Relatório Técnico – Trabalho 1

Disciplina: CI1009 – Programação Paralela com GPUs

Professor: W. Zola

Instituição: UFPR

Semestre: 2º/2025

Data: Outubro de 2025

Autores: Cristiano Creppo Mendieta e Thiago Ruiz

1. Objetivo

O objetivo deste trabalho é implementar e comparar diferentes estratégias de **kernels CUDA** para realizar a operação de **redução paralela**, especificamente para encontrar o valor máximo em vetores de ponto flutuante (float). Foram implementadas duas abordagens principais (Many-threads e Persistente) e os resultados foram comparados com a implementação de referência da biblioteca **Thrust**.

2. Metodologia

2.1. Especificações de Hardware

- GPU: NVIDIA GeForce GTX 750 Ti

2.2. Configuração dos Experimentos

Parâmetro	Valor
Tamanhos de entrada	10■ e 16×10■ elementos
Número de repetições (nR)	30
Tipo de dado	float (32 bits)
Threads por bloco	1024
Blocos (kernel persistente)	32

2.3. Geração dos Dados de Entrada

```
for (int i = 0; i < nTotalElements; i++) {
    int a = rand();  // Número pseudo-aleatório [0, RAND_MAX]
    int b = rand();  // Número pseudo-aleatório [0, RAND_MAX]
    float v = a * 100.0 + b;
    Input[i] = v;
}
```

2.4. Implementação dos Kernels

Kernel 1 – Many-threads (reduceMax)

Implementação clássica de redução paralela com as seguintes fases: cada thread carrega um elemento para memória compartilhada; redução em árvore (tree reduction) no bloco; a thread 0 grava o resultado parcial; múltiplas invocações do kernel até obter um único valor.

Kernel 2 – Persistente (reduceMax_atomic_persist)

Versão otimizada com kernel persistente: cada thread processa múltiplos elementos (coalesced access), redução intra-bloco usando atomicMax em shared memory e thread 0 de cada bloco aplicando atomicMax na memória global.

Thrust

Utilizou-se a função thrust::max_element como referência pela sua implementação altamente otimizada.

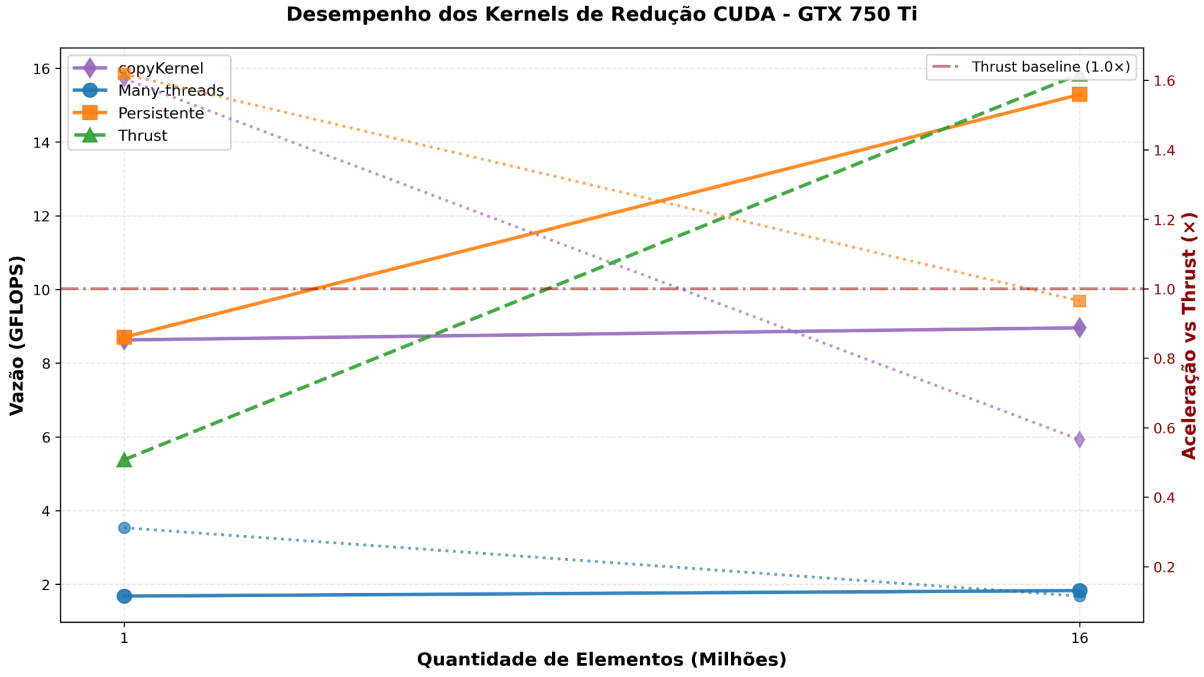
3. Resultados Experimentais

3.1. Tabela de Resultados

Teste	Kernel	Elementos	Tempo Médio (ns)	Vazão (GFLOPS)	Aceleração vs Thrust
1M	copyKernel	1.000.000	115.856	8,630	1,60×
1M	Many-threads	1.000.000	595.779	1,678	0,31×
1M	Thrust	1.000.000	185.944	5,378	1,00×
1M	Persistente	1.000.000	114.926	8,701	1,62×
16M	Many-threads	16.000.000	8.750.399	1,828	0,12×
16M	Thrust	16.000.000	1.009.741	15,846	1,00×
16M	copyKernel	16.000.000	1.785.718	8,960	0,57×
16M	Persistente	16.000.000	1.046.368	15,291	0,96×

3.2. Gráfico de Desempenho

Figura 1 – Desempenho dos Kernels de Redução CUDA (GTX 750 Ti).



4. Análise e Discussão

4.1. copyKernel – Baseline de Largura de Banda

Tamanho	Vazão (GFLOPS)	Largura de banda (GB/s)	Eficiência
1M	8,630	69,05	79,9%
16M	8,960	71,68	83,0%

Os valores indicam operação próxima ao limite teórico (86,4 GB/s), confirmando que o copyKernel é uma operação memory-bound eficiente.

4.2. Kernel Many-threads

O kernel Many-threads apresentou desempenho significativamente inferior (≈1.7–1.8 GFLOPS). As razões principais incluem múltiplas invocações do kernel, overhead de sincronização e leituras/gravações repetidas na memória global.

4.3. Kernel Persistente

O kernel Persistente obteve bom desempenho, com 8,701 GFLOPS (1M) e 15,291 GFLOPS (16M). Beneficia-se de uma única invocação, acessos coalescidos e uso eficiente de atomics em shared memory.

4.4. Thrust

A biblioteca Thrust apresentou desempenho muito bom, especialmente para 16M elementos (15,846 GFLOPS), provavelmente devido a heurísticas de balanceamento de carga, unrolling e otimizações específicas da arquitetura.

4.5. Escalabilidade

Kernel	Vazão 1M	Vazão 16M	Ganho
copyKernel	8,63	8,96	1,04×
Many-threads	1,68	1,83	1,09×
Persistente	8,70	15,29	1,76×
Thrust	5,38	15,85	2,95×

5. Conclusões

• copyKernel define o limite de largura de banda, alcançando 80–83% da capacidade teórica.
• Kernel Persistente alcança eficiência próxima ao copyKernel para 1M elementos, indicando que está limitado por banda e não por computação.
• Para vetores menores, o Persistente supera o Thrust (1,62×) devido à redução de overhead.
• Para vetores maiores, o Thrust apresenta leve vantagem graças a otimizações de cache e escalabilidade.
• Many-threads é inadequado para este problema, apresentando penalidades significativas de desempenho.
• Operações atômicas em shared memory mostraram-se eficientes e sem impacto negativo relevante.

6. Decisões de Implementação e Limitações

6.1. Interface de Execução

```
./cudaReduceMax <nTotalElements>           # Kernel Many-threads
./cudaReduceMax <nTotalElements> <nBlocks> # Kernel Persistente

Nota: o número de repetições nR = 30 está fixo na constante NTIMES no código-fonte.
```

7. Arquivos Entregues

• Código-fonte: cudaReduceMax.cu, helper_cuda.h, chrono.c
• Scripts: compila.sh, executar_experimentos.sh, scripts/processar_resultados_completo.py
• Dados experimentais: resultados/dados_*.txt e resultados/resultados_completos.csv
• Gráficos: resultados/plots/resultado_final.png

8. Reprodutibilidade dos Experimentos

```
# Compilar o programa
make

# Executar todos os experimentos
make test

# Execuções individuais
./copyKernel 1000000 > resultados/dados_1M_copy.txt
./cudaReduceMax 1000000 > resultados/dados_1M_many.txt
./cudaReduceMax 1000000 32 > resultados/dados_1M_persist.txt
./copyKernel 16000000 > resultados/dados_16M_copy.txt
./cudaReduceMax 16000000 > resultados/dados_16M_many.txt
./cudaReduceMax 16000000 32 > resultados/dados_16M_persist.txt

# Processar resultados e gerar gráficos
python3 scripts/processar_resultados_completo.py
```