Improving throughput by maximising resource overlap

IPACO Project Report
Group-6

Anirudh Saikrishnan - CS22BTECH11004
Kaipa Venkata Tuhil - CS22BTECH11030
Muskan Jaiswal - CS21BTECH11037

May 2025

Abstract

Modern NVIDIA GPUs partition a Streaming Multiprocessor (SM) into four independent *SM sub-partitions* (SMSPs). While this change reduces both area and energy, it aggravates *resource imbalance*: each SMSP owns only a subset of INT32, FP32, FP64 and Tensor cores together with an independent warp scheduler. A warp executed on one SMSP typically exercises a *single* numeric datapath, leaving the others idle. This work studies whether *static* kernel-level techniques can overlap the utilisation of different datapaths and thereby improve throughput without hardware changes. We use the matrix- multiplication benchmarks (`2MM, 3MM, GEMM`) from PolyBench/GPU as an open playground. Two concrete ideas are explored:

1. **Inter-datapath overlap:** mixing FP32 and FP64 instructions inside the same thread-block.
2. **Intra-datapath pipelining:** hiding global-memory latency from FP32 computations via warp specialization.

The report documents our design space exploration, experimental evaluation on an RTX 3050 Ti Laptop GPU (`sm_80`).

Github Repo: Link to repo

Contents

1 Introduction

1.1 The resource-overlap problem

- **Partitioned SMs.** Since *Ampere*, every SM is split into four *SMSPs*. Each owns a private register file, warp scheduler and a small slice of INT32, FP32, FP64 and Tensor cores.
- **Inefficiency.** A CUDA warp issues one instruction at a time—e.g. an FP32 FMA—which occupies *only the matching functional units* inside the hosting SMSP, keeping the rest idle.
- **Goal.** Craft kernel schedules that *overlap* multiple datapaths either (i) within the same warp or (ii) across specialised warps inside the same thread-block, increasing instantaneous SMSP occupancy.

1.2 Warp specialisation

Warp Specialisation (WS) dedicates some warps to **producers** (global →shared copies, integer address arithmetic, . . .) and others to **consumers** (FP/Tensor math). Independent warp schedulers may then run producer and consumer warps concurrently, masking latency while occupying different functional units.

1.3 Opportunities explored

1. **FP32 & FP64 overlap.** Divide the work amongst warps in such a way that some of them work on FP32 cores and some work on FP64 cores. Maximise overlap in this fashion.
2. **Multistage pipeline.** Use pipelined code with warp specialization that fetches the memory required for next round of computations while current computations are running.

2 Base Code Analysis

2.1 2 MM (α `*` `A` `*` `B` `*` `C` + β `*` `D`)

What each kernel does

- **K1** – `tmp` = $\alpha \cdot$ `A` $\cdot$ `B`
  Tiled 32×32 SGEMM producing a temporary buffer.
- **K2** – `D` = $\beta \cdot$ `D` + `tmp` $\cdot$ `C`
  Second SGEMM accumulating into `D`.

Table 1: GPU "speed-of-light" metrics — 2 MM

Kernel	SM Throughput [%]	Duration [ms]	Memory Throughput [%]
K1 (tmp = α·A·B)	82.28	7.65	82.28
K2 (D = β·D + tmp·C)	81.80	7.71	81.80

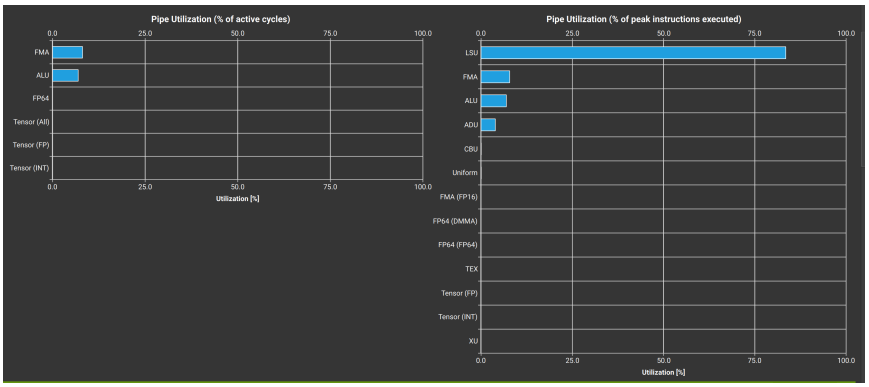

Figure 1: Compute work-load analysis – (Nearly identical for both kernels))

2.2 3 MM ((A·B)·(C·D))

Kernel roles

- **K1 – E = A·B**
- **K2 – F = C·D**
- **K3 – G = E·F**

Table 2: GPU "speed-of-light" metrics — 3 MM

Kernel	SM Throughput [%]	Duration [ms]	Memory Throughput [%]
K1 (E = A·B)	84.28	935.68	84.28
K2 (F = C·D)	84.25	933.76	84.25
K3 (G = E·F)	84.27	936.26	84.27

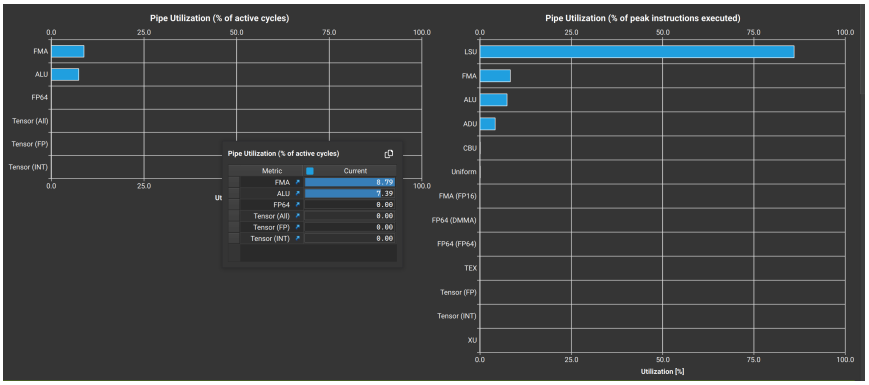

Figure 2: Compute work-load analysis – 3 MM K1 (Again, similar for all 3 kernels)

2.3 GEMM (C = $\alpha \cdot$ A $\cdot$ B + $\beta \cdot$ C)

Kernel Single tiled SGEMM with 32×32 blocks.

Table 3: GPU "speed-of-light" metrics — GEMM

Kernel	SM Throughput [%]	Duration [ms]	Memory Throughput [%]
GEMM_tiled	83.76	945.50	83.76

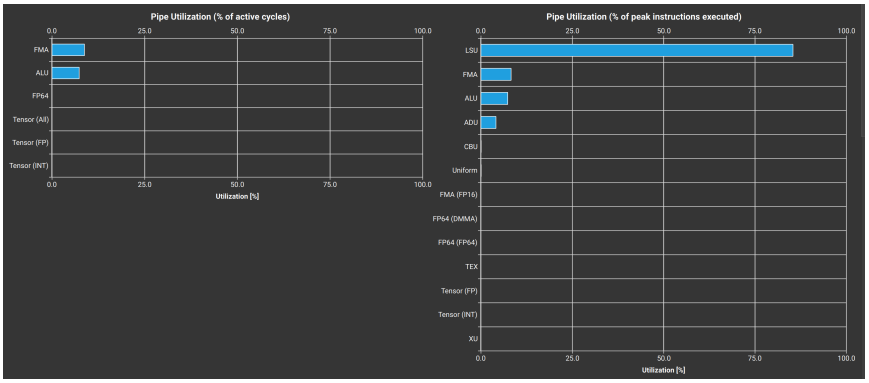

Figure 3: Compute work-load analysis – GEMM

2.4 General Remarks

The collected data looks extremely similar across all the kernels that were profiled. The reason for this is not surprising at all. Every kernel is performing one matrix multiplication operation along with some auxiliary operation which is much cheaper. This main matrix operation is similar for every kernel.

The main bottleneck for all the codes here is clear by the overwhelming utilization of the LSU units of the GPU. Each kernel is spending a majority of its time in just accessing data from the global memory and bringing it into the shared memory. As a result, the other compute units sit idle for most of the time, which is what gives such a lopsided result.

3 Methodology and Experiments

3.1 Overlapping FP32 and FP64 Compute Units

We explore two methods to try to overlap FP32 and FP64 execution on Ampere GPUs (where each SM has only 2 FP64 pipelines and FP64 TFLOP rate is 1/64 that of FP32[1]). Both methods failed to improve end-to-end throughput due to severe resource imbalance—FP64 units simply cannot keep up with the FP32-heavy workload.

3.1.1 Method A: In-kernel Warp-ID Scheduling

We modified each tiled kernel so that threads in warps whose warp-ID w satisfies

$$w \bmod P = r$$

would execute double-precision accumulates instead of single-precision, while the others remained FP32. For example:

```
// inside mm2_kernel1_tiled:
int warp_id = threadIdx.x / 32;
if ((warp_id % 2) == 0) {
  // perform FP32 multiply-add into float accumulators
} else {
  // perform FP64 multiply-add into double accumulators
}
```

Results:

- The warps that went down the FP32 path executed significantly faster than the warps that went down the FP64 path.
- The FP64 warps turned into the bottleneck which caused the performance to significantly worsen as compared to pure FP32 implementation.

[1] NVIDIA Ampere Architecture Whitepaper, https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf

3.1.2 Method B: Templated Two-Stream FP32+FP64 Pipelines

We then implemented separate templated kernels (one for `float`, one for `double`) and launched them concurrently on two CUDA streams. At runtime the scheduler interleaves warps:

```
// Stream 0: FP32 path
mm2_kernel1_tiled<float><<<grid,block,0,stream32>>>(...);

// Stream 1: FP64 path
mm2_kernel1_tiled<double><<<grid,block,0,stream64>>>(...);
```

Results:

- Similar to the previous method, the FP64 kernels ended up being the bottleneck
- The FP64 pipelines were almost 100% occupied while the other computational pipelines were less than 1% occupied
- Since we used templated kernels, we had double the number as the regular tiled version. The FP32 kernels here performed very similar to the regular tiled kernels while the FP64 kernels performed significantly worse

Main Takeaway: On Ampere GPUs (Like the ones we tested on), the FP64 cores exist only for calculations requiring extremely high precision and not for any sort of performance reasons. The fact that the number of FP32 cores is signficantly higher than the number of FP64 cores means that trying to overlap the two will only make the code run slower due to the unnecessary overhead added. //
However, on GPUs with more FP64 cores, this could certainly be a viable method to explore. Unfortunately, we were unable to test our codes on such GPUs and as such, cannot make any claims regarding them.

3.1.3 Method C: pipelining the memory fetches

We realized that trying to split the workload between fp32 and fp64 cores was not working, so we decided to try pipelining the memory fetches in the tiled kernels using warp specialization. The basic idea is that some warps only fetch the memory, while others only do computations on the fetched memory. This way, while some warps are doing computations on the current tile, the others are already fetching the next tile required. Here is the pseudocode for this method:

```
__shared__ DATA_TYPE Asub1[TILE_SIZE][TILE_SIZE];
__shared__ DATA_TYPE Bsub1[TILE_SIZE][TILE_SIZE];
__shared__ DATA_TYPE Asub2[TILE_SIZE][TILE_SIZE];
__shared__ DATA_TYPE Bsub2[TILE_SIZE][TILE_SIZE];
DATA_TYPE* Asubptr = &Asub1[0][0];
DATA_TYPE* Bsubptr = &Bsub1[0][0];
if (threadIdx.y < TILE_SIZE/2){
    // fetch the data for i= 2*threadIdx.y and i=2*threadIdx.y+1
}
DATA_TYPE* Asubptr2 = Asubptr;
DATA_TYPE* Bsubptr2 = Bsubptr;
Asubptr = &Asub2[0][0];
Bsubptr = &Bsub2[0][0];
__syncthreads();
for (int t = 0; t < ((nk + TILE_SIZE - 1) / TILE_SIZE)-1; t++)
{
    if (threadIdx.y < TILE_SIZE/2){
        // load t+1's data into Asubptr and Bsubptr, for
        // i = 2*threadIdx.y and 2*threadIdx.y+1
    } else {
        // perform ops on Asubptr2 and Bsubptr2 for
        // i = 2*(threadIdx.y-16) and 2*(threadIdx.y-16)+1
    }
    // Swap Asubptr and Asubptr2
    // Swap Bsubptr and Bsubptr2
    __syncthreads();
}
if (threadIdx.y >= TILE_SIZE/2){
    // perform ops on the last data in Asubptr2 and
    // Bsubptr2, for i = 2*(threadIdx.y-16) and 2*(threadIdx.y-16)+1

    // Store the final accumulated value in matrix C
}
```

Basicly, in a thread block(16x16 in this case), the first 8 warps are always fetching the tile for the next computation while the other 8 warps do the computations for the current tile. This way, there is an overlap between memory operations and computations, leading to better throughput. In previous tiled kernel that we used as the base code for comparing this, the memory operations and computations were done sequentially.

Results:

- All the three kernels ran roughly 50% faster
- The fp32 core utilization increased

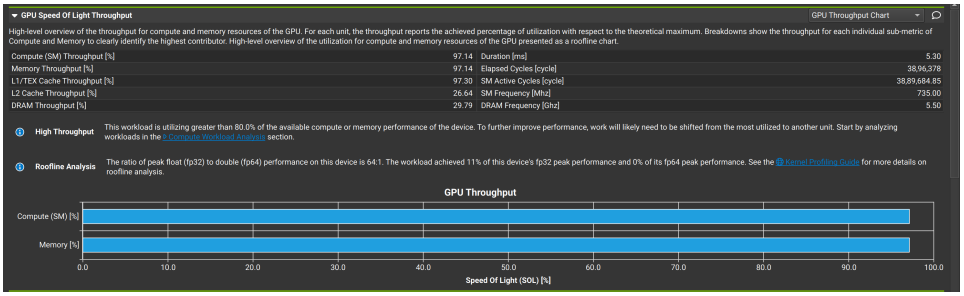

Figure 4: GPU "speed-of-light" metrics — 2 MM

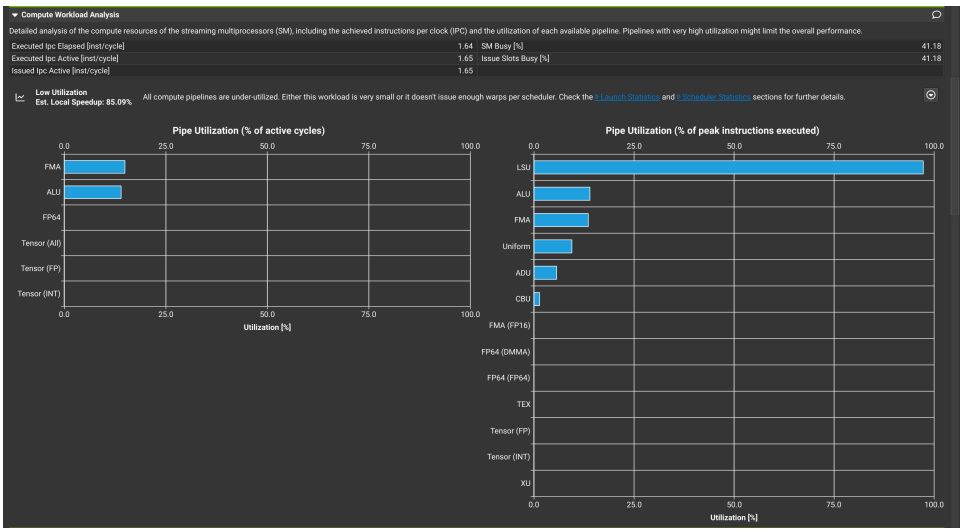

Figure 5: Compute workload analysis- 2 MM

Here are the metrics from nvidia nsight compute: **2MM: 3MM: GEMM:**

4 Final Code Analysis

Although both methods A and B successfully inserted FP64 instructions into the execution stream, the imbalance between abundant FP32 and scarce FP64 hardware makes overlap infeasible:

- **Hardware constraint:** Each Ampere SM contains only 2 FP64 pipelines versus dozens of FP32 pipelines.
- **Throughput mismatch:** The theoretical FP64 peak is only $\frac{1}{64}$ of

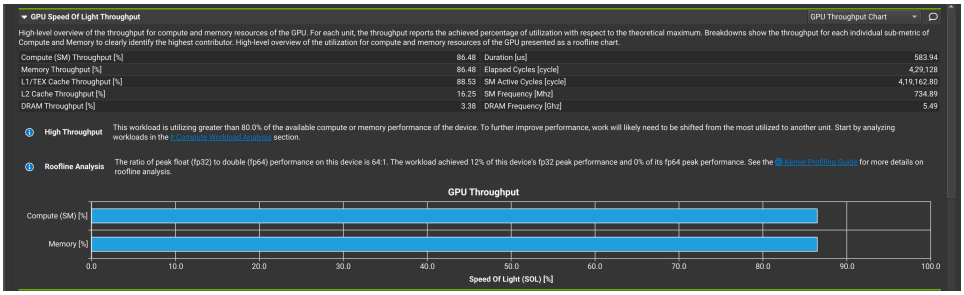

Figure 6: GPU "speed-of-light" metrics — 3 MM

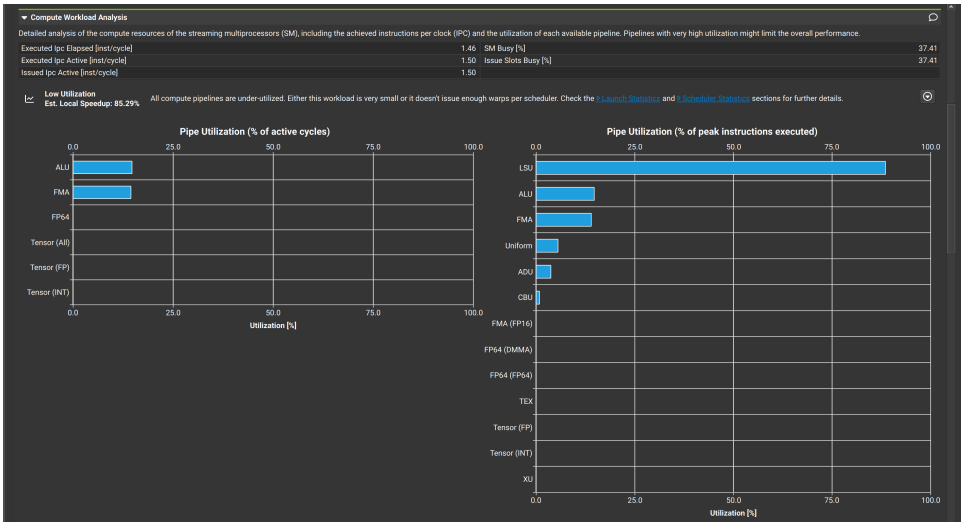

Figure 7: Compute workload analysis- 3 MM

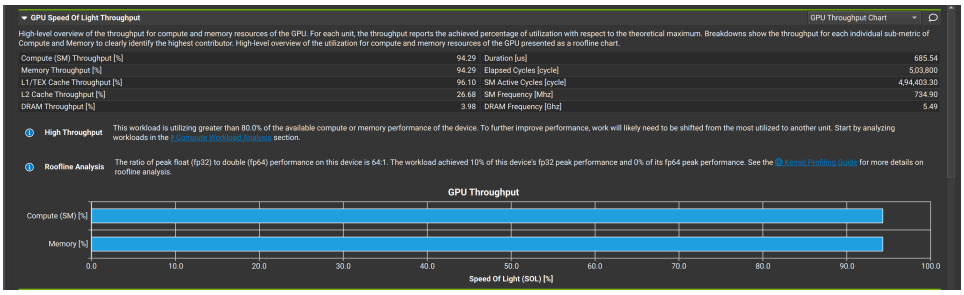

Figure 8: GPU "speed-of-light" metrics — GEMM

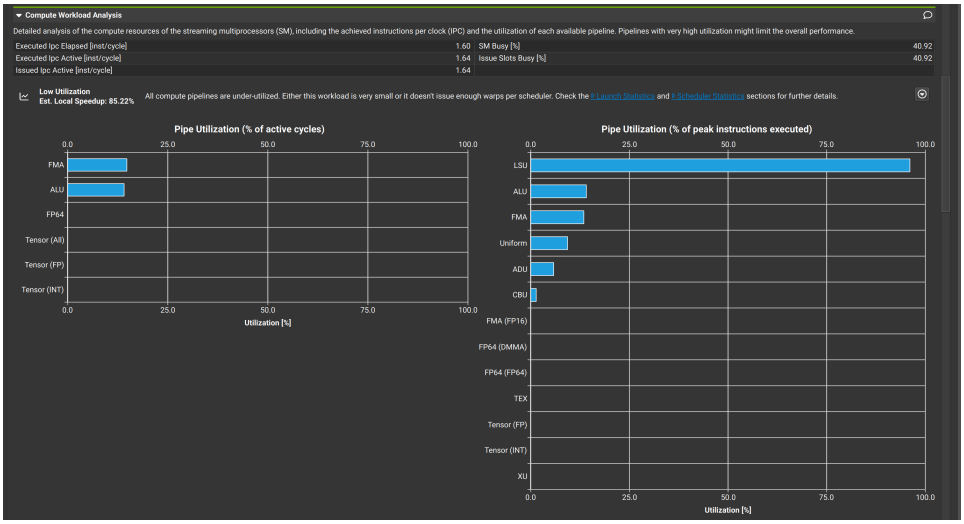

Figure 9: Compute workload analysis- GEMM

	2MM	**3MM**	**GEMM**
Baseline	0.01678	0.00280	0.5377
Tiled	0.00558	0.00111	0.1728
Pipelined with warp specialization	0.00361	0.00068	0.1164
Speedup over tiled	1.54	1.63	1.48

Table 4: Results for our pipelined kernel with warp specialization

FP32 peak, so even executing all SMs' FP64 warps continuously would saturate the FP64 units far below the FP32 rate.

- **Scheduler stalling:** Introducing FP64 warps delays the much more efficient FP32 warps, reducing overall SIMD utilization.

In summary, software-level warp dispatch or multi-stream overlap cannot overcome the fundamental hardware imbalance. Any FP64 work in a FP32-dominated kernel effectively becomes a performance bottleneck rather than a resource overlap opportunity. However, utilizing warp specialization to pipeline memory accesses provides major improvements over non-pipelined kernels.

5 Bibliography & References

[1] M. Bauer, H. Cook, and B. Khailany, "Cudadma: Optimizing GPU memory bandwidth via warp specialization," in *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC '11)*, New York, NY, USA, 2011, Association for Computing Machinery.

[2] K. Ho, H. Zhao, A. Jog, and S. Mohanty, "Improving GPU throughput through parallel execution using tensor cores and CUDA cores," in *2022 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2022, pp. 223–228.

[3] NVIDIA, "Nvidia Nsight Compute," NVIDIA Developer. Accessed April 5, 2025. Available: https://developer.nvidia.com/nsight-compute

[4] NVIDIA, "NVIDIA Ampere GA102 GPU Architecture Whitepaper," Version 2, 2020. Available: https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf

[5] Colfax International Research, "CUTLASS Tutorial: Design of a GEMM Kernel," 2021. Available: https://research.colfax-intl.com/cutlass-tutorial-design-of-a-gemm-kernel/