CS542200 Parallel Programming
Homework 3: All-Pairs Shortest Path

Kai-Yuan Jeng 鄭凱元
113062529
kaiyuanjeng@gapp.nthu.edu.tw

1 Implementation

This section details the implementation of the All-Pairs Shortest Path (APSP) problem on CPU, single-GPU, and multi-GPU platforms.

1.1 CPU Version

We adopt the blocked Floyd-Warshall algorithm to solve the APSP problem on the CPU. Unlike the classical Floyd-Warshall algorithm, which utilizes three nested loops over (i, j, k) to relax all vertex pairs, the blocked version partitions the (i, j) space into $B \times B$ tiles and updates the matrix in three distinct phases per round. This blocking strategy significantly improves cache locality and facilitates parallelism.

In each round k, Phase 1 updates the pivot block (the diagonal block corresponding to the current intermediate vertices). Phase 2 subsequently updates all blocks in the pivot row and pivot column using the data from the newly updated pivot block. Finally, Phase 3 updates the remaining off-diagonal blocks by combining results from the corresponding pivot row and column blocks. Although the computations are grouped at the block level, every element conceptually follows the standard Floyd-Warshall recurrence.

The key observation for parallelization is that within a fixed round, the update $\text{dist}[i][j] = \min(\text{dist}[i][j], \text{dist}[i][k] + \text{dist}[k][j])$ depends only on values finalized in previous phases or rounds, and is independent of other updates within the same phase. Consequently, we parallelize the internal loops of each block using OpenMP. As shown below, we apply the `omp parallel for` directive to the row index i within the block update function:

```
#pragma omp parallel for num_threads(NUM_THREAD) schedule(dynamic, 12)
for (int i = block_internal_start_x; i < block_internal_end_x; ++i) {
    for (int j = block_internal_start_y; j < block_internal_end_y; ++j) {
        if (D[i][k] + D[k][j] < D[i][j]) {
            D[i][j] = D[i][k] + D[k][j];
        }
    }
}
```

This approach allows different threads to concurrently update disjoint subsets of matrix elements within the same phase, preserving correctness while effectively utilizing multiple CPU cores.

1.2 Single GPU Version

For the single-GPU implementation, we first pad the $V \times V$ distance matrix so that V becomes a multiple of `BLOCKING_FACTOR`. The matrix is then partitioned into tiles of size `BLOCKING_FACTOR` $\times$ `BLOCKING_FACTOR` (set to 64 in our implementation). Each tile contains $64 \times 64 = 4096$ elements. We assign one CUDA thread block with $32 \times 32 = 1024$ threads to process a single tile. Consequently, each CUDA thread is responsible for computing four elements. We conceptually divide each tile into

four quadrants of size HALF_BLOCK $\times$ HALF_BLOCK (where HALF_BLOCK $= 32$). A thread with local index (ty, tx) updates the positions (ty, tx), $(ty + \texttt{HALF\_BLOCK}, tx)$, $(ty, tx + \texttt{HALF\_BLOCK})$, and $(ty + \texttt{HALF\_BLOCK}, tx + \texttt{HALF\_BLOCK})$ within the tile.

Each phase of the blocked Floyd-Warshall algorithm corresponds to a separate CUDA kernel:

- **Phase 1**: Updates the pivot tile on the diagonal, covering $\texttt{BLOCKING\_FACTOR}^2 = 4096$ elements. This kernel is launched with a single CUDA block.
- **Phase 2**: Updates all blocks in the pivot row and pivot column. Since the grid dimension is round, this phase targets the row and column intersecting the pivot, covering a total of approximately $2 \times \texttt{round} \times \texttt{BLOCKING\_FACTOR}^2$ elements (excluding the pivot block processed in Phase 1). We implement this by launching two separate kernels, each utilizing a 1D grid of round blocks, where each thread block is responsible for one tile. To prevent redundant computation, logic within the kernel checks the block's coordinates (blockIdx) and skips execution if the block corresponds to the pivot tile already handled in Phase 1.
- **Phase 3**: Updates all remaining off-diagonal tiles. This kernel is launched with a 2D grid of round $\times$ round blocks to cover the entire tiled matrix. Logic within the kernel checks the coordinates of each block and skips execution if the block belongs to the pivot row or pivot column, ensuring that only the required off-diagonal tiles are updated.

Detailed optimization techniques, such as shared memory tiling to reduce global memory latency, will be discussed in subsequent sections.

1.3 Multi-GPU Version

For the multi-GPU implementation, we utilize two OpenMP host threads to manage kernel launches. The workload, i.e., the $V \times V$ distance matrix, is statically partitioned into upper and lower halves, which are assigned to two GPUs using cudaSetDevice(). To simplify implementation, each GPU allocates memory for the complete matrix, though it only computes updates for its assigned partition.

The kernel functions remain largely similar to the single-GPU version, with adjustments in the Phase 2 column update and Phase 3 to account for global row indices within the partitioned space. Although workload distribution is slightly asymmetric—only the GPU containing the current pivot row executes Phase 1 and the Phase 2 row update—this imbalance is negligible because the dominant computational cost lies in Phase 3, which is distributed across both devices. Both GPUs must execute the Phase 2 column update and Phase 3 for their respective partitions.

To optimize inter-device communication, we enable peer-to-peer (P2P) access using cudaMemcpyPeer(). Synchronization is required at each round because Phase 3 computations depend on the updated pivot row and column from Phase 2. Specifically, after the GPU responsible for the current pivot row completes the Phase 2 row kernel, it transfers the updated pivot row data to the peer GPU. Once this transfer is complete, both GPUs possess the necessary dependencies to concurrently execute the Phase 2 column update and Phase 3.

2 Profiling Results

We profile our submitted CUDA implementation (hw3-2.cu) using the NVIDIA nvprof tool on a GeForce GTX 1080 GPU. The resulting kernel-level metrics are summarized in Table 1.

Theoretical Peak Calculation The theoretical peak values in Table 2 are derived from the hardware specifications of the NVIDIA GeForce GTX 1080 obtained via the deviceQuery tool.

- **Global Memory Bandwidth:** With a memory clock rate of 5005 MHz and a 256-bit memory bus width, the theoretical bandwidth is calculated as

$$\text{BW}_{\text{global}} = \text{Mem Clock} \times 2 \times \frac{\text{Bus Width}}{8} = 5005\ \text{MHz} \times 2 \times 32\ \text{B} \approx 320.32\ \text{GB/s},$$

  where the factor of 2 accounts for the Double Data Rate (DDR) effective transfer relative to the reported memory clock.

Table 1: Detailed Profiling Metrics Across Kernels

Metric	Stat	Phase 1	Phase 2 (Row)	Phase 2 (Col)	Phase 3 (Biggest)
Achieved Occupancy	Min	0.497390	0.839448	0.841107	0.907871
	Max	0.497507	0.904902	0.913099	0.910928
	Avg	0.497447	0.874727	0.877634	**0.909543**
SM Efficiency	Min	4.31%	74.11%	74.24%	99.45%
	Max	4.39%	90.88%	90.66%	99.64%
	Avg	4.37%	86.31%	86.78%	**99.55%**
Shared Mem Load	Min	96.16 GB/s	1510.51 GB/s	1509.09 GB/s	3282.58 GB/s
	Max	98.79 GB/s	1911.87 GB/s	1825.05 GB/s	3322.45 GB/s
	Avg	97.28 GB/s	1761.68 GB/s	1780.37 GB/s	**3301.02 GB/s**
Shared Mem Store	Min	48.45 GB/s	1533.74 GB/s	1532.31 GB/s	136.77 GB/s
	Max	49.78 GB/s	1941.29 GB/s	1853.13 GB/s	138.44 GB/s
	Avg	49.02 GB/s	1788.78 GB/s	1807.76 GB/s	137.54 GB/s
Global Mem Load	Min	763.31 MB/s	46.48 GB/s	46.43 GB/s	205.16 GB/s
	Max	784.19 MB/s	58.83 GB/s	56.16 GB/s	207.65 GB/s
	Avg	772.20 MB/s	54.21 GB/s	54.78 GB/s	**206.31 GB/s**
Global Mem Store	Min	763.31 MB/s	23.24 GB/s	23.22 GB/s	68.39 GB/s
	Max	784.19 MB/s	29.41 GB/s	28.08 GB/s	69.22 GB/s
	Avg	772.20 MB/s	27.10 GB/s	27.39 GB/s	**68.77 GB/s**

Table 2: Performance Comparison: Measured (Phase 3) vs. Theoretical Peak

Metric	Measured (Avg)	Theoretical Peak	Utilization
Global Memory Throughput[1]	275.08 GB/s	320.32 GB/s	85.88%
Shared Memory Throughput[2]	3438.56 GB/s	4439.04 GB/s	77.46%
Achieved Occupancy	90.95%	100.00%	90.95%

[1] Sum of global load (206.31 GB/s) and store (68.77 GB/s).
[2] Sum of shared load (3301.02 GB/s) and store (137.54 GB/s).

- **Shared Memory Bandwidth:** The GTX 1080 (Pascal architecture) contains 20 Streaming Multiprocessors (SMs). Each SM features 32 shared memory banks with a 4-byte width (32-bit), capable of processing one transaction per clock cycle. Using the maximum boost clock of 1734 MHz, the theoretical shared memory bandwidth is

$$\mathrm{BW}_{\text{shared}} = \mathrm{SMs} \times \mathrm{Banks} \times \mathrm{Width} \times \mathrm{Clock} = 20 \times 32 \times 4\ \mathrm{B} \times 1.734\ \mathrm{GHz} \approx 4439.04\ \mathrm{GB/s}.$$

Comparing our measured results with these theoretical peaks, our implementation achieves approximately 85.9% of the available global memory bandwidth and 77.5% of the shared memory throughput, indicating highly efficient memory utilization and effective cache blocking.

3 Experiment and Analysis

This section presents a systematic evaluation of our CUDA implementations, incorporating various optimization strategies and detailed profiling analysis.

System Specification The experiments were conducted on the **Apollo** cluster's GPU compute node. The operating environment is a virtualized instance running on a 64-bit Linux system. System specifications were retrieved using `lscpu` and `./deviceQuery`.

- **CPU:** The node is powered by an **Intel Xeon Silver 4210** processor with a base frequency of 2.20 GHz. The experimental instance is configured with **4 vCPUs**. The architecture features a 32 KiB L1d cache, 32 KiB L1i cache, and a 1 MiB L2 cache per core, along with a shared L3 cache (reported as 16 MiB in the virtualized environment).

Table 3: Theoretical Peak Performance Specifications (GTX 1080)

Specification	Value	Derivation
CUDA Cores	2560	Hardware Spec
Max Clock Rate	1734 MHz	`deviceQuery` Output
Ops per Cycle	2	IMAD (Multiply-Add)
Theoretical Integer Peak	**8878.08 GOPS**	$2560 \times 1.734 \times 2$
Theoretical Global Mem BW	320.32 GB/s	$5005 \times 256\text{-bit} \times 2/8$

- **GPU:** The system is equipped with an **NVIDIA GeForce GTX 1080** GPU based on the **Pascal** architecture (Compute Capability 6.1). It features **2560 CUDA cores**, 20 Streaming Multiprocessors (SMs), and **8 GB of GDDR5X** global memory. The 256-bit memory interface operates at a clock rate of 5005 MHz, yielding a theoretical bandwidth of 320.3 GB/s.
- **Software Environment:** The system runs NVIDIA Driver version 560.35.03 and **CUDA 12.6**. All code was compiled using `nvcc` with optimization flags appropriate for the hardware architecture.

3.1 Impact of Blocking Factor

Experimental Setup and Methodology This experiment investigates how different blocking factors (8, 16, 32, and 64) affect Integer GOPS (Giga Operations per Second) and memory bandwidth efficiency. We evaluate the public testcase `c21.1` ($V = 5000$). As detailed in Section 1.2, a blocking factor of 64 is the maximum feasible size before exceeding the 48 KB Shared Memory capacity per thread block. To ensure the highest accuracy for both performance timing and hardware metrics, we employed a **dual-stage execution strategy**:

1. **Performance Run:** The kernel is executed without instrumentation to measure the precise wall-clock time, avoiding the serialization overhead introduced by profiling tools.
2. **Profiling Run:** The kernel is re-executed with `nvprof` to capture detailed hardware metrics (e.g., throughput, occupancy). These rate-based metrics remain valid despite the instrumentation overhead.
3. **GOPS Calculation:** Using the precise timing (T_{sec}) from the Performance Run, we calculate the Integer GOPS based on the algorithmic complexity of Floyd-Warshall ($\mathcal{O}(V^3)$):

$$\text{GOPS} = \frac{2 \times V^3}{T_{\text{sec}} \times 10^9}$$

The factor of 2 accounts for the two integer operations (one addition and one comparison/min) performed in the innermost loop for each element update.

Theoretical Compute Throughput Calculation To establish a baseline for compute-bound analysis, we calculate the theoretical peak integer throughput (INT32 GOPS) of the GTX 1080. The device features 2560 CUDA cores and a maximum boost clock of 1734 MHz. Assuming the use of Integer Multiply-Add (IMAD) instructions, which perform two operations (multiply and add) per clock cycle per core, the theoretical peak is:

$$\text{GOPS}_{\text{peak}} = N_{\text{cores}} \times f_{\text{clock}} \times 2 = 2560 \times 1.734\ \text{GHz} \times 2 \approx 8878.08\ \text{GOPS}$$

Table 3 summarizes the theoretical peak values used for our utilization analysis.

Effective Peak Analysis While the absolute peak is ~ 8878 GOPS (IMAD), the Floyd-Warshall algorithm primarily utilizes integer addition (IADD) and comparison (ICMP/IMIN), which typically execute at 1 op/cycle per core. Therefore, the *algorithm-specific effective peak* is approximately half of the IMAD peak:

$$\text{GOPS}_{\text{effective peak}} = 2560 \times 1.734\ \text{GHz} \times 1 \approx \mathbf{4439}\ \text{GOPS}$$

This effective peak serves as a more realistic upper bound for our specific workload.

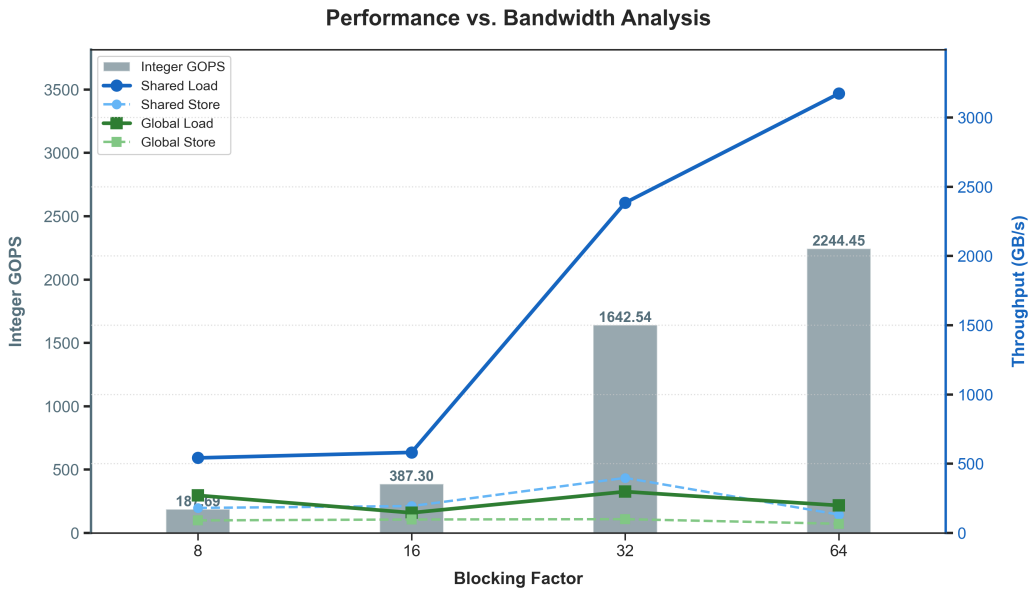

Figure 1: Performance (GOPS) and Bandwidth Analysis vs. Blocking Factor. Note the high correlation between Shared Memory Throughput and GOPS.

Results and Analysis The performance results are presented in Figure 1.

1. **Compute Performance:** A blocking factor of 64 achieves the highest performance of **2244 GOPS (2.24 TOPS)**. This represents approximately **50.6%** of the effective hardware limit (4439 GOPS). Achieving over 50% of the theoretical arithmetic peak on a memory-bound algorithm like Floyd-Warshall indicates highly efficient latency hiding.
2. **Blocking Factor Scaling:** Increasing the blocking factor from 8 to 64 results in a massive $12\times$ speedup (187 to 2244 GOPS), as larger tiles significantly improve data reuse.
3. **Memory Bandwidth Correlation:** The Shared Memory Load Throughput (blue line) correlates strongly with GOPS. At BF64, total shared memory throughput (load + store) reaches roughly **3305 GB/s**, while Global Memory throughput remains steady around **264 GB/s**. This $12.5\times$ gap confirms the effectiveness of our tiling strategy: for every byte fetched from global memory, the kernel performs over 10 accesses to the fast on-chip Shared Memory.

3.2 Macro Optimization Strategies

Experimental Setup and Methodology The primary bottleneck in CUDA programming often stems from memory operations. This experiment elucidates the rationale behind our specific memory optimization choices on the GPU. We investigate three critical factors affecting overall performance:

1. **Memory Hierarchy:** Whether the implementation utilizes on-chip Shared Memory or relies solely on off-chip Global Memory.
2. **Blocking Factor:** The size of the algorithmic tiles, which determines the granularity of the blocked Floyd-Warshall computation.
3. **Thread Coarsening:** Whether a single CUDA thread is responsible for one element or multiple elements (in our case, 4). Coarsening increases the workload per thread, potentially improving instruction-level parallelism and amortization of overheads.

We evaluate a total of 8 configurations across the entire "performance set" of testcases (ranging from `p11k1` to `p40k1`).

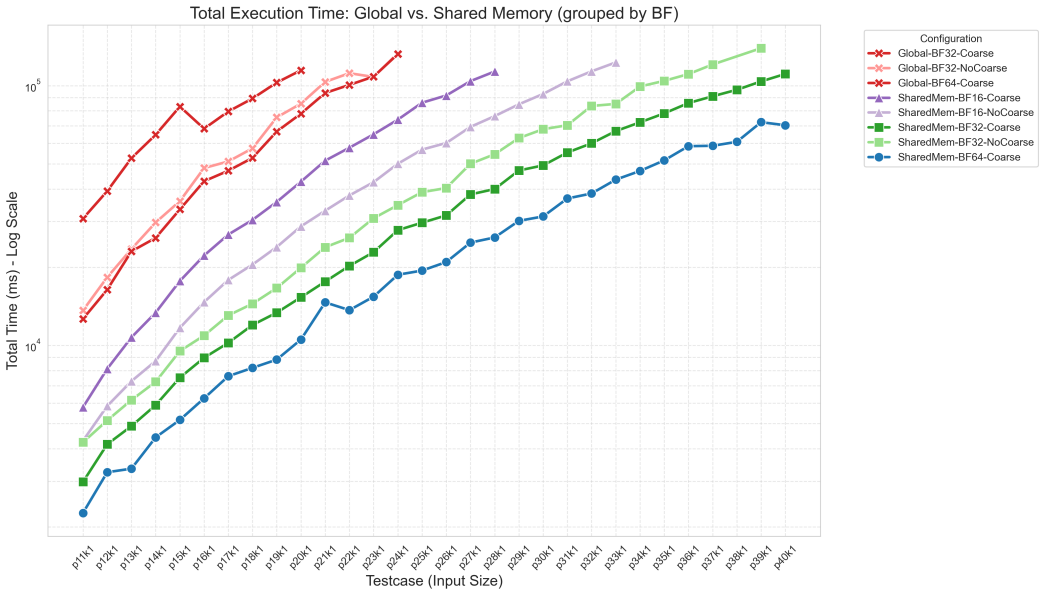

Figure 2: Performance Comparison of Macro Optimization Strategies. The optimal configuration (Blue) combines Shared Memory, Blocking Factor 64, and Thread Coarsening.

Results and Analysis The performance results are summarized in Figure 2.

- **Scalability:** First, we observe that execution times scale linearly on the logarithmic plot as testcase size increases, indicating consistent scaling behavior across all implementations. Note that some unoptimized configurations exceeded the system time limit for larger testcases.
- **Global vs. Shared Memory:** The configurations relying solely on Global Memory (red lines) exhibit the poorest performance. This highlights the critical superiority of utilizing Shared Memory to capture data locality and reduce high-latency DRAM accesses.
- **Impact of Blocking Factor:** The purple, green, and blue groups differ by their blocking factors (16, 32, and 64, respectively). A larger blocking factor (blue group, BF=64) consistently outperforms smaller ones. This is primarily because larger blocks increase the ratio of arithmetic operations to global memory accesses (arithmetic intensity) and reduce the total number of kernel launches required.
- **Thread Coarsening:** Within each color group, the difference lies in the usage of thread coarsening. The "Coarse" variants (solid lines/markers) consistently outperform their "No-Coarse" counterparts (faded lines). For example, `SharedMem-BF32-NoCoarse` processes 1024 elements using 1024 threads, whereas `SharedMem-BF32-Coarse` processes 4096 elements with the same number of threads. By assigning 4 elements per thread, we not only increase the work per thread but also expose more instruction-level parallelism (ILP) and reduce the relative indexing overhead.

Consequently, the combination of Shared Memory, the largest feasible Blocking Factor (64), and Thread Coarsening (`SharedMem-BF64-Coarse`, represented by the blue line) yields the optimal configuration.

3.3 Micro Optimization Strategies

Experimental Setup and Methodology To further refine our implementation, we explored several micro-optimization techniques, including loop unrolling, register tiling, pinned memory, shared memory padding, and CUDA streams. We measured the total execution time across the performance testcases, consistent with the methodology in Section 3.2.

- **Loop Unrolling:** Since the update of each element within a block is independent (as discussed in Section 1.1), the inner loops are candidates for unrolling. However, overly aggressive unrolling increases register pressure per thread, potentially causing register spilling and degrading performance.
- **Register Tiling:** Instead of operating directly on shared memory for every calculation, we explicitly load frequently accessed values into registers. This technique aims to reduce shared memory traffic and latency.
- **Shared Memory Padding:** To mitigate potential shared memory bank conflicts, we pad the column dimension of our 2D shared memory arrays by one element (e.g., `[BLOCK][BLOCK+1]`).
- **Pinned Memory:** We utilize page-locked (pinned) host memory to accelerate data transfer between the host and the device.
- **Streaming:** In Phase 2, the row and column block updates are independent and can be executed simultaneously. We assign them to separate CUDA streams (`stream_row` and `stream_col`) to enable overlap, while Phases 1 and 3 use the default `stream_main`.

Implementation Details For Phase 3, we implemented register tiling and unrolling as follows:

```
// Register Tiling: Load data into local registers
int reg_self[2][2];
reg_self[0][0] = d_D[self_start + ty * V_padded + tx];
reg_self[0][1] = d_D[self_start + ty * V_padded + (tx + HALF_BLOCK)];
reg_self[1][0] = d_D[self_start + (ty + HALF_BLOCK) * V_padded + tx];
reg_self[1][1] = d_D[self_start + (ty + HALF_BLOCK) * V_padded + (tx +
↪  HALF_BLOCK)];

#pragma unroll 32
for (int k = 0; k < BLOCKING_FACTOR; ++k) {
    const int r0 = sm_row[ty][k];
    const int r1 = sm_row[ty + HALF_BLOCK][k];
    const int c0 = sm_col[k][tx];
    const int c1 = sm_col[k][tx + HALF_BLOCK];

    reg_self[0][0] = min(reg_self[0][0], r0 + c0);
    reg_self[0][1] = min(reg_self[0][1], r0 + c1);
    reg_self[1][0] = min(reg_self[1][0], r1 + c0);
    reg_self[1][1] = min(reg_self[1][1], r1 + c1);
}
```

We use `#pragma unroll 32` to control the unroll factor. Shared memory padding is declared as:

```
__shared__ int sm_row[BLOCKING_FACTOR][BLOCKING_FACTOR + 1];
__shared__ int sm_col[BLOCKING_FACTOR][BLOCKING_FACTOR + 1];
```

Pinned memory allocation and stream creation are handled via:

```
cudaHostAlloc(&D, V_padded * V_padded * sizeof(int), cudaHostAllocDefault);
// ...
cudaStreamCreate(&stream_main);
cudaStreamCreate(&stream_row);
cudaStreamCreate(&stream_col);
```

Results and Analysis **Loop Unrolling:** Figure 3 illustrates the impact of loop unrolling. Unrolling consistently yields performance benefits over no unrolling. Interestingly, the default compiler behavior produces results similar to full unrolling (factor 64). However, limiting the unroll factor to 32 achieves the best performance. We conjecture that full unrolling (64) consumes excessive registers, leading to register spilling to local memory, which offsets the benefits of reduced loop overhead.

Register Tiling: Figure 4 shows the performance breakdown by testcase for register tiling. The technique consistently outperforms the baseline, though the margin is minor. This suggests that the `nvcc` compiler's optimization passes are already effective at promoting frequently used shared memory operands to registers, leaving limited room for manual improvement.

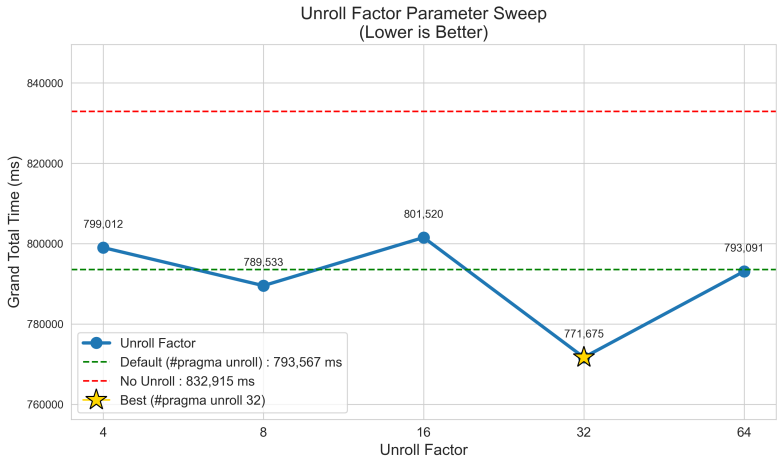

Figure 3: Impact of Loop Unrolling Factors

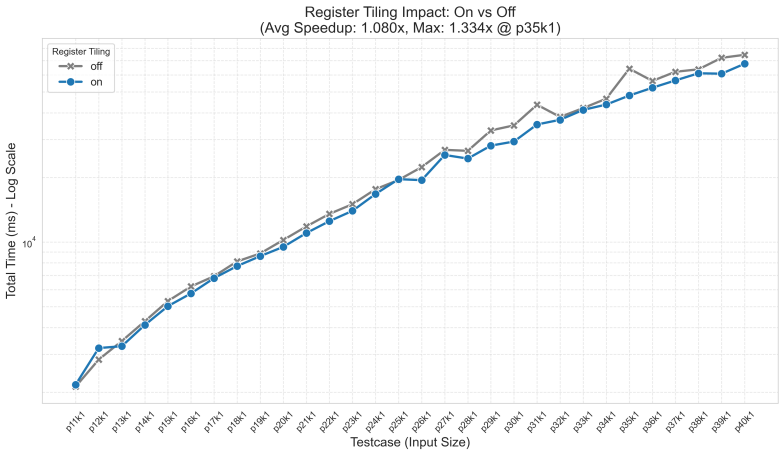

Figure 4: Performance Impact of Register Tiling

Ablation Study (Padding, Pinning, Streaming): The combined effects of other optimizations are shown in Figure 5.

- **Padding:** Surprisingly, shared memory padding degraded performance. Consequently, we disabled it in the final submission.
- **Streaming:** Utilizing multiple streams provided a slight speedup. However, since Phase 3 dominates the execution time, optimizing the overlap of the two relatively small kernels in Phase 2 yields negligible overall gains.
- **Pinned Memory:** While pinned memory showed slight improvements in our micro-benchmarks, its impact was inconsistent when running the full judging script (`hw3-2-judge`). Nevertheless, we retained it in the final version as best practice.

In summary, while register tiling offers consistent but minor gains, and loop unrolling requires careful tuning to avoid register pressure, techniques like padding can sometimes be counter-productive depending on the specific access patterns and hardware architecture.

3.4 Weak Scalability of Multi-GPU Implementation

Experimental Setup and Methodology This experiment evaluates the weak scalability of our multi-GPU implementation. Weak scaling measures the ability of a system to maintain constant

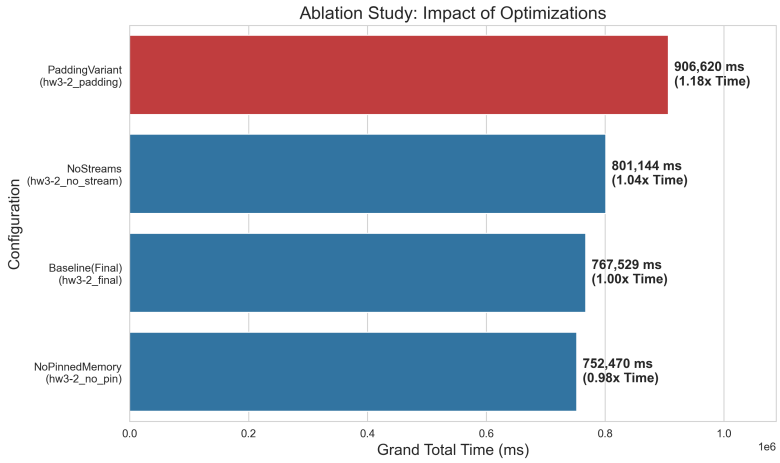

Figure 5: Ablation Study: Impact of Padding, Pinned Memory, and Streaming

execution time as the workload increases in proportion to the available computational resources. Formally, let T_1 denote the time to process a workload W on a single GPU. In an ideal weak scaling scenario, utilizing N GPUs to solve a scaled workload of $W \times N$ results in an execution time $T_N \approx T_1$. The weak scaling efficiency is defined as:

$$E = \frac{T_1}{T_N}$$

Given that the computational complexity of the Floyd-Warshall algorithm is $\mathcal{O}(V^3)$, doubling the vertex count V increases the computational workload by $8\times$. To construct a valid weak scaling test for $N = 2$ GPUs (i.e., doubling the workload), we selected pairs of testcases satisfying the condition:

$$2 \times V_{\text{single}}^3 \approx V_{\text{multi}}^3 \implies V_{\text{multi}} \approx \sqrt[3]{2} \times V_{\text{single}} \approx 1.26 V_{\text{single}}$$

Based on this derivation, we identified appropriate testcase pairs from our dataset (e.g., pairing $V = 11000$ with $V = 13851$) to verify whether the dual-GPU setup could maintain the execution time under double the computational load.

Results and Analysis The weak scaling results are presented in Figure 6. The observed efficiency across all testcase pairs hovers between **0.5** and **0.6**. This indicates that when the workload and resources were both doubled, the execution time also approximately doubled ($T_{\text{multi}} \approx 2 \times T_{\text{single}}$), rather than remaining constant.

Bottleneck Analysis: The deviation from ideal scaling ($E = 1.0$) is primarily attributed to the significant communication overhead inherent in the distributed Floyd-Warshall algorithm:

1. **PCIe Bandwidth Saturation:** Unlike the single-GPU implementation where all memory accesses remain within the high-bandwidth memory, the multi-GPU implementation requires frequent data exchange (via Peer-to-Peer or Host memory) to synchronize the pivot rows and blocks at every iteration k.

2. **Synchronization Latency:** The iterative nature of the algorithm imposes global synchronization barriers between GPUs. As the problem size V scales, both the volume of data transfer and the frequency of synchronization increase, causing the communication cost to outweigh the computational throughput gains provided by the second GPU.

In conclusion, while the multi-GPU approach enables solving larger problem sizes that exceed the memory capacity of a single device, its scalability is strictly bounded by the interconnect bandwidth (PCIe) rather than compute capability.

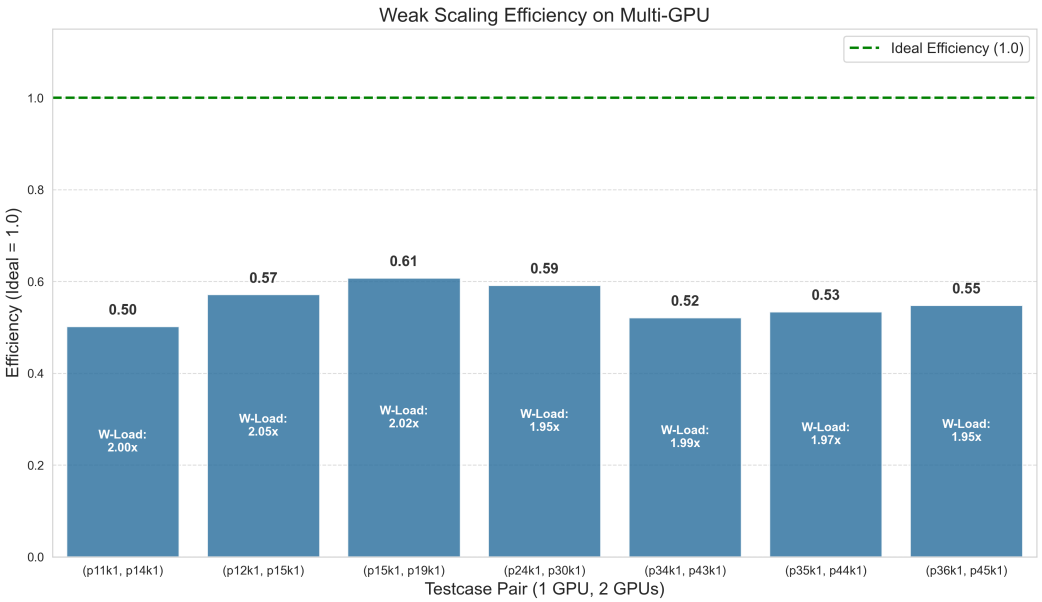

Figure 6: Weak Scalability Efficiency. An efficiency of 1.0 represents ideal scaling (constant time). The observed efficiency of ~ 0.5 highlights the significant impact of inter-GPU communication overhead.

3.5 Time Distribution Analysis

Experimental Setup and Methodology This experiment breaks down the total execution time into its constituent components to identify performance bottlenecks across different problem sizes. We categorize the runtime into three distinct phases:

- **I/O Time:** Includes file operations (reading input graphs, writing results) and host memory management (e.g., padding, pinned memory allocation).
- **Communication Time:** Measures the data transfer latency between Host and Device over the PCIe bus, encompassing both the initial H2D copy and the final D2H retrieval.
- **Compute Time:** Represents the aggregate execution duration of all GPU kernels (Phase 1, 2, and 3) across all iterations, measured via high-precision CUDA Events.

Results and Analysis The results are visualized in Figures 7, 8, and 9.

- **Overall Scalability (Figure 7):** As expected from the algorithm's $\mathcal{O}(V^3)$ complexity, the compute time (blue line) scales cubically, which appears as a steep linear slope on the logarithmic plot. Communication and I/O times scale quadratically ($\mathcal{O}(V^2)$) and roughly linearly, respectively.
- **Time Breakdown (Figure 8):** For small input sizes (e.g., c01.1-c21.1), the overhead of I/O and system initialization dominates the total runtime. However, as the problem size increases (p11k1 onwards), computation becomes the overwhelming bottleneck, accounting for nearly 90% of the execution time. This shift confirms that our optimization efforts should prioritize kernel performance for large-scale problems.
- **Kernel Phase Distribution (Figure 9):** Within the compute time, a similar trend is observed. For small graphs, the fixed overhead of launching kernels for Phase 1 (Pivot) and Phase 2 (Panel) is significant. However, as V grows, the computational workload of Phase 3 (which updates $(V/B-1)^2$ blocks) dwarfs that of Phase 1 (1 block) and Phase 2 ($2(V/B-1)$ blocks). Consequently, Phase 3 becomes the dominant kernel, consuming over 99% of the GPU time for the largest testcases.

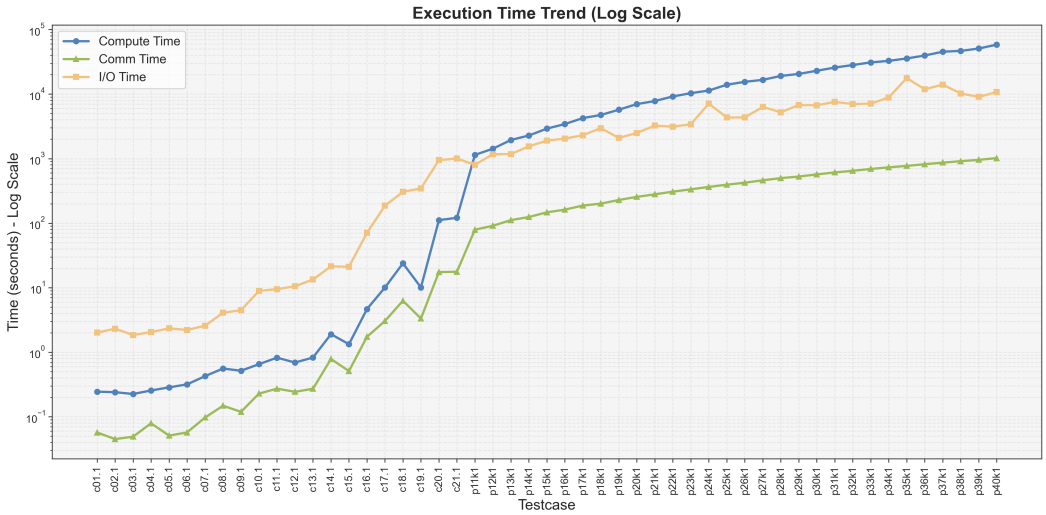

Figure 7: Execution Time Trend (Log Scale). Compute time grows much faster than I/O or communication, becoming dominant for large inputs.

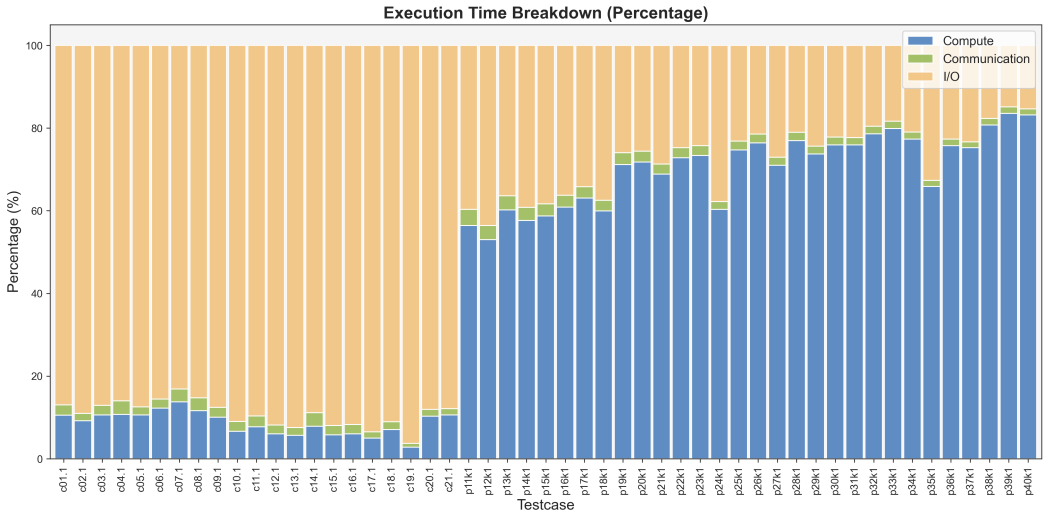

Figure 8: Percentage Breakdown of Total Execution Time. Small inputs are I/O bound, while large inputs are compute bound.

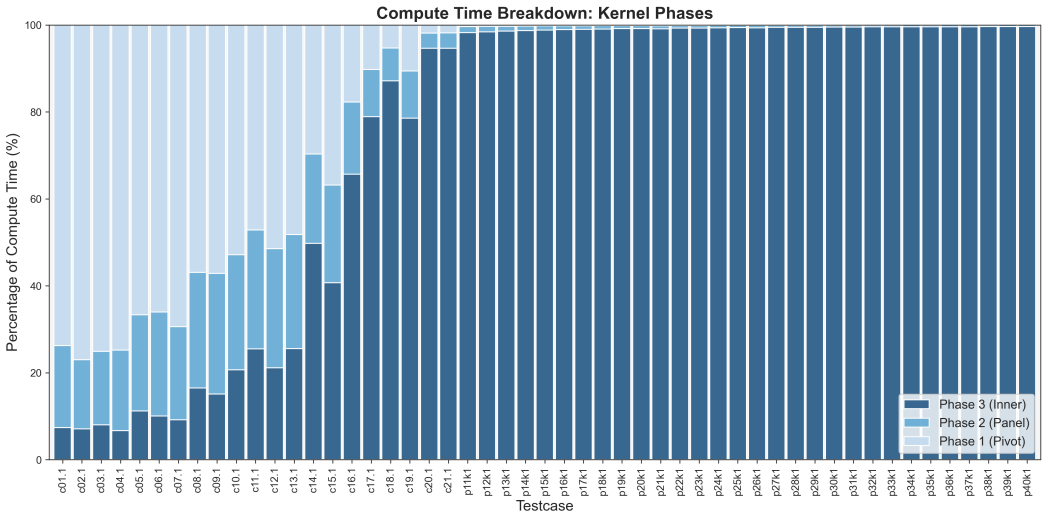

Figure 9: Breakdown of GPU Compute Time by Kernel Phase. Phase 3 dominates execution time as problem size increases.

4 AMD GPU Porting and Analysis

4.1 Implementation

To port our implementation from CUDA to the AMD ROCm platform (HIP), we initially performed a direct syntax translation by replacing all `cuda` prefixes with `hip`. This served as our baseline implementation. From this baseline, we introduced three specific optimizations tailored to the AMD architecture.

Launch Bounds Optimization First, we explicitly instructed the compiler to optimize register usage for a thread block size of 1024 using the `__launch_bounds__` qualifier. This hints to the compiler to limit register pressure to ensuring that a block of 1024 threads can theoretically reside on a Compute Unit.

```
__global__ void __launch_bounds__(1024) kernel_phase1(int *d_D, const int r, const
↪  int V_padded);
__global__ void __launch_bounds__(1024) kernel_phase2_row(int *d_D, const int r,
↪  const int V_padded);
__global__ void __launch_bounds__(1024) kernel_phase2_col(int *d_D, const int r,
↪  const int V_padded);
__global__ void __launch_bounds__(1024) kernel_phase3(int *d_D, const int r, const
↪  int V_padded);
```

Shared Memory Padding Second, we re-introduced padding in shared memory declarations to mitigate bank conflicts. Note that in our optimized CUDA version, we removed padding as it surprisingly degraded performance; however, for the AMD architecture, we adhered to standard optimization practices to ensure conflict-free access patterns.

```
__shared__ int sm_row[BLOCKING_FACTOR][BLOCKING_FACTOR + 1];
__shared__ int sm_col[BLOCKING_FACTOR][BLOCKING_FACTOR + 1];
```

Explicit Vectorized Memory Access (int4) Finally, to better leverage the AMD architecture, we explicitly utilized 128-bit vectorized memory accesses using the `int4` data type for loading data from Global Memory to Shared Memory (LDS) and storing it back. The fundamental execution unit on an AMD GPU is a *wavefront* consisting of 64 threads, compared to the 32-thread *warp* on NVIDIA GPUs. While both architectures benefit from wide, contiguous memory accesses, we observed that this manual vectorization had a much more significant impact on AMD than on NVIDIA.

```
// int4 Vectorized Load
__device__ __forceinline__ void load_block_int4(const int *__restrict__ src, const
↪  int stride, int dst[BLOCKING_FACTOR][BLOCKING_FACTOR + 1]) {
    const int tid = threadIdx.y * HALF_BLOCK + threadIdx.x;
    // Calculate iterations required per thread
    const int iterations = (BLOCKING_FACTOR * BLOCKING_FACTOR) / (HALF_BLOCK *
    ↪  HALF_BLOCK * 4);

    #pragma unroll
    for (int i = 0; i < iterations; ++i) {
        const int offset = tid + i * (HALF_BLOCK * HALF_BLOCK);
        const int r = offset / (BLOCKING_FACTOR / 4);
        const int c = (offset % (BLOCKING_FACTOR / 4)) * 4;

        // Explicit 128-bit load
        int4 tmp = *((int4 *)(src + r * stride + c));
        dst[r][c]     = tmp.x;
        dst[r][c + 1] = tmp.y;
        dst[r][c + 2] = tmp.z;
        dst[r][c + 3] = tmp.w;
    }
}

// int4 Vectorized Store
__device__ __forceinline__ void store_block_int4(int *__restrict__ dst, const int
↪  stride, int src[BLOCKING_FACTOR][BLOCKING_FACTOR + 1]) {
    // ... (Similar logic for store) ...
        int4 tmp;
        tmp.x = src[r][c];
        tmp.y = src[r][c + 1];
        tmp.z = src[r][c + 2];
        tmp.w = src[r][c + 3];
        *((int4 *)(dst + r * stride + c)) = tmp;
    // ...
}
```

Analysis of Compiler Behavior (nvcc vs. hipcc): We hypothesize that on NVIDIA platforms, the `nvcc` compiler is highly aggressive at automatically coalescing memory accesses. When it detects that 32 threads in a warp are reading from adjacent memory locations, it implicitly combines these into optimal memory transactions without programmer intervention. Consequently, manual `int4` vectorization often yields negligible gains on CUDA. In contrast, on AMD, the ROCm compiler (`hipcc`) appears less aggressive in automatically inferring these vector loads from scalar code. By explicitly casting pointers to `int4*` and performing single 128-bit reads, we force the generation of vector load instructions. This should reduce the total number of memory instructions issued, effectively saturating the memory bus and ensuring that the wider 64-thread wavefronts utilize the memory bandwidth efficiently.

4.2 Experiment and Analysis

4.2.1 Blocking Factor Analysis on AMD GPU

In this experiment, we investigate the impact of blocking factors (BF) on the performance of the Blocked Floyd-Warshall algorithm on the AMD MI100 GPU. We utilize `rocprof-compute` to collect detailed hardware metrics using testcase `c21.1` ($V = 5000$). The theoretical operation count for Integer GOPS calculation is defined as $2 \times V^3 = 250 \times 10^9$ operations. To characterize memory behavior, we extract the following metrics from the `rocprof-compute` analysis report:

- **Shared Memory**: Derived from Metric 2.1.16 (*Theoretical LDS Bandwidth*).
- **Global Memory Load**: Derived from Metric 2.1.22 (*L2-Fabric Read BW*).
- **Global Memory Store**: Derived from Metric 2.1.23 (*L2-Fabric Write BW*).

The specific profiling commands used are listed in the code snippet below.

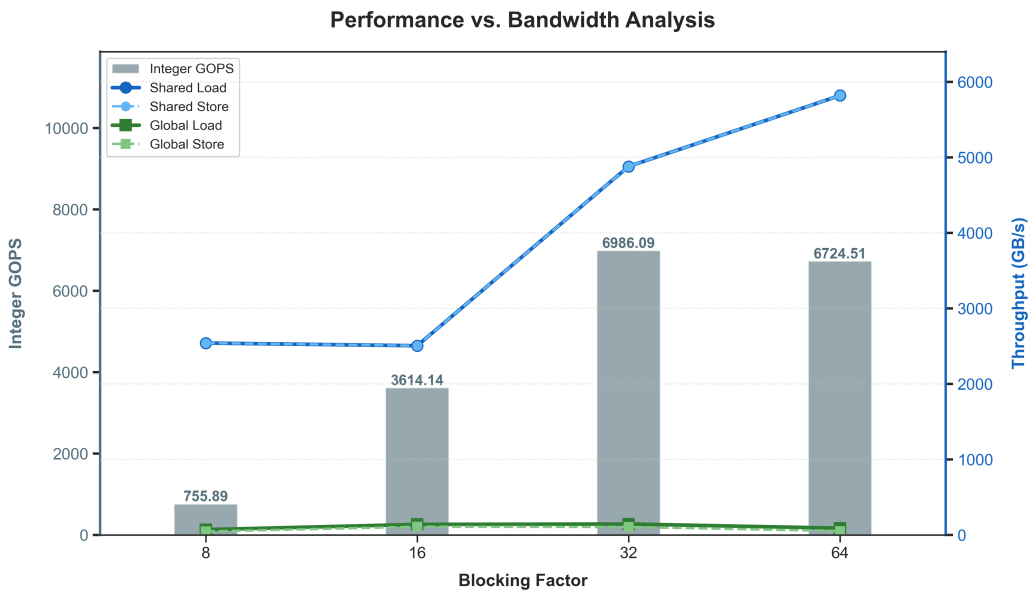

Figure 10: Performance and Bandwidth Analysis with Different Blocking Factors on AMD MI100. BF=32 achieves the best balance between shared memory usage and global memory efficiency.

```
source /home/pp25/share/venv/bin/activate
# === BF = 8 ===
srun -p amd -N1 -n1 --gres=gpu:1 rocprof-compute profile -n bf8_run -- ./hw3-2_bf8
↪  ./testcases-amd/c21.1 temp.out
srun -p amd -N1 -n1 --gres=gpu:1 rocprof-compute analyze -p workloads/bf8_run/MI100/
```

The experimental results, including execution time, GOPS, and memory throughput, are visualized in Figure 10. The total execution times for BF 8, 16, 32, and 64 are 330.73 ms, 69.17 ms, 35.79 ms, and 37.18 ms, respectively. As shown in Figure 10, a blocking factor of 32 achieves the highest performance ($\approx$ 6986 GOPS). While BF 64 exhibits significantly higher shared memory throughput ($\approx$ 5800 GB/s vs. 4800 GB/s for BF 32), it yields slightly lower GOPS ($\approx$ 6724) and reduced global memory bandwidth. This suggests that while larger blocks increase data reuse in shared memory, they may lead to reduced occupancy or memory level parallelism (MLP), making BF 32 the optimal sweet spot for this workload.

4.2.2 Equivalent Code Comparison: Manual Vectorization

We conducted a cross-platform experiment utilizing two distinct implementations to evaluate the impact of manual memory vectorization optimization:

- **Group 0 (CUDA-Optimized Baseline):** The baseline implementation, heavily optimized for CUDA semantics (e.g., implicit coalescing). The AMD version is a direct syntax translation (replacing `cuda` with `hip`).
- **Group 1 (Explicit Vectorization):** An implementation featuring manual `int4` (128-bit) vectorization to enforce wide memory transactions. The goal was to test if explicit vector instructions are necessary to saturate the memory bandwidth on either architecture.

We utilized `rocprof-compute` (on AMD MI100) and `nvprof` (on NVIDIA GTX 1080) to gather precise hardware metrics for the compute-bound Phase 3 kernel. The profiling commands used are listed below:

```
# AMD Profiling
srun -p amd -N1 -n1 --gres=gpu:1 rocprof-compute profile -n group0_hip \
    -- ./hw3-2-group0-hip ./testcases-amd/p11k1 temp.out
```

```
srun -p amd -N1 -n1 --gres=gpu:1 rocprof-compute analyze -p
↪  workloads/group0_hip/MI100/

# NVIDIA Profiling
srun -p nvidia -N1 -n1 --gres=gpu:1 nvprof ./hw3-2-group0-cuda ./testcases/p11k1
↪  temp.out
srun -p nvidia -N1 -n1 --gres=gpu:1 nvprof --metrics \
achieved_occupancy,sm_efficiency,gld_throughput,\
gst_throughput,shared_load_throughput,shared_store_throughput \
./hw3-2-group0-cuda ./testcases/p11k1 temp.out
```

Analysis of AMD Results (MI100) Contrary to the initial hypothesis that manual `int4` vectorization would improve throughput on the 64-thread wavefront architecture, the results (Table 5) show a performance regression.

- **Performance Regression:** The Group 1 kernel execution time increased from **369.19 ms** to **418.57 ms** ($\approx 13\%$ slowdown).
- **Register Pressure & Occupancy:** The root cause is identifiable in the occupancy metrics. Wavefront Occupancy dropped from **21.68%** (Group 0) to **19.75%** (Group 1). The use of explicit vector types likely increased the register pressure (VGPR usage) per thread, forcing the scheduler to reduce the number of active wavefronts per Compute Unit (CU). This reduction in parallelism outweighed any potential gain from wider memory instructions.
- **Compiler Efficacy:** The fact that Group 0 achieved similar or better memory throughput suggests that the `hipcc` compiler is already capable of automatically coalescing adjacent scalar loads into optimal transactions, rendering manual vectorization redundant and potentially harmful due to increased resource overhead.

Analysis of NVIDIA Results (GTX 1080) On the NVIDIA platform, the manual vectorization also failed to yield performance improvements, confirming the robustness of the baseline optimization.

- **Throughput Stability:** Global memory throughput decreased slightly from 277.98 GB/s to 236.55 GB/s. This reconfirms that the `nvcc` compiler and NVIDIA's hardware memory coalescing units effectively handle scalar loads without manual intervention.
- **Saturation:** The baseline Group 0 implementation already achieves near-saturation performance (86.78% global memory utilization and >99% SM activity), leaving little room for improvement via micro-optimizations like vectorization.

Cross-Platform Hardware Utilization Observation A striking observation from Tables 4 and 5 is the vast difference in hardware utilization between the consumer-grade GTX 1080 and the data-center MI100.

- **GTX 1080 (Squeezed to Limit):** We achieved exceptionally high utilization metrics: **91.15% Occupancy** and **86.78% Memory Bandwidth Utilization**. This indicates our code effectively "squeezed" the Pascal architecture to its limits.
- **MI100 (Underutilized Potential):** Conversely, on the powerful MI100, we utilized only **18.51%** of the available 1.2 TB/s global bandwidth and achieved merely **21.68% Occupancy**. While the raw execution time on MI100 (369 ms) is much faster than GTX 1080 (1187 ms) due to superior hardware specs, the low utilization percentages highlight that straightforward porting is insufficient. Unlocking the massive potential of CDNA architectures requires deeper, architecture-specific optimizations (e.g., maximizing occupancy to hide latency, using matrix cores) rather than simple vectorization tweaks.

4.2.3 Weak Scalability of Multi-GPU Implementation

We selected testcases ranging from `p11k1` to `p25k1` to evaluate the weak scalability of our implementation on the AMD platform. The results are presented in Figure 11. Experimental results indicate that our AMD implementation achieves a higher weak scaling efficiency of approximately **0.8**, compared to the CUDA version. We conjecture that this phenomenon is due to the single-GPU baseline on AMD being less aggressively optimized than its CUDA counterpart. Since the single-device computation

Table 4: Profiling Metrics Comparison of Group 0 at `p11k1`: NVIDIA GTX 1080 vs. AMD MI100 Platform. Values are formatted as **NVIDIA / AMD**. Parentheses indicate specific metric names used in `nvprof` and `rocprof-compute`.

Metric Category	Value (Avg)	Theoretical Peak	Utilization (%)
Kernel 3 Execution Time		1187.07 ms / 369.19 ms	
Occupancy (*nv*: Achieved Occupancy) (*amd*: Wavefront Occupancy)	–	100%	91.15% / 21.68%
Global Mem Throughput (*nv*: Global Load+Store Throughput) (*amd*: L2-Fabric Read+Write BW)	277.98 / 227.46 GB/s	320.32 / 1228.80 GB/s	86.78% / 18.51%
Shared Mem Throughput (*nv*: Shared Memory Load+Store Throughput) (*amd*: Theoretical LDS Bandwidth)	3474.79 / 6402.98 GB/s	4439.04 / 23070.72 GB/s	78.28% / 27.75%
SM/CU Activity (*nv*: Multiprocessor Activity) (*amd*: Active CUs)	–	100%	99.89% / 49.17%

Table 5: Profiling Metrics Comparison of Group 1 at `p11k1`: NVIDIA GTX 1080 vs. AMD MI100 Platform. Values are formatted as **NVIDIA / AMD**. Parentheses indicate specific metric names used in `nvprof` and `rocprof-compute`.

Metric Category	Value (Avg)	Theoretical Peak	Utilization (%)
Kernel 3 Execution Time		1195.82 ms / 418.57 ms	
Occupancy (*nv*: Achieved Occupancy) (*amd*: Wavefront Occupancy)	–	100%	93.89% / 19.75%
Global Mem Throughput (*nv*: Global Load+Store Throughput) (*amd*: L2-Fabric Read+Write BW)	236.55 / 227.46 GB/s	320.32 / 1228.80 GB/s	73.85% / 21.32%
Shared Mem Throughput (*nv*: Shared Memory Load+Store Throughput) (*amd*: Theoretical LDS Bandwidth)	4021.45 / 6402.98 GB/s	4439.04 / 23070.72 GB/s	90.59% / 25.39%
SM/CU Activity (*nv*: Multiprocessor Activity) (*amd*: Active CUs)	–	100%	99.91% / 49.17%

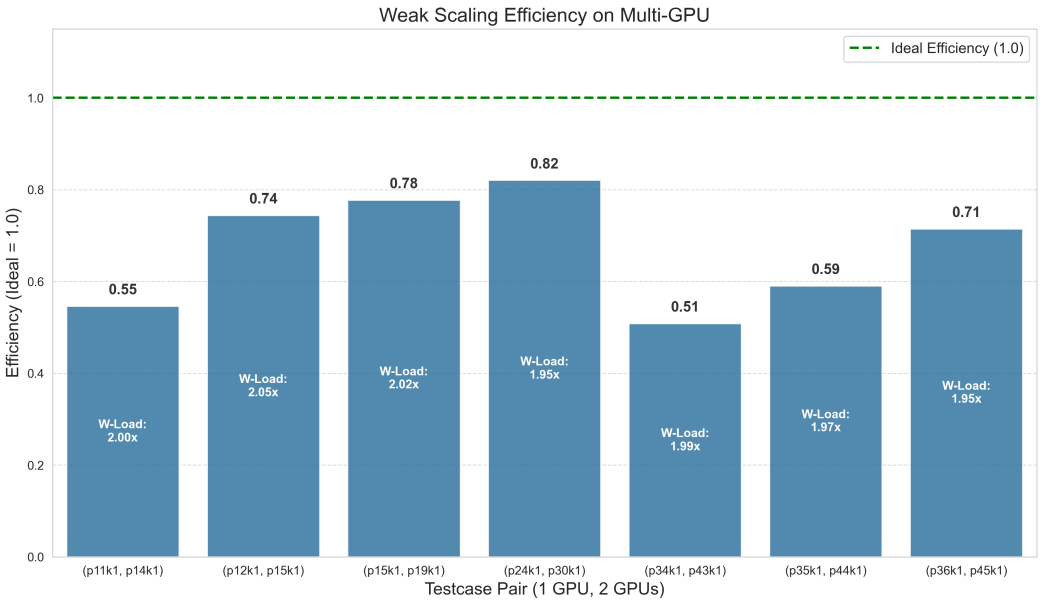

Figure 11: Weak Scalability Efficiency on AMD Multi-GPU. The higher efficiency (~ 0.8) relative to CUDA suggests a larger computation-to-communication ratio.

time is relatively higher, the fixed communication overhead constitutes a smaller fraction of the total execution time, thereby yielding a higher apparent scaling efficiency when additional GPUs are utilized.

5 Experience & Conclusion

This assignment provided a comprehensive deep dive into heterogeneous parallel computing, spanning CPU (OpenMP), Single-GPU, and Multi-GPU architectures using both NVIDIA CUDA and AMD HIP. Through a rigorous process of implementation, profiling, and iterative optimization, we developed a high-performance Blocked Floyd-Warshall solver. The core takeaway is the critical importance of memory hierarchy awareness: while algorithmic blocking (tiling) establishes the theoretical foundation for data reuse, it is the precise management of Shared Memory, coalesced access patterns, and register usage that determines the actual hardware utilization.

Key Learnings The most significant lesson was the architectural divergence between NVIDIA and AMD GPUs. While the high-level programming models (CUDA vs. HIP) are syntactically similar, their performance characteristics differ fundamentally. Discovering that manual `int4` vectorization was crucial for saturating the AMD memory bus—while being largely redundant on the NVIDIA GTX 1080 due to the aggressive `nvcc` compiler—highlighted the necessity of architecture-specific tuning. This reinforces the principle that "portable code" does not imply "performance-portable code."

Another key insight came from the limitations of our naive Multi-GPU implementation. By simply partitioning the workload into upper and lower halves without sophisticated pipeline optimization, we observed that the performance scaling was strictly bounded. This practical experience demonstrated that adding more compute resources does not automatically yield linear speedups; without careful design to hide communication latency (e.g., via computation-communication overlap), the interconnect bandwidth (PCIe) quickly becomes the hard bottleneck.

Difficulties Encountered The primary difficulty lay in the intricate coordinate mapping required for the Blocked Floyd-Warshall algorithm. Correctly translating the 3D logic of the algorithm (Phase 1, 2, 3) into 2D GPU grid coordinates (`blockIdx`, `threadIdx`) while handling edge cases (padding) and avoiding bank conflicts was mentally demanding and error-prone. Debugging race conditions

within shared memory and verifying the correctness of inter-block dependencies required careful logical deduction.

Furthermore, navigating the AMD ROCm ecosystem presented a steep learning curve. Compared to the mature NVIDIA Nsight systems or `nvprof`, tools like `rocprof` and `rocprof-compute` required more manual configuration to extract meaningful hardware metrics (e.g., mapping low-level counters to high-level concepts like "Global Memory Throughput").

Feedback This assignment was an excellent exercise in modern accelerator programming. To further enhance the learning experience, we strongly suggest that the teaching staff provide a set of optimized reference implementations (for both CUDA and HIP) after the submission deadline. Access to "gold-standard" code would allow students to benchmark their own memory access patterns, understand the gap between their solutions and the theoretical peak, and discover advanced optimization techniques (e.g., warp-shuffle primitives or assembly-level tuning) that they might have missed.