foram realizados na máquina *Orval* do DInf, equipada com uma GTX 750 Ti. Utilizou-se o maior número possível de threads para o `blockBitonicSort`, isto é, 1024 threads, com média de 100 repetições por experimento.

5 Resultados

Tempos

Table 1: Tempo de execução (ms)

Elementos	**1M**	**2M**	**4M**	**8M**
MPP	6.2	8.3	17.1	54.919
Thrust	2.4	4.5	8.8	17.287
Speedup	0.39x	0.55x	0.51x	0.31x

Vazão

Table 2: Vazão (GElements/s)

Elementos	**1M**	**2M**	**4M**	**8M**
MPP	0.160	0.241	0.233	0.146
Thrust	0.406	0.437	0.454	0.463