IPACO Project Report

Introduction | Optimising CUDA Code to Maximise GPU Resource Overlap and

The objective of this project is to identify and implement static optimisation techniques for GPU resource utilisation in matrix multiplication benchmarks from PolyBench/GPU. The focus is on improving performance by leveraging the overlap between different GPU resources without relying on hardware modifications.

Improve Throughput

The throughput can be increased by utilising different SMSP units (FP32, FP64, Tensor etc.) in parallel. In this project, we have tried multiple methods to parallelise unit usage and increase overall throughput.

Github Link: https://github.com/abhinavydv/IPACO-Project-GPU-Throughput

FP32 and FP64 Parallelism

In this section, we'll focus on parallelising the FP32 and FP64 units of CUDA SMSPs. Various experiments were carried out to find out how they can be parallelised, and then the method was tested with matrix multiplication for speed-up.

Experiments

ILP (Instruction Level Parallelism)

The cuda warp scheduler determines the processing unit to be used for the current instruction. If two adjacent instructions need different units for execution and are independent of each other then the scheduler runs them in parallel. This is called ILP.

Here are the details of the experiment:

1. **On NVIDIA RTX 3050 ti (Ampere architecture) (Laptop GPU)**
   The FP32 and FP64 performance has a very high disparity. The throughput ratio is 64:1, i.e., FP32 is 64 times faster than FP64. Even if the FP64 operations run in parallel, there is not much speed-up.

   The following code was used:

```
__global__ void test_speed(float *data){
   double a = 0;
   float b = 0.0;
   __shared__ float sdata[32];
   __shared__ double sdata2[32];
   if (threadIdx.x < 32){
         sdata[threadIdx.x] = 1.0f;
         sdata2[threadIdx.x] = 1.0;
   }
  __syncthreads();
   for (int i=0; i<10000; i++){
      a += sdata2[0];
      b += sdata[0];
      b += sdata[1];
      …
      b += sdata[31];
   }
   data[threadIdx.x] = a + b;
}
```

The above code does multiple FP32 and 1 FP64 multiplication.
The running time is shown in the below table:

No of FP32 / iter	No of FP64 / iter	Time (ms)
0	1	23.6
65	0	20.4
64	1	23.6
97	0	30.76
96	1	31.8

We can observe that 1 FP64 instruction takes 23.6 ms. 1 FP64 instruction along with 64 FP32 instructions also takes 23.6 ms. From this, we can conclude that they are running in parallel when adjacent.
To confirm, we also had a case where the FP64 computation was dependent on FP32 computations. This was done by removing ‘a += sdata2[0];’ at the top and adding ‘a+=b’ at the bottom inside the for loop. In this case it takes around 46 ms. Hence, we can conclude that only if the instructions are independent then they run in parallel.

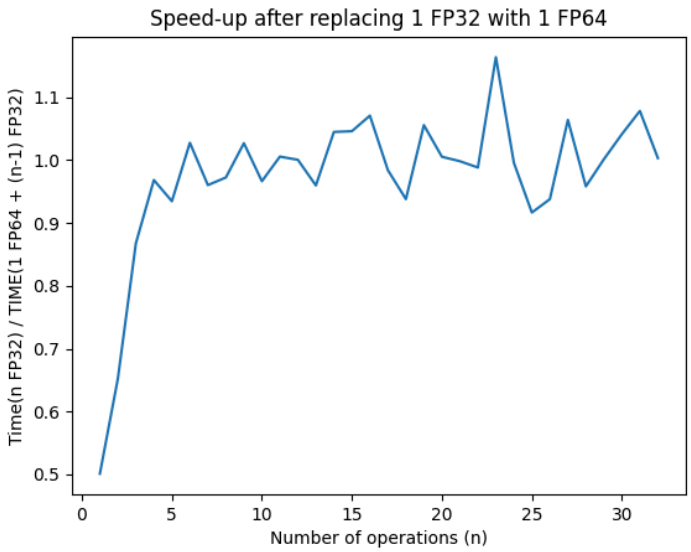
2. **On NVIDIA A100 (Ampere Architecture)**
   The throughput on the NVIDIA A100 is 2:1, i.e., FP32 units are 2 times faster than FP64 units. The following kernel was used to test their parallelism:

```
__global__ void test_speed(float *data){
    double a = 0;
    float b = 0.0;
    __shared__ float sdata[32];
    if (threadIdx.x < 32){
            sdata[threadIdx.x] = 1.0f;
    }
   __syncthreads();
    for (int i=0; i<1000000; i++){
        #ifdef FP32_ONLY
        for (int j = 0; j < NUM_OPS; j++){
            b *= sdata[(i+j)%32];
        }
        #else
        a *= sdata[i%32];
        for (int j = 0; j < NUM_OPS-1; j++){
            b *= sdata[(i+j)%32];
        }
        #endif
    }
    data[threadIdx.x] = a + b;
}
```

Here is a detailed explanation of the experiment using above kernel:
Two types of scenarios are compared:

a. n FP32 operations
b. 1 FP64 and (n-1) FP32 operations

The number of operations was a variable value (n=1 to 32). Here is a plot that shows the speed-up when 1 FP32 operation was replaced with 1 FP64 out of the total operations.



This experiment was performed to determine the number of FP32 operations per FP64 operation that can result in maximum speed-up.
As we can observe from the plot, the speed-up increases till n=7 and gets almost constant after that. So a good ratio of FP64 and FP32 operations can be 1:7.
The speed-up is close to 5%.

WARP Specialisation

This refers to the situation where different warps are assigned different tasks/roles.
In our case, different warps were given different units, i.e., some warps did FP32 operations and others did FP64 operations. The warp specialisation was also tested on both the GPUs:

1. **On NVIDIA RTX 3050 ti (Ampere architecture) (Laptop GPU)**
    a. Setup used
        i. 1 SMSP is given 8 warps
        ii. 1 warp does FP64 computations and 7 warps do FP32
    b. The time taken is more compared to 8 warps executing FP32.
    c. Time taken in different scenarios:
        i. No specialisation: 20.4 ms
        ii. With specialisation: 113.8 ms
        iii. 1 Warp per SMSP with only FP64: 112.6 ms

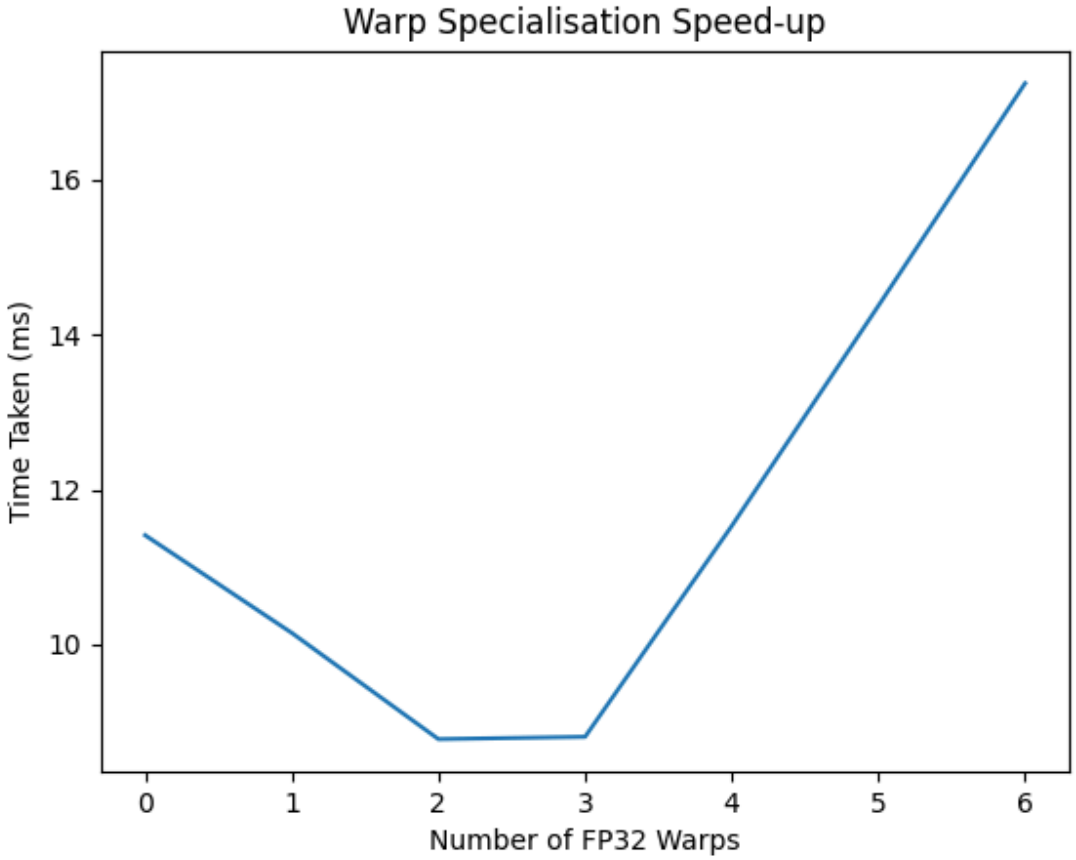
2. **On NVIDIA A100 (Ampere Architecture)**
    The kernel used in this case is:

```
__global__ void test_speed(float *data){
    double a = 1;
    float b = 1.0;
    __shared__ float sdata[32];
    if (threadIdx.x < 32){
            sdata[threadIdx.x] = 1.0f;
    }
    __syncthreads();
    if (threadIdx.y < NUM_FP64){
        for (int i=0; i<1000000; i++){
            a *= sdata[i%32];
        }
    } else {
        for (int i=0; i<1000000; i++){
            b *= sdata[i%32];
        }
    }
    data[threadIdx.x] = a + b;
}
```

First 4 warps do FP64 and others do FP32 operations. The work distribution on SMSPs looks like shown in the following figure:

FP32	FP32	FP32	FP32
FP32	FP32	FP32	FP32
FP32	FP32	FP32	FP32
FP32	FP32	FP32	FP32
FP32	FP32	FP32	FP32
FP32	FP32	FP32	FP32
FP32	FP32	FP32	FP32
FP64	FP64	FP64	FP64

Each SMSP gets 1 FP64 and 7 FP32 warps. So the FP32:FP64 operations ratio is 7:1. When continuous warps are of the same type, the above distribution is guaranteed due to strict Round Robin scheduling inside an SM.



The time taken in different scenarios is as follows:

a. No specialisation (only FP32): 11.45 ms
b. With specialisation: 8.81 ms

We can observe that warp specialisation reduces time required for the whole process. Hence warp specialisation is also a fair approach but the speed-up is 22.7 % which is better than ILP speed-up which is 5%.

Files

1. Experiments/figs: Folder containing the plot
2. Experiments/calc_time.py: Calculates time
3. Experiments/fp32_fp64_warp.cu: Experiment for warp specialisation
4. Experiments/fp32_fp64.cu: Experiment for ILP
5. Experiments/gen_plot_data.py: Generates data for plot
6. Experiments/plot_data.csv: The generated data for plot
7. Experiments/plot.py: The python code to plot the data

To run the codes:

1. ILP
    a. Compile Experiments/fp32_fp64.cu
    cd Experiments && mkdir -p bin
    nvcc fp32_fp64.cu -gencode arch=compute_80,code=sm_80 -D NUM_OPS=<number of operations> -o bin/fp32_fp64
    b. Run the compiled code
    ./bin/fp32_fp64
    It will print the time taken.
    c. To run the python code for plot
    python gen_plot_data.py
    python plot.py
2. Warp Specialisation
    a. Compile Experiments/fp32_fp64_warp.cu
    nvcc fp32_fp64.cu -gencode arch=compute_80,code=sm_80 -o bin/fp32_fp64_warp
    b. Run
    ./bin/fp32_fp64_warp
    It will print the time taken.

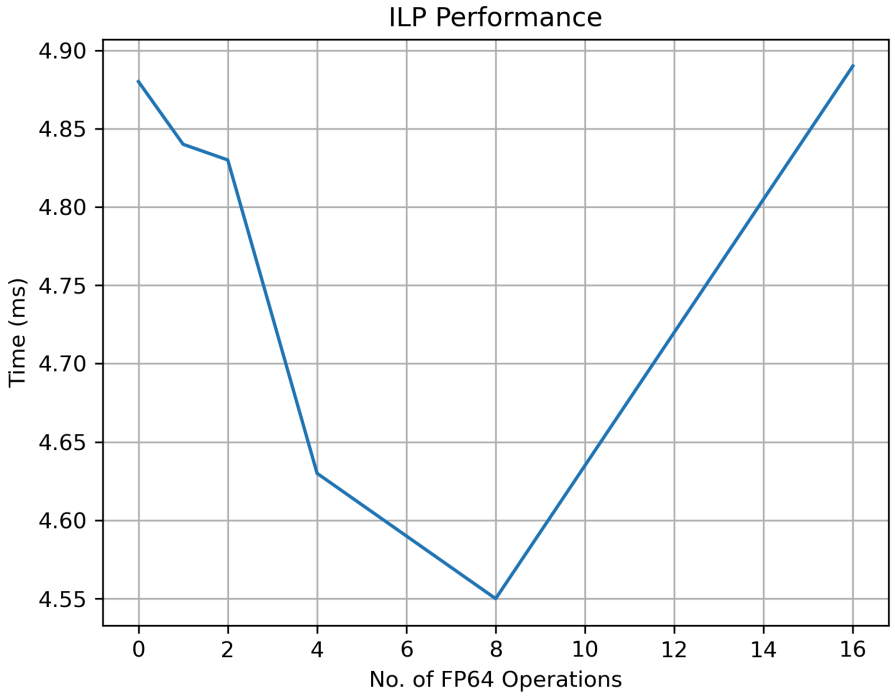
Matrix Multiplication Modifications

The above findings were then tried with matrix multiplication and the speed-up was recorded. Here is a detailed explanation:

ILP (Instruction Level Parallelism)

Each thread computes one point in the resultant matrix. For this it does 32 multiplication operations (Refer code to get a clarity on this part). Out of the 32 FP32 operations, some were

replaced by FP64 operations and the time taken was recorded. Here is the plot showing the results:



We can observe that there is a decrease in the time taken as no. of FP64 is increased to 8 but then it increases again when no of FP64 goes to 16. Hence 24:8 should be a good ratio of no of FP32 and no of FP64 for parallelism.
The performance increase is around 6.76%.

WARP Specialisation

Different warps do different types of multiplication, i.e., FP32 and FP64. In each SMSP, there can be 8 warps. Out of these, some are used for FP32 and others for FP64.
Here is the code snippet used:

```
if (threadIdx.y < 4){
    for (int j=0; j<WARP_SIZE; j++){
        mult(j, tmp2)   // FP64
    }
} else {
    for (int j=0; j<WARP_SIZE; j++){
        mult(j, tmp)   // FP32
    }
}
```

The time measurements are listed below:

No. of FP32 Warps	No. of FP64 Warps	Time Taken
8	0	4.82
7	1	5.07
6	2	5.04
5	3	5.07

We can observe that warp specialisation doesn't benefit matrix multiplication. The time taken increases with no. of FP64 warps.
Hence, this is not suitable for matrix multiplication.

Files

1. MatrixMulFP32FP64/mat_mul.cu: Matrix multiplication using only shared memory
2. MatrixMulFP32FP64/mat_mul_fp32_fp64.cu: Matrix multiplication using ILP
3. MatrixMulFP32FP64/mat_mul_fp32_fp64_warp.cu: Matrix multiplication using warp specialisation
4. MatrixMulFP32FP64/mat_mul_pipeline.cu: Matrix multiplication using cuda pipeline API

How to run

Just compile the code and run the executables:
nvcc <file>.cu -gencode arch=compute_80,code=sm_80 -o executable
./executable <mat1> <mat2> <output_path>
The input files must be csv.

Cuda Pipeline API

The cuda pipeline API allows us to write code that can be used to parallelise memory and compute operations. The pipeline API was used to write the matrix multiplication. But there was no performance improvement. The Nsight program showed that there were many bank conflicts after using pipeline API. This might be a reason for the increase in time.
Here are the time measurements for two 1024x1024 matrices:

1. Non-pipelined time: 12.25 ms
2. Pipelined time: 14.19 ms

Methodology

In normal matrix multiplication, blocks of memory (32x32) are fetched and then computation is performed. This is done in a loop for large matrices. We can parallelise the memory fetch and computation using the cuda pipeline API.

1. Use two buffers for fetching a block of matrix
2. Fetch the first block in first buffer
3. Start the fetch of the second block asynchronously in the second buffer and perform computation on the first buffer.
4. Then after the first computation and second fetch, start fetching the block in the first buffer and perform computation on the second buffer.
5. Continue this till all the blocks of matrices are fetched and processed.

Files

MatrixMulFP32FP64/mat_mul_pipeline.cu: Matrix multiplication using cuda pipeline API

How to run

Just compile the code and run the executables:
nvcc <file>.cu -gencode arch=compute_80,code=sm_80 -o executable
./executable <mat1> <mat2> <output_path>

CudaDMA | Warp Specialisation

Parallelism in CudaDMA

CudaDMA enables blocking and unblocking of warps based on threadIdx.x, allowing memory-fetching threads to signal compute threads. Warps are divided into:

1. **DMA warps:** Handle data transfer from global to shared memory.
2. **Compute warps:** Perform computation on prefetched data.

I have used the following architecture -

- 2 DMA warp for memory fetching
- 1 Compute warp for matrix multiplication

This setup overlaps memory transfer with computation using explicit synchronisation calls in CudaDMA.

Base Code

This version of matrix multiplication is used as a baseline for performance comparison.

Pseudo Code:

1. Input matrices are loaded from CSV files.
2. Dimensions are padded to be multiples of the warp size (32).
3. Each CUDA thread block:

    a. Loads a 32×32 tile from each input matrix into shared memory.
    b. Performs partial matrix multiplication using these tiles.
    c. Synchronises threads between loading and computation.
4. Results are written to a CSV file.
5. Execution time is recorded for performance analysis.

Size of Matrix A	Size of Matrix B	Time Taken
2048x2048	2048x2048	10.09s
1024x1024	1024x1024	1.94s
128x128	128x128	0.092
32x32	32x32	0.051

Algorithms Tried

1. Algorithm-1 | One Compute warp and One DMA warp

Pseudo Code:

Warp 1 (DMA Warp) Behaviour:

- Fetch memory from global memory.
- Once the memory is fetched, send a signal indicating the fetch is complete.

Warp 2 (Compute Warp) Behaviour:

- Perform computation using the previously fetched memory from Warp 1.
- Upon completing the computation, signal Warp 1 to start fetching new memory.

Memory Fetching and Computation Loop:

1. Warp 1 starts fetching memory after receiving the signal from Warp 2 (indicating that the previous memory fetch is complete).
2. Warp 2 continues computation once Warp 1 has completed the memory fetch.
3. This process continues in a continuous loop, alternating between memory fetching and computation.

However, this algorithm could not be successfully implemented due to a limitation in CudaDMA. CudaDMA provides only blocking and non-blocking calls, but it lacks a mechanism to trigger signals that would allow conditional branching within warps (such as if-else conditions). Without such signalling capabilities, the warps cannot synchronise dynamically based on computational or memory-fetching needs, making the desired control flow unachievable.

2. Algorithm-2 | One Compute Warp and One DMA Warp

Pseudo Code:

Initialise Shared Memory Buffers:

- Two shared memory buffers, A_shared and B_shared, are declared to hold tiles of the input matrices A and B, respectively.

- These buffers are sized for 32×32 tiles, enabling warp-level cooperative loading and computation.

Start Asynchronous DMA Operations:

- A single DMA warp is initialized using the cudaDMASequential class.

- This warp is responsible for asynchronously loading data tiles from global memory into shared memory before they are consumed by the compute warp.
- The compute warp initiates an asynchronous DMA load using `dma_ld_0.start_async_dma()`.
- While the compute warp waits for the data to be ready (`dma_ld_0.wait_for_dma_finish()`), the DMA warp begins transferring the next tile.

- Once data is available in shared memory, the compute warp performs tile-level matrix multiplication, accumulating partial results into the output matrix C.

Matrix Multiplication Loop:

- Each thread in the compute warp computes a row of the tile matrix product.

- The results are accumulated into the global output matrix C.

Store the Results:

- The computed results stored in C are copied back to host memory after kernel execution.

- Execution time is measured to evaluate performance.

Size of Matrix A	**Size of Matrix B**	**Time Taken**	Base Time Taken in Milliseconds
2048x2048	2048x2048	12.146	5.475
1024x1024	1024x1024	1.733	0.792

128x128	128x128	0.085	0.047
32x32	32x32	0.055	0.041

Observations:

1. DMA-Based Execution Is Slower Than Base Implementation:
2. The overhead introduced by managing asynchronous DMA operations and warp coordination appears to outweigh the benefits of overlapped execution in this configuration. This is particularly evident in smaller matrices (128×128), where the DMA implementation incurs nearly double the execution time compared to the base case.

3. Algorithm-3 | Two Compute Warps and Two DMA Warps (Overlapping Fetch and Computation)

1. **Initialise Shared Memory Buffers:**
    a. Use `buff` and `mat` as shared memory buffers to store matrix tiles.
2. **Start Asynchronous DMA Operations:**
    a. Use two DMA warps (`dma_ld_0` and `dma_ld_1`) to load two different tiles of the input matrices into shared memory asynchronously.
3. **Matrix Multiplication in Parallel:**
    a. Warp 1 (Compute Warp) starts processing the first tile of matrix `mat1` and `mat2`.
    b. Warp 2 (Compute Warp) begins with the second tile of matrix `mat1` and `mat2`, processing the data while Warp 1 waits for the next fetch.
4. **Synchronisation for Tile Switch:**
    a. After finishing computation on one tile, compute warps wait for DMA warps to complete loading the next tiles.
    b. Once DMA warps finish loading the next set of data, the compute warps continue with the new tiles.
5. **Final Result Storage:**
    a. After all tiles are processed, store the results in the output matrix `y`.

Algorithm 3 offers limited improvement in performance. The time complexity just gets halved according to Moore's Law, similar to traditional tiling, since the overall computational load doesn't change. While overlapping memory transfers and computation helps, the process is still constrained by memory latency and the need for synchronisation between DMA and compute warps, which adds overhead. The fixed tile size of 32×32 further limits flexibility and doesn't optimise for varying matrix sizes, meaning there is little to no gain in performance for different problem sizes.

To achieve better performance, improvements like **adaptive tiling** (dynamically adjusting tile sizes based on matrix dimensions) could enhance memory utilisation. Additionally, leveraging **multiple CUDA streams** to overlap operations further and minimising synchronisation costs between warps would reduce bottlenecks and improve overall efficiency. Thus, while

Algorithm-3 introduces parallelism, it doesn't significantly reduce execution time due to the memory access and synchronisation constraints.

Key Takeaways

1. As cudaDMA does not have any functions or signals that can be used to write code that lets warps decide which control flow path to take.
2. Hence, the role of the warps, both DMA and compute, must be well defined before and should have a singular purpose.
3. This eliminates the possibility of better and more efficient algorithms.

Ending Note

1. I could not debug the code for cudaDMA due to time contraints and could not obtain benchmarking-related numbers for comparison.

Files

1. `submission/CudaDMA/mat_mul.cu` is the file with the code for tiling.
2. `submission/CudaDMA/mat_mul_2.cu` is the file containing the algorithm for Algorithm 3.
3. Instructions to Run
    a. nvcc matmul.cu -o matmul
    b. nvcc matmul_base.cu -o matmul_base
    c. ./matmul <path to file1> <path to file2>
    d. ./matmul_base <path to file1> <path to file2>

Tensor Cores for Matrix Multiplication

For utilizing tensor cores for matrix multiplication, wmma api was used
The following kernels were implemented to compare the performance of fp32 and tensor cores:

Kernel that executes just fp32 cores:

```
__global__ void gemm_kernel(float *A, float *B, float *C, int M, int N, int K) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    if (row < M && col < N) {
        float sum = 0.0f;
        for (int k = 0; k < K; ++k) {
            sum += A[row * K + k] * B[k * N + col];
        }
        C[row * N + col] = sum;
    }
}
```

Kernel that just executes tensor cores:

```
// WMMA kernel for Tensor Core GEMM (assumes padded inputs)
__global__ void wmma_gemm(half *A, half *B, float *C, int M, int N, int K) {
    using namespace nvcuda::wmma;
    const int WMMA_M = 16, WMMA_N = 16, WMMA_K = 16;

    int warpM = (blockIdx.x * blockDim.x + threadIdx.x) / warpSize;
    int warpN = blockIdx.y;

    if (warpM * WMMA_M >= M || warpN * WMMA_N >= N) return;

    fragment<matrix_a, WMMA_M, WMMA_N, WMMA_K, half, col_major> a_frag;
    fragment<matrix_b, WMMA_M, WMMA_N, WMMA_K, half, row_major> b_frag;
    fragment<accumulator, WMMA_M, WMMA_N, WMMA_K, float> acc_frag;

    fill_fragment(acc_frag, 0.0f);

    for (int k = 0; k < K; k += WMMA_K) {
        if (k + WMMA_K <= K) {
            load_matrix_sync(a_frag, A + warpM * WMMA_M * K + k, K);
```

```
        load_matrix_sync(b_frag, B + k * N + warpN * WMMA_N, N);
        mma_sync(acc_frag, a_frag, b_frag, acc_frag);
    }
  }

  store_matrix_sync(C + warpM * WMMA_M * N + warpN * WMMA_N, acc_frag, N,
mem_row_major);
}
```

For CUDA cores, the kernel configuration was:

```
dim3 block(16, 16);
dim3 grid((N + block.x - 1)/block.x, (M + block.y - 1)/block.y);
```

For tensor cores:

```
dim3 grid_tc(M_pad / 16, N_pad / 16);
dim3 block_tc(32, 4);
```

Where M_pad and N_pad are such that the matrices are padded to the nearest multiple of 16.

Performance comparison:

- For 32*32 input:

  ```
  Regular CUDA Time: 0.027 ms
  Tensor Core Time: 0.009 ms
  ```

- For 1024*1024 input:

  ```
  Regular CUDA Time: 1.123 ms
  Tensor Core Time: 0.694 ms
  ```

- For 2048*2048 input:

  ```
  Regular CUDA Time: 7.226 ms
  Tensor Core Time: 5.373 ms
  ```

- For 2*2 input:

  ```
  Regular CUDA Time: 0.024 ms
  ```

```
Tensor Core Time: 0.008 ms
```

- For 128*128 input:

```
Regular CUDA Time: 0.028 ms
Tensor Core Time: 0.013 ms
```

Overall, tensor cores give a better performance than the fp32 cores for general matrix multiplication case.

Attempt to execute tensor cores and fp32 cores in parallel:

In our attempt to leverage both Tensor Cores and FP32 cores simultaneously for matrix multiplication, we encountered several challenges rooted in the unique ways these cores operate. Below is a detailed breakdown of why warp specialization, where some threads use Tensor Cores and others use FP32 cores, did not yield the desired performance.

1. Warp Divergence
In CUDA, a warp is a group of 32 threads that execute the same instruction in parallel. For efficient computation, all threads within a warp need to execute the same type of instruction.

Tensor Cores are optimized for high-throughput matrix operations, particularly using mixed precision (FP16 inputs with FP32 accumulation). They operate on small, structured matrix tiles (e.g., 16x16), and the entire warp of 32 threads is responsible for a portion of the computation using specialized instructions like wmma::load_matrix_sync and mma_sync.

FP32 Cores are general-purpose cores designed for single-precision floating-point operations. These cores work at the thread level, where each thread independently performs a computation (without needing specialized Tensor Core instructions).

The primary issue when attempting warp specialization arises from the incompatibility of these two computational models within the same warp.

Warp Divergence occurs when some threads are executing Tensor Core operations (e.g., matrix multiplication), while others are performing FP32-based operations. This causes threads within the warp to execute different instructions, leading to inefficiencies. Some threads may be idle while others are executing, causing underutilization of resources and reducing the overall efficiency of the warp.

Why this is a problem: Efficient warp-level execution is achieved when all threads in the warp perform the same instruction. Mixing Tensor Cores and FP32 in the same warp violates this principle, resulting in warp divergence and reduced performance.

2. Resource Contention
Both Tensor Cores and FP32 Cores share limited hardware resources, such as shared memory, registers, and execution units. When these cores are used in parallel within the same kernel, they may compete for these shared resources.

For example, if Tensor Cores are working on one part of the matrix and FP32 cores are handling another part, both types of cores may need to access shared memory simultaneously, leading to resource contention.

Why this is a problem: The GPU scheduler can only issue one type of instruction per warp. If a warp contains threads that use both Tensor Cores and FP32 cores, the scheduler may not fully utilize either resource, leading to serialization (delays) or stalling (pauses in execution), both of which hinder performance.

3. Latency and Synchronization Overheads
Parallel execution of Tensor Cores and FP32 cores introduces potential latency and synchronization issues.

If some threads are performing computations using Tensor Cores and others are using FP32 cores, there may be data dependencies where one group's results are required by another group. Ensuring proper synchronization between these groups is non-trivial.

Data Dependencies must be managed carefully. For example, if one group of threads computes part of matrix C using Tensor Cores and another group uses FP32 cores to compute a different part, synchronization is required to ensure that there are no race conditions (i.e., simultaneous access to shared memory) and that the results from one group are ready before another starts using them.

Why this is a problem: The need for synchronization between groups of threads leads to overhead. This overhead can significantly slow down execution and prevent us from achieving the potential performance gains from parallel computation.

4. Conceptual Summary of Tensor Cores
Tensor Cores are designed to efficiently handle small matrix tiles (e.g., 16x16) and are optimized for high-throughput matrix multiplication using mixed precision (FP16 inputs with FP32 accumulation).

Tensor Cores rely on structured data (matrix tiles) for maximum efficiency. When mixed with FP32 cores in an unstructured or non-uniform manner, the performance benefits of Tensor Cores are diminished.

Tensor Cores operate most efficiently at the warp level, with all threads performing the same type of matrix operation. Attempting to mix them with FP32 cores within the same warp disrupts this high-throughput, warp-level parallelism, resulting in suboptimal performance.

Why this is a problem: Tensor Cores are specialized for warp-level matrix operations with structured data. Mixing them with FP32 cores in the same kernel introduces inefficiencies due to warp divergence, resource contention, and synchronization challenges.

Files
Tensor/mat_mul_tensor.cu

To compile the code:
nvcc -arch=sm_80 -o tensor_mat_mul tensor_mat_mul.cu

Conclusion

In this project we explored ILP and Warp Specialisation for FP32 and FP64 parallelism, cuda DMA for parallel memory and compute operations, and tensor cores for fast less precision matrix multiplication. We found that ILP can increase the throughput and decrease matrix multiplication time by ~6%.
Warp specialization, where Tensor Cores and FP32 cores are mixed within the same warp, leads to several inefficiencies, including warp divergence, resource contention, and synchronization overheads.

As cudaDMA does not have any functions or signals that can be used to write code that lets warps decide which control flow path to take.
Hence, the role of the warps, both DMA and compute, must be well defined before and should have a singular purpose.
This eliminates the possibility of better and more efficient algorithms.

Github Link: https://github.com/abhinavydv/IPACO-Project-GPU-Throughput