CENG 5030
Energy Efficient Computing

Implementation 05: CUDA

Bei Yu
CSE Department, CUHK
byu@cse.cuhk.edu.hk

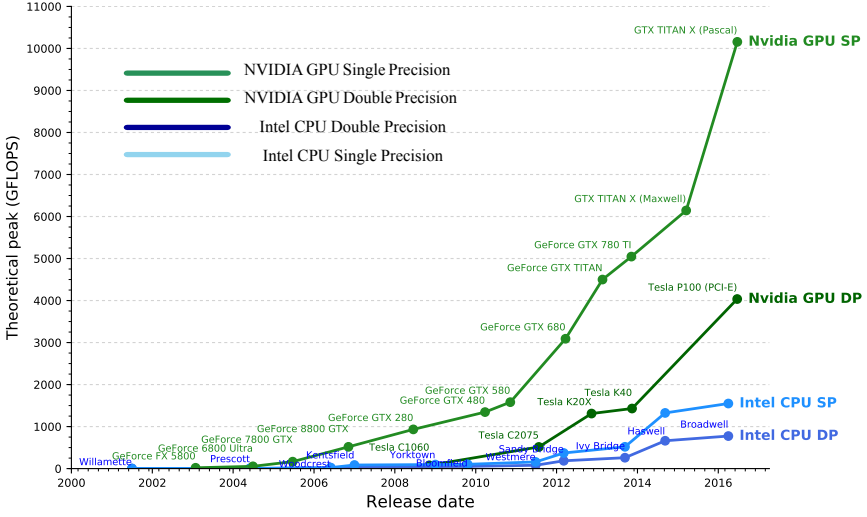
(Latest update: December 4, 2023)

2023 Fall

Overview

Introduction

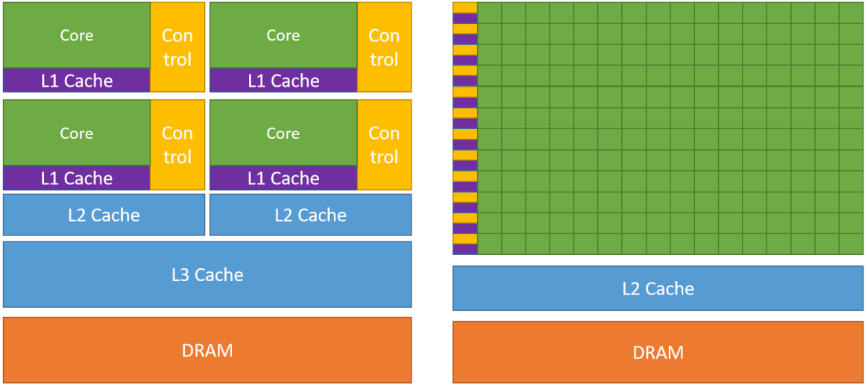
Extreme Computational Power of GPU's



GPU vs. CPU:

- Provide much higher instruction throughput and memory bandwidth than CPU within a similar price and power envelop.

Difference between CPU's and GPU's



CPU - Minimize latency

- Majority of transistors are dedicated to:
  - Advanced Control Logic
  - Large Cache

GPU - Maximize throughput

- Majority of transistors are dedicated to:
  - Data processing

Key Advantages of GPU Architecture

- **High Throughput and Parallelism:** GPUs excel in executing the same program on many data elements simultaneously.
- **Energy Efficient and Flexibility:** GPUs can be energy efficient like FPGAs but offer much more programming flexibility.
- **Maximized Performance:** In applications, utilization of GPUs with CPUs can exploit the high degree of parallelism, achieving superior performance.

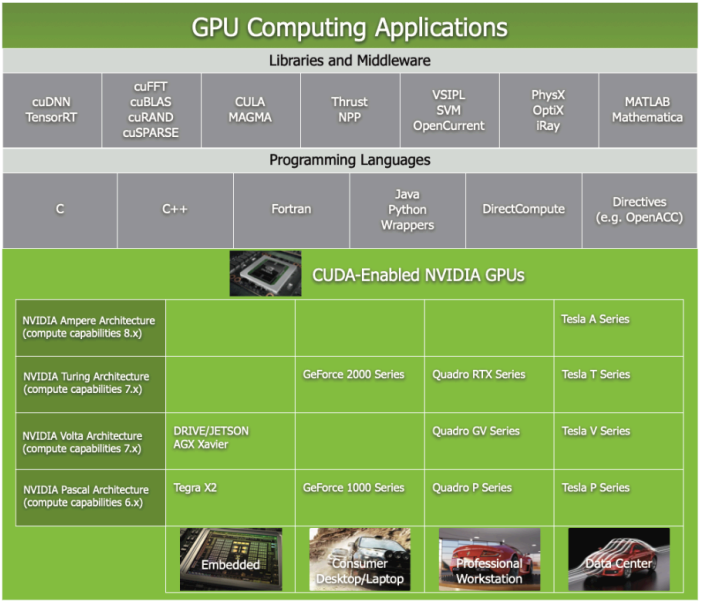
CUDA: Compute Unified Device Architecture

CUDA is a parallel computing platform by NVIDIA that leverages the power of GPUs.

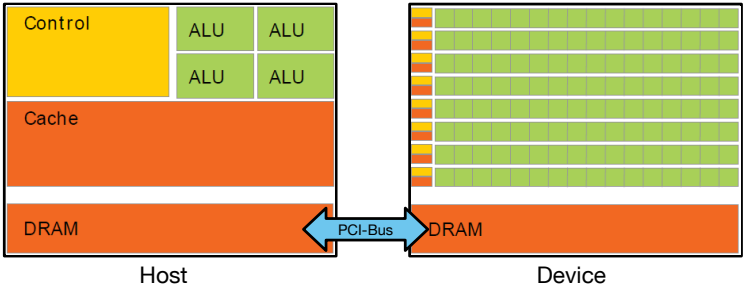
Key Features

- Enables more efficient problem-solving than on a CPU (By libraries).
- Supports C++ as a high-level programming language.
- Accommodates other languages and APIs, such as Python, DirectCompute, and OpenACC.

CUDA: Compute Unified Device Architecture



Host, Device, and CUDA Execution Process



Concepts

- Host: CPU
- Device: GPU
- Heterogeneous:
  - Combination: Host + Device
  - Leverages both for optimum performance

CUDA Execution Process

1. Transfer data from host to device.
2. Perform computations using CUDA kernel on device.
3. Transfer results from device to host.

How to utilize the massive number of CUDA cores?

- **Kernels:** Functions that run on the GPU.
- Kernels are executed N times in parallel by N different threads.
  - **Execution:** CUDA threads execute in a SIMT (Single Instruction Multiple Threads) fashion.
    - *Enable thread-level parallel code for independent, scalar threads.*
  - **Divergence:** Within a wrap, branch divergence occurs.
    - *Diverge via a data-dependent conditional branch.*

Thread Hierarchy

Threads

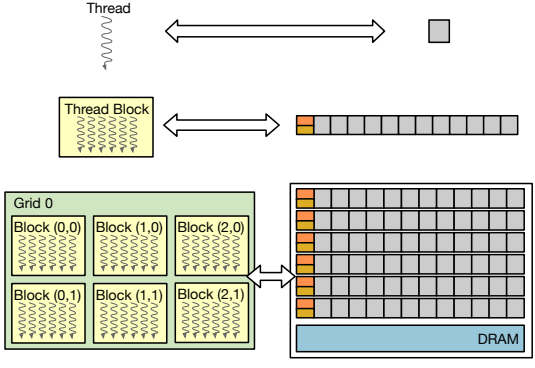
- Kernels are executed by Threads

Blocks

- Threads on the same SM are grouped into Blocks
  - *SM: streaming multiprocessor*

Grid

- Blocks are grouped into Grids
- Each Kernel launch creates a single Grid

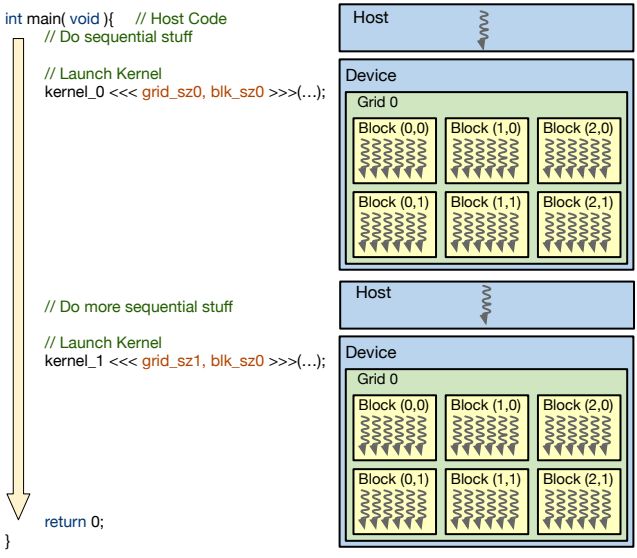


$$\text{Thread} \in \text{Block} \in \text{Grid}$$

Programming Model

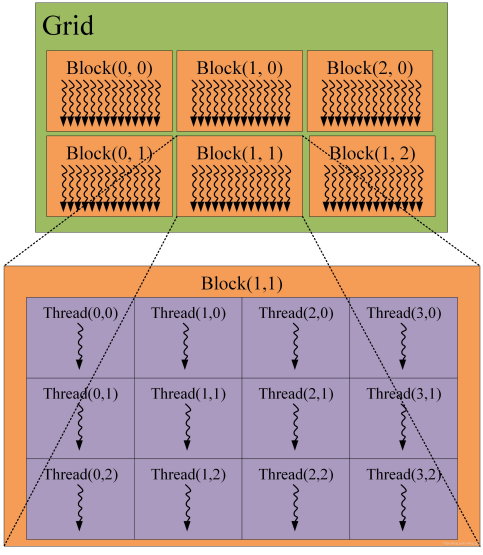
Program Flow: Overview

- Host Code
  - Serial work
  - Launch Kernels
- Device Code
  - Parallel work



Program Flow: Kernel Launch Syntax

```
// Block and Grid dimensions
// Default values are (1,1,1)
dim3 grid_size(x, y, z);
dim3 block_size(x, y, z);


// Launch Kernel
kernelName <<< grid_size,  block_size >>>( … );
```



Program Flow: An example

```
int main( void ){

    // Declare variables
    int *h_c, *d_c;

    // Allocate memory on the device
    cudaMalloc( (void**)&d_c, sizeof(int) );

    // Copy data from Host to Device
    cudaMemcpy( d_c, h_c, sizeof(int), cudaMemcpyHostToDevice );

    // Configuration Parameters
    dim3 grid_size(1);   dim3 block_size(1);

    // Launch Kernel
    kernel_0 <<< grid_size, block_size >>>(...);

    // Copy data from Device to Host
    cudaMemcpy( h_c, d_c, sizeof(int), cudaMemcpyDeviceToHost);

    // De-allocate memory
    cudaFree(  d_c );  free( h_c );

    return 0;
}
```

Parallelizing a For-Loop

Kernel Definition

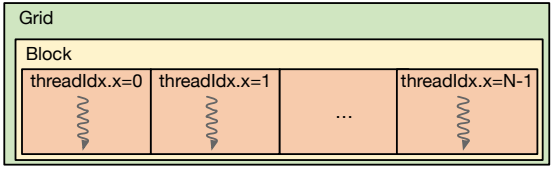
```
__global__ void kernel( int *d_out, int *d_in)
{
        // Perform this operation for every thread
        d_out[0] = d_in[0];
}
```

Thread Index

- Accessible within a Kernel through the built in threadIdx variable.
- Thread Blocks can have as many as 3-dimensions, therefore there is a corresponding index for each dimension:

threadIdx.x
threadIdx.y
threadIdx.z

```
// Configuration Parameters
dim3 grid_size(1);
dim3 block_size(N);

Grid Dimension: 1*1*1 —> 1 Block

Block Dimension: N*1*1 —> N Threads
```



Parallelizing a For-Loop: Comparison

CUDA Program

```
// Kernel Definition
__global__ void increment_gpu( int *a, int *N)
{
      int i = threadIdx.x;
      if (i < N)
          a[i] = a[i] + 1;
}

int main( void )
{
      int h_a[N] = // …

      // Allocate arrays in Device memory
      int* d_a; cudaMalloc( (void**)&d_a, N * sizeof(int) );

      // Copy memory from Host to Device
      cudaMemcpy( d_a, h_a, N * sizeof(int), cudaMemcpyHostToDevice );

      // Configuration Parameters
      dim3 grid_size(1);  dim3 block_size(N);

      // Launch Kernel
      increment_gpu <<< grid_size, block_size >>>(d_a, N);

      // …
      return 0;
}
```

CPU Program

```
// Function Definition
void increment_cpu( int *a, int *N)
{
      for (int I=0; i<N; I++)
          a[i] = a[i] + 1;
}


int main( void )
{
      int a[N] = // …

      // Call Function
      increment_cpu( a, N );

      // …
      return 0;
}
```

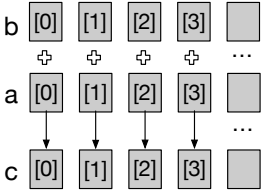
Vector Addition - A Very Parallel Problem

Consider two vectors, $\boldsymbol{a}$ and $\boldsymbol{b}$, each of size N:

$$\boldsymbol{a} = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{N-1} \end{bmatrix}, \quad \boldsymbol{b} = \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{N-1} \end{bmatrix}$$

The vector addition operation $\boldsymbol{c} = \boldsymbol{a} + \boldsymbol{b}$ can be executed in parallel:

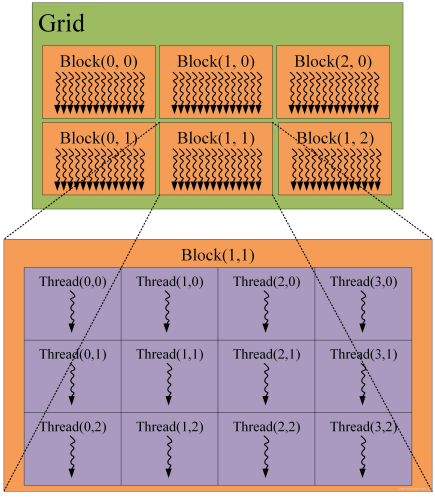
$$\boldsymbol{c} = \begin{bmatrix} a_0 + b_0 \\ a_1 + b_1 \\ \vdots \\ a_{N-1} + b_{N-1} \end{bmatrix} = \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{N-1} \end{bmatrix}$$



Each addition operation can be executed simultaneously, demonstrating the advantage of parallel computing.

Indexing Threads within Grids and Blocks

Built-In Variables



Dimension of a Grid

```
dim3 gridDim;
int gridDim.x;
int gridDim.y;
int gridDim.z;
```

Index of a Block

```
dim3 blockIdx;
int blockIdx.x;
int blockIdx.y;
int blockIdx.z;
```

Dimension of a Grid

```
dim3 blockDim;
int blockDim.x;
int blockDim.y;
int blockDim.z;
```

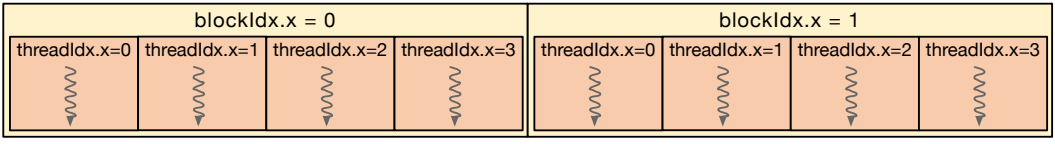
Index of a Thread

```
dim3 threadIdx;
int threadIdx.x;
int threadIdx.y;
int threadIdx.z;
```

Indexing Within Grid

- threadIdx is only unique within its own Thread Block
- To determine the unique Grid index of a Thread:

```
i = threadIdx.x + blockIdx.x * blockDim.x;
```



Example

```
// Launch Kernel
kernel <<< 3, 4 >>>(a);
```

`__global__ void kernel( int *a)`
`{`
`  int i = threadIdx.x + blockIdx.x * blockDim.x;`
`  a[i] = blockDim.x;`
`}`
	a : 4 4 4 4 4 4 4 4 4 4 4 4

`__global__ void kernel( int *a)`
`{`
`  int i = threadIdx.x + blockIdx.x * blockDim.x;`
`  a[i] = threadIdx.x;`
`}`
	a : 0 1 2 3 0 1 2 3 0 1 2 3

`__global__ void kernel( int *a)`
`{`
`  int i = threadIdx.x + blockIdx.x * blockDim.x;`
`  a[i] = blockIdx.x;`
`}`
	a : 0 0 0 0 1 1 1 1 2 2 2 2

`__global__ void kernel( int *a)`
`{`
`  int i = threadIdx.x + blockIdx.x * blockDim.x;`
`  a[i] = i;`
`}`
	a : 0 1 2 3 4 5 6 7 8 9 10 11

Thread-Memory Correspondence

Threads $\leftrightharpoons$ Local Memory (and Registers)

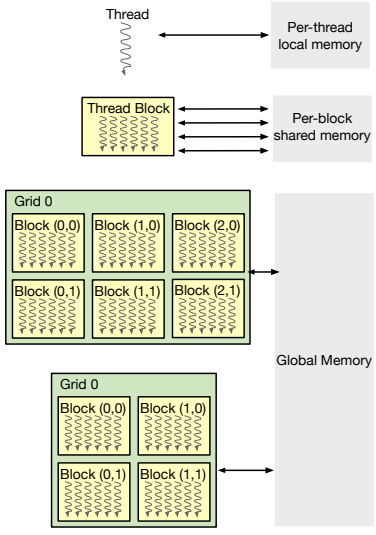
- Scope: Private to its corresponding Thread
- Lifetime: Thread

Blocks $\leftrightharpoons$ Shared Memory

- Scope: Every Thread in the Block has assess
- Lifetime: Block

Grids $\leftrightharpoons$ Global Memory

- Scope: Every Thread in all Grids have access
- Lifetime: Entire program in Host code - `main()`

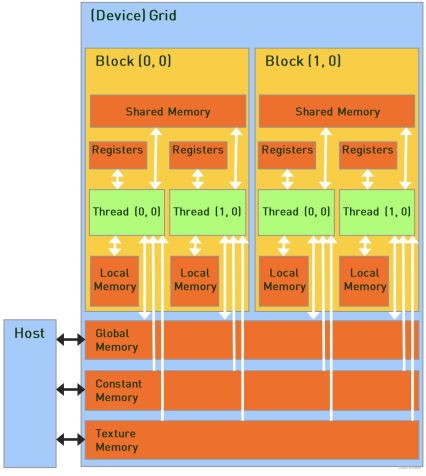


Memory Speed

- Relative speed of memory spaces: "Bandwidth"/"Latency"

Registers < Shared << Local ≈ Global << Host (PCIe)

~8TB/s	~1.5TB/s	~200GB/s	~5GB/s
~1clock	~32clock	~800clock	



Registers:

- Variables declared in a Kernel are stored in Registers
  - On-Chip
  - Fastest form of memory

Global Memory:

- Accessed with: `cudaMalloc()`, `cudaMemset()`, `cudaMemCopy()`, `cudaFree()`

Local Memory:

- Arrays too large to fit into Registers spill over into Local memory
  - Off-Chip
  - Compiler controlled
  - Local to each Thread

Shared Memory:

- Allows Threads within a Block to communicate with each other
  - Use synchronization
- Very fast
  - Only Registers are faster
- Can use as "Scratch-pad" memory

```
__global__ void kernel( int *in )
{
        int i = threadIdx.x + blockDim.x * blockIdx.x;

        // Allocate a shared array
        extern __shared__ int shared_array [];

       // each thread writes to one element of shared_array
       shared_array[i] = in[i];

       // Do more stu!
       //
}
```

Programming Practice

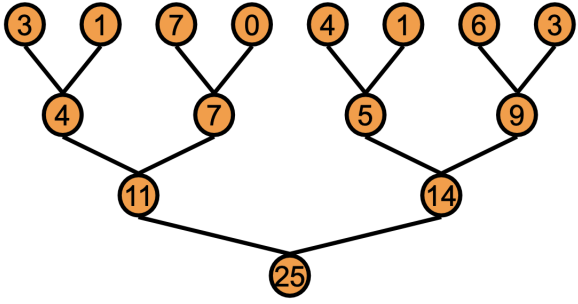
Optimizing Parallel Reduction in CUDA

Parallel Reduction

- **Vector reduction:** common and important data parallel primitive.
  - Computing vector dot-product.
  - Computing the norm of a vector.
  - Computing the average value of the elements in a vector.
  - …
- Easy to implement in CUDA (but hard to make it efficient).
- Serves as a great optimization example.
  - We'll walk step by step through 7 different versions.
  - Demonstrates several important optimization strategies.

Parallel Reduction

- Tree-based approach used within each thread block:



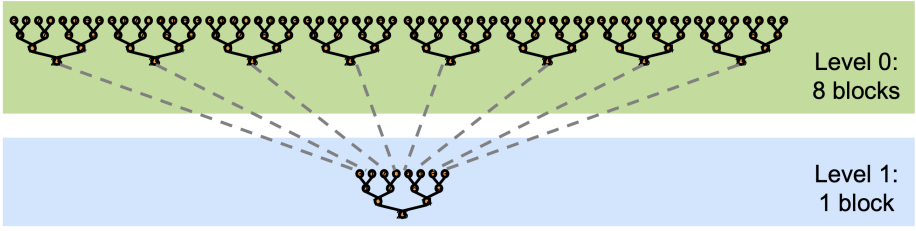
- Need to be able to use multiple thread blocks.
  - To process very large arrays.
  - To keep all multiprocessors on the GPU busy.
  - Each thread block reduces a portion of the array.
- But how do we communicate partial results between thread blocks?

Problem: Global Synchronization

- If we could synchronize across all thread blocks, could easily reduce very large arrays.
  - Global sync after each block produces its result;
  - Once all blocks reach sync, continue recursively.
- However, CUDA has no global synchronization. Reason:
  - Expensive to build in hardware for GPUs with high processor count.
  - Would force programmer to run fewer blocks (no more than # multiprocessors * # resident blocks / multiprocessor) to avoid deadlock, which may reduce overall efficiency.
- Solution: decompose into multiple kernels.
  - Kernel launch serves as a global synchronization point.
  - Kernel launch has negligible HW overhead, low SW overhead.

Solution: Kernel Decomposition

- Avoid global sync by decomposing computation into multiple kernel invocations.
- In the case of reductions, code for all levels is the same: recursive kernel invocation.



Optimization Goal

- We should strive to reach GPU peak performance.
- Choose the right metric:
  - GFLOP/s: for compute-bound kernels.
  - Bandwidth: for memory-bound kernels.
- For reductions: very low arithmetic intensity (1 FLOP per element loaded, bandwidth-optimal)
- Therefore, we should strive for **peak bandwidth**

- Example device: Nvidia G80 GPU
  - 384-bit memory interface, 900 MHz DDR.
  - Bandwidth 384 * 1800 / 8 = 86.4 GB/s

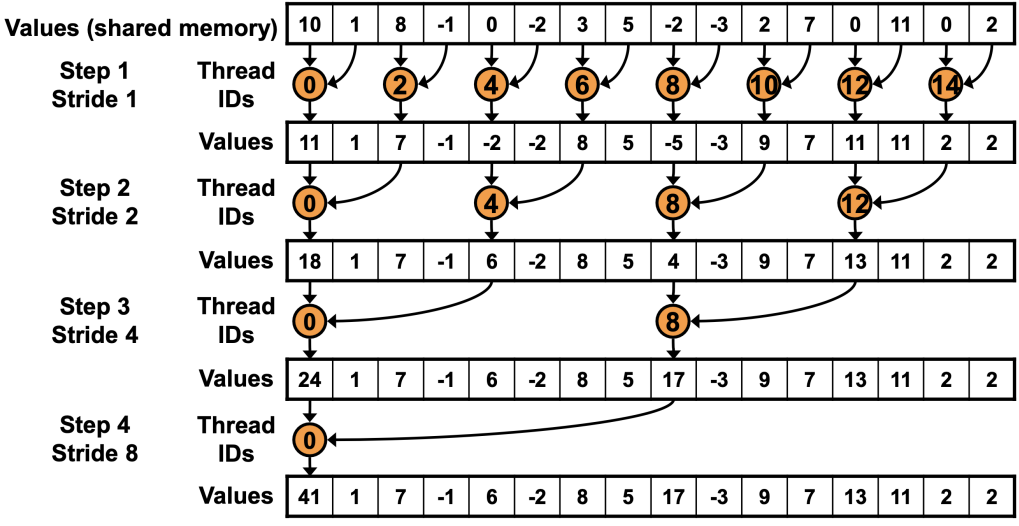
Impl # 1: Interleaved Addressing

Kernel function code:

```
__global__ void reduce0(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];

    // each thread loads one element from global to shared mem
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    __syncthreads();

    // do reduction in shared mem
    for(unsigned int s=1; s < blockDim.x; s *= 2) {
        if (tid % (2*s) == 0) {
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }

    // write result for this block to global mem
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Impl # 1: Interleaved Addressing

An illustration of the workflow of threads:



Impl # 1: Interleaved Addressing

Problem of implementation # 1:

```
__global__ void reduce1(int *g_idata, int *g_odata) {
  extern __shared__ int sdata[];

  // each thread loads one element from global to shared mem
  unsigned int tid = threadIdx.x;
  unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
  sdata[tid] = g_idata[i];
  __syncthreads();

  // do reduction in shared mem
  for (unsigned int s=1; s < blockDim.x; s *= 2) {
    if (tid % (2*s) == 0) {
      sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
  }

  // write result for this block to global mem
  if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Problem: highly divergent warps are very inefficient, and % operator is very slow

Performance Evaluation for Reduction with 4M Elements

	Time (2^{22} ints)	Bandwidth
Kernel 1: **interleaved addressing with divergent branching**	**8.054 ms**	**2.083 GB/s**

Note: Block Size = 128 threads for all tests

Impl # 2: Interleaved Addressing with Non-divergent Branch

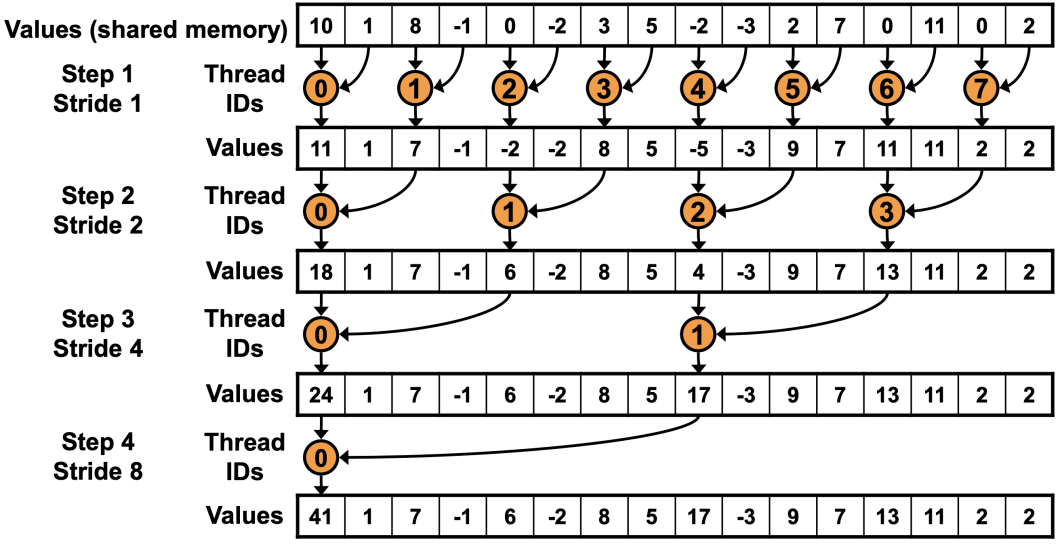
Replace divergent branch in inner loop:

```
for (unsigned int s=1; s < blockDim.x; s *= 2)  {
    if (tid % (2*s) == 0) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```

With strided index and non-divergent branch:

```
for (unsigned int s=1; s < blockDim.x; s *= 2)  {
    int index = 2 * s * tid;

    if (index < blockDim.x) {
        sdata[index] += sdata[index + s];
    }
    __syncthreads();
}
```

Impl # 2: Interleaved Addressing with Non-divergent Branch



New Problem: Shared Memory Bank Conflicts

Performance Evaluation for Reduction with 4M Elements

	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x

Impl # 3: Sequential Addressing

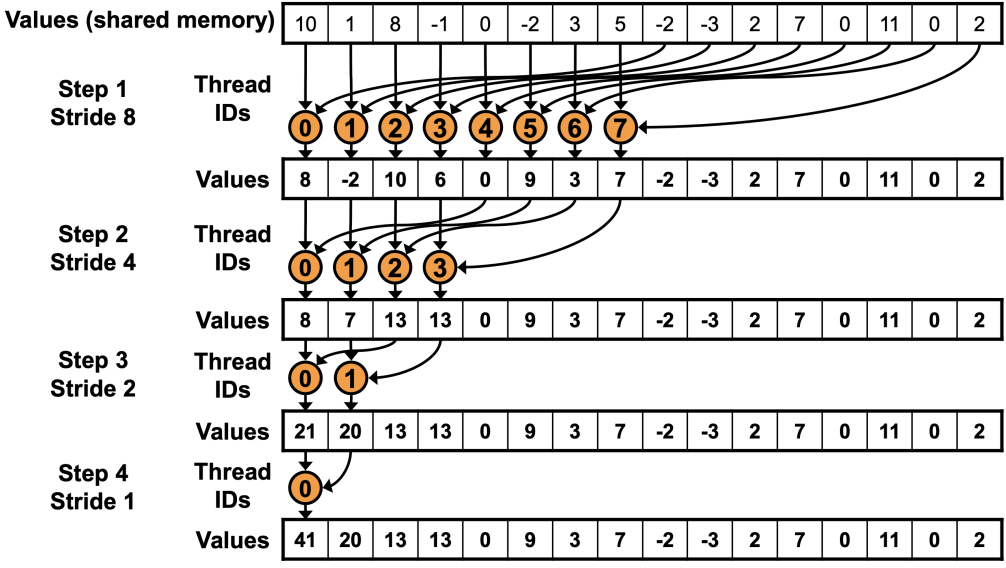
Replace strided indexing in inner loop:

```
for (unsigned int s=1; s < blockDim.x; s *= 2)  {
    int index = 2 * s * tid;

    if (index < blockDim.x) {
        sdata[index] += sdata[index + s];
    }
    __syncthreads();
}
```

With reverse loop and threadID-based indexing:

```
for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
    if (tid < s) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```

Impl # 3: Sequential Addressing



Sequential addressing is conflict free

Performance Evaluation for Reduction with 4M Elements

	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x

Impl # 3: Sequential Addressing

Problem: Idle Threads

```
for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
    if (tid < s) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```

Half of the threads are idle on first loop iteration!

Impl # 4: First Add During Load

Halve the number of blocks, and replace single load:

```
// each thread loads one element from global to shared mem
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
sdata[tid] = g_idata[i];
__syncthreads();
```

With two loads and first add of the reduction:

```
// perform first level of reduction,
// reading from global memory, writing to shared memory
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
__syncthreads();
```

Performance Evaluation for Reduction with 4M Elements

	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x
Kernel 4: first add during global load	0.965 ms	17.377 GB/s	1.78x	8.34x

Bottleneck: Instruction Overhead

- At 17 GB/s, we're far from bandwidth bound.
  - And we know reduction has low arithmetic intensity.
- Therefore, a likely bottleneck is instruction overhead.
  - Ancillary instructions that are not loads, stores, or arithmetic for the core computation.
  - In other words: address arithmetic and loop overhead.
- Strategy: **Loop Unrool**

Strategy: Unroll the Last Warp

- As reduction proceeds, the number of "active" threads decreases.
  - When stride $s \leq 32$, we have only one warp left.
- Instructions are SIMD synchronous within a warp.
- That means when stride $s \leq 32$:
  - We don't need to `__syncthreads()`.
  - We don't need "`if (tid < s)`" because it doesn't save any work.
- Solution: unroll the last 6 iterations of the inner loop.

Impl # 5: Last Warp Unrolling

```
__device__ void warpReduce(volatile int* sdata, int tid) {
      sdata[tid] += sdata[tid + 32];
      sdata[tid] += sdata[tid + 16];
      sdata[tid] += sdata[tid +  8];
      sdata[tid] += sdata[tid +  4];
      sdata[tid] += sdata[tid +  2];
      sdata[tid] += sdata[tid +  1];
 }
```

IMPORTANT:
For this to be correct, we must use the "volatile" keyword!

```
// later…
 for (unsigned int s=blockDim.x/2; s>32; s>>=1) {
      if (tid < s)
         sdata[tid] += sdata[tid + s];
      __syncthreads();
 }

 if (tid < 32) warpReduce(sdata, tid);
```

Note: This saves useless work in all warps, not just the last one! Without unrolling, all warps execute every iteration of the for loop and if statement.

Performance Evaluation for Reduction with 4M Elements

	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x
Kernel 4: first add during global load	0.965 ms	17.377 GB/s	1.78x	8.34x
Kernel 5: unroll last warp	0.536 ms	31.289 GB/s	1.8x	15.01x

Impl # 6: Fully Unrolling

- If we know the number of iterations at compile time, we could completely unroll the reduction.
  - The block size (number of threads in a block) is limited by the GPU. (512 for G80 GPU)
  - We assume that the block sizes are power-of-2.
- We can easily unroll for a fixed block size, but we need to be generic – how can we unroll for block sizes that we don't know at compile time?
- Solution: **Templates**.
  - CUDA supports C++ template parameters on device and host functions.ß

Impl # 6: Fully Unrolling

Unrolling with Templates:

- Specify block size as a function template parameter.

```
template <unsigned int blockSize>
__global__ void reduce5(int *g_idata, int *g_odata)
```

Impl # 6: Fully Unrolling

```
Template <unsigned int blockSize>
__device__ void warpReduce(volatile int* sdata, int tid) {
    if (blockSize >= 64) sdata[tid] += sdata[tid + 32];
    if (blockSize >= 32) sdata[tid] += sdata[tid + 16];
    if (blockSize >= 16) sdata[tid] += sdata[tid +  8];
    if (blockSize >=  8) sdata[tid] += sdata[tid +  4];
    if (blockSize >=  4) sdata[tid] += sdata[tid +  2];
    if (blockSize >=  2) sdata[tid] += sdata[tid +  1];
}
```

```
if (blockSize >= 512) {
    if (tid < 256) { sdata[tid] += sdata[tid + 256]; } __syncthreads(); }
if (blockSize >= 256) {
    if (tid < 128) { sdata[tid] += sdata[tid + 128]; } __syncthreads(); }
if (blockSize >= 128) {
    if (tid <  64) { sdata[tid] += sdata[tid +  64]; } __syncthreads(); }

if (tid < 32) warpReduce<blockSize>(sdata, tid);
```

Note: all code in RED will be evaluated at compile time! Results in a very efficient inner loop.

Impl # 6: Fully Unrolling

Q: Do we still need block size at compile time?
A: Nope, just a switch statement for 10 possible block sizes:

```
switch (threads)
    {
    case 512:
      reduce5<512><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 256:
      reduce5<256><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 128:
      reduce5<128><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 64:
      reduce5< 64><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 32:
      reduce5< 32><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 16:
      reduce5< 16><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case  8:
      reduce5<  8><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case  4:
      reduce5<  4><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case  2:
      reduce5<  2><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case  1:
      reduce5<  1><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    }
```

Performance Evaluation for Reduction with 4M Elements

	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x
Kernel 4: first add during global load	0.965 ms	17.377 GB/s	1.78x	8.34x
Kernel 5: unroll last warp	0.536 ms	31.289 GB/s	1.8x	15.01x
Kernel 6: completely unrolled	0.381 ms	43.996 GB/s	1.41x	21.16x

Impl # 7: Multiple Adds per Thread

Replace load and add of two elements:

```
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
__syncthreads();
```

With a while loop to add as many as necessary:

```
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockSize*2) + threadIdx.x;
unsigned int gridSize = blockSize*2*gridDim.x;
sdata[tid] = 0;

while (i < n) {
   sdata[tid] += g_idata[i] + g_idata[i+blockSize];
   i += gridSize;
}
__syncthreads();
```

Impl # 7: Multiple Adds per Thread

Replace load and add of two elements:

```
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
__syncthreads();
```

With a while loop to add as many as necessary:

```
unsigned int tid = th
unsigned int i = blo
unsigned int gridSi
sdata[tid] = 0;

while (i < n) {
   sdata[tid] += g_id[i] + g_idata[i+blockSize];
   i += gridSize;
}
__syncthreads();
```

Note: gridSize loop stride to maintain coalescing!

Performance Evaluation for Reduction with 4M Elements

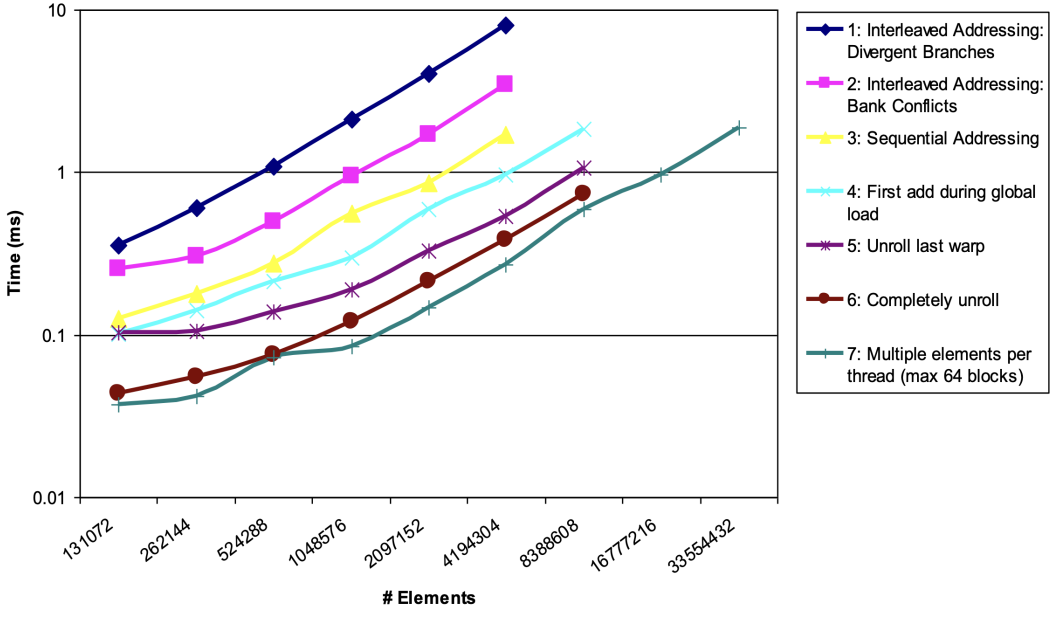
	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x
Kernel 4: first add during global load	0.965 ms	17.377 GB/s	1.78x	8.34x
Kernel 5: unroll last warp	0.536 ms	31.289 GB/s	1.8x	15.01x
Kernel 6: completely unrolled	0.381 ms	43.996 GB/s	1.41x	21.16x
Kernel 7: multiple elements per thread	0.268 ms	62.671 GB/s	1.42x	30.04x

Final Optimized Kernel Function

```
template <unsigned int blockSize>
__device__ void warpReduce(volatile int *sdata, unsigned int tid) {
   if (blockSize >=  64) sdata[tid] += sdata[tid + 32];
   if (blockSize >=  32) sdata[tid] += sdata[tid + 16];
   if (blockSize >=  16) sdata[tid] += sdata[tid +  8];
   if (blockSize >=   8) sdata[tid] += sdata[tid +  4];
   if (blockSize >=   4) sdata[tid] += sdata[tid +  2];
   if (blockSize >=   2) sdata[tid] += sdata[tid +  1];
}
template <unsigned int blockSize>
__global__ void reduce6(int *g_idata, int *g_odata, unsigned int n) {
   extern __shared__ int sdata[];
   unsigned int tid = threadIdx.x;
   unsigned int i = blockIdx.x*(blockSize*2) + tid;
   unsigned int gridSize = blockSize*2*gridDim.x;
   sdata[tid] = 0;

   while (i < n) { sdata[tid] += g_idata[i] + g_idata[i+blockSize];  i += gridSize;  }
   __syncthreads();

   if (blockSize >= 512) { if (tid < 256) { sdata[tid] += sdata[tid + 256]; } __syncthreads(); }
   if (blockSize >= 256) { if (tid < 128) { sdata[tid] += sdata[tid + 128]; } __syncthreads(); }
   if (blockSize >= 128) { if (tid <  64) { sdata[tid] += sdata[tid +  64]; } __syncthreads(); }

   if (tid < 32) warpReduce(sdata, tid);
   if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

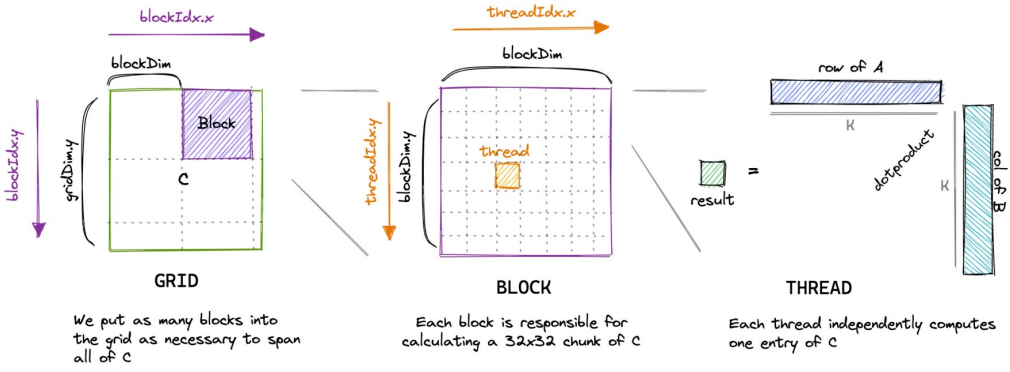
Final Optimized Kernel

Performance Comparison of Impl # 1 to # 7



Optimizing GEMM in CUDA

Naive implementation, each thread compute one entity in C.



This kernel takes about 0.5s to process three 4092×4092 fp32 matrices on A6000 GPU.

Matmul - lowerbound

- Total FLOPS: $2 * 4092^3 + 4092^2 = 137$ GFLOPS
- Total data to read (minimum!): $3 * 4092^2 * 4\text{B} = 201$ MB
- Total data to store: $4092^2 * 4\text{B} = 67$ MB

GPU : 30TFLOPs/s of compute throughput and 768GB/s of global memory bandwidth.

Ideally, 4.5ms for calculation and 0.34ms for memory.

compute-bound

Matmul - lowerbound

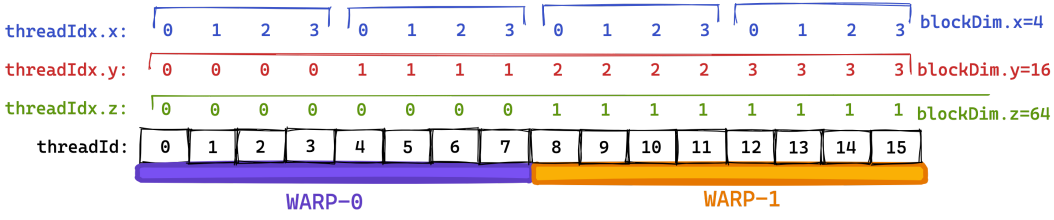
We achieves ~300GFLOPs when multiplying two 4092×4092 fp32 matrices.
Upperbound ~30TFLOPs.

So how can we start to make this faster?

One way is to optimize the memory access pattern of our kernel such that global memory accesses can be coalesced (=combined) into fewer accesses.

Global Memory Coalescing

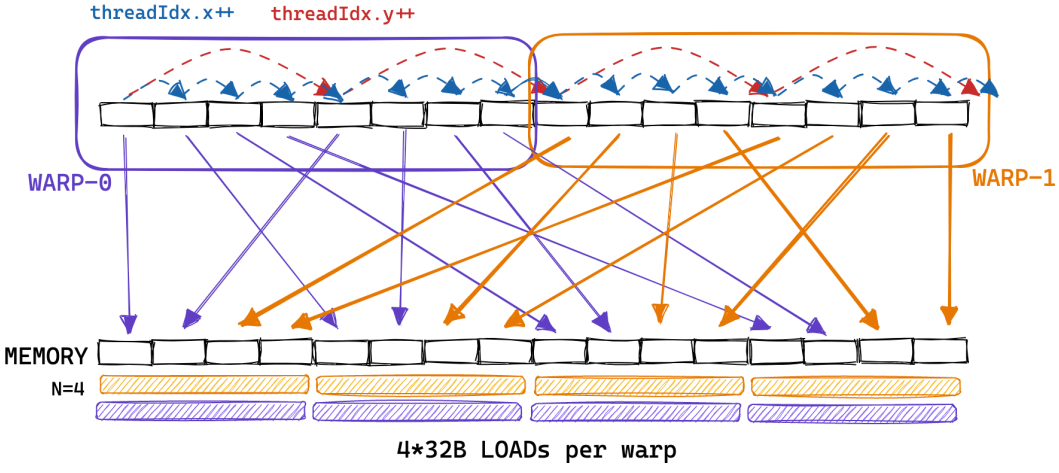
The threads of a block are grouped into so-called warps, consisting of 32 threads. A warp is then assigned to a warp scheduler, which is the physical core that executes the instructions. There are four warp schedulers per multiprocessor. The grouping into warps happens based on a consecutive threadId.




```
threadId = threadIdx.x + blockDim.x*threadIdx.y + blockDim.x*blockDim.y*threadIdx.z
```

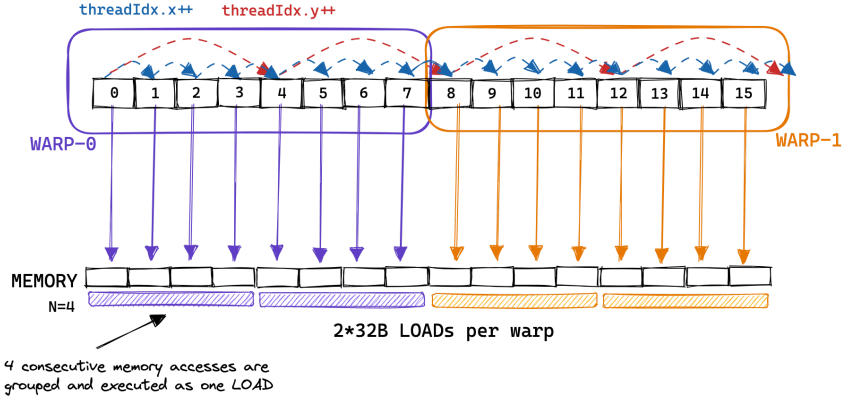
Global Memory Coalescing

Threads of the same warp (those with consecutive threadIdx.x) were loading the rows of A non-consecutively from memory. The naive kernel's pattern of accessing the memory of A looked more like so:



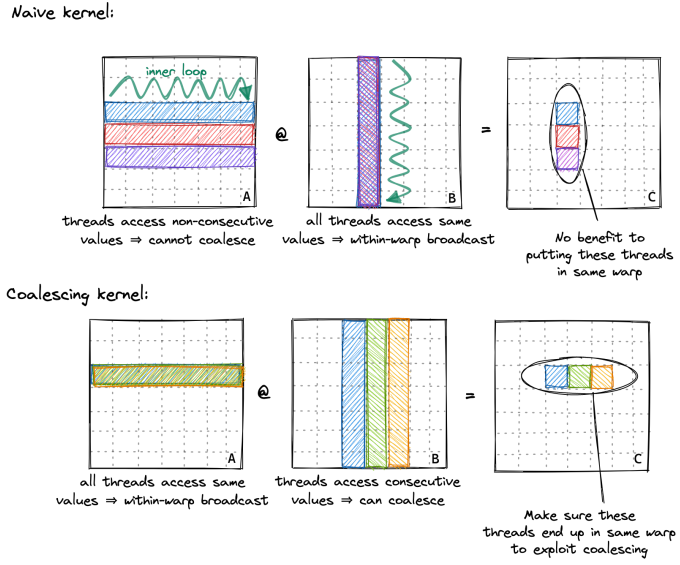
Global Memory Coalescing

The concept of a warp is relevant for this second kernel, as sequential memory accesses by threads that are part of the same warp can be grouped and executed as one. This is referred to as **global memory coalescing**. It's the most important thing to keep in mind when optimizing a kernel's GMEM memory accesses toward achieving the peak bandwidth.



Global Memory Coalescing

To enable coalescing, we can change how we assign positions of the result matrix C to threads. This change in the global memory access pattern is illustrated below:



Global Memory Coalescing

To implement coalescing kernel:

```
template <const uint BLOCKSIZE>
__global__ void sgemm_global_mem_coalesce(int M, int N, int K, float alpha,
                                          const float *A, const float *B,
                                          float beta, float *C) {
  const int cRow = blockIdx.x * BLOCKSIZE + (threadIdx.x / BLOCKSIZE);
  const int cCol = blockIdx.y * BLOCKSIZE + (threadIdx.x % BLOCKSIZE);

  // if statement is necessary to make things work under tile quantization
  if (cRow < M && cCol < N) {
    float tmp = 0.0;
    for (int i = 0; i < K; ++i) {
      tmp += A[cRow * K + i] * B[i * N + cCol];
    }
    C[cRow * N + cCol] = alpha * tmp + beta * C[cRow * N + cCol];
  }
}
```

Global memory coalescing increases memory throughput from 15GB/s to 110GB/s. Performance reaches 2000 GFLOPS, a big improvement compared to the 300 GFLOPS of the first, naive kernel.

Shared Memory Cache-Blocking

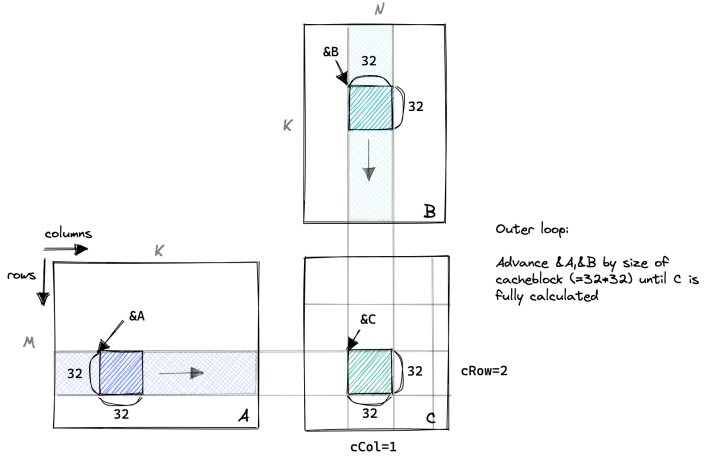
Next to the large global memory, a GPU has a much smaller region of memory that is physically located on the chip, called shared memory (SMEM).

Physically, there's one shared memory per SM. Logically, this shared memory is partitioned among the blocks. This means that a thread can communicate with the other threads in its block via the shared memory chunk.

As the shared memory is located on-chip, it has a much lower latency and higher bandwidth than global memory.

Shared Memory Cache-Blocking

So for this kernel, we'll load a chunk of A and a chunk of B from global memory into shared memory. Then we'll perform as much work as possible on the two chunks, with each thread still being assigned one entry of C. We'll move the chunks along the columns of A and the rows of B performing partial sums on C until the result is computed.

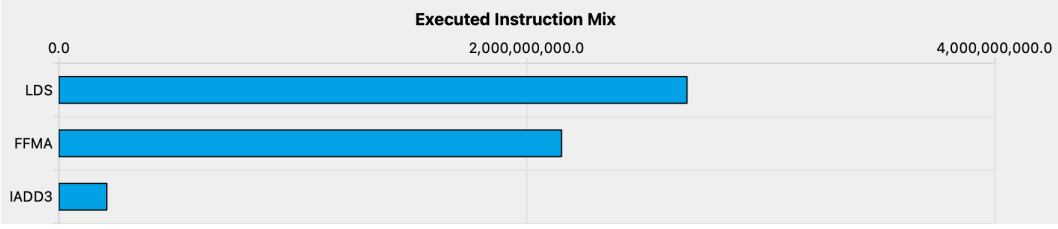


Shared Memory Cache-Blocking

```
// advance pointers to the starting positions
A += cRow * BLOCKSIZE * K;                    // row=cRow, col=0
B += cCol * BLOCKSIZE;                        // row=0, col=cCol
C += cRow * BLOCKSIZE * N + cCol * BLOCKSIZE; // row=cRow, col=cCol
float tmp = 0.0;
for (int bkIdx = 0; bkIdx < K; bkIdx += BLOCKSIZE) {
  // Have each thread load one of the elements in A & B
  // Make the threadCol (=threadIdx.x) the consecutive index
  // to allow global memory access coalescing
  As[threadRow * BLOCKSIZE + threadCol] = A[threadRow * K + threadCol];
  Bs[threadRow * BLOCKSIZE + threadCol] = B[threadRow * N + threadCol];
  // block threads in this block until cache is fully populated
  __syncthreads();
  A += BLOCKSIZE;
  B += BLOCKSIZE * N;
  // execute the dotproduct on the currently cached block
  for (int dotIdx = 0; dotIdx < BLOCKSIZE; ++dotIdx) {
    tmp += As[threadRow * BLOCKSIZE + dotIdx] *
           Bs[dotIdx * BLOCKSIZE + threadCol];
  }
  // need to sync again at the end, to avoid faster threads
  // fetching the next block into the cache before slower threads are done
  __syncthreads();
}
C[threadRow * N + threadCol] =
    alpha * tmp + beta * C[threadRow * N + threadCol];
```

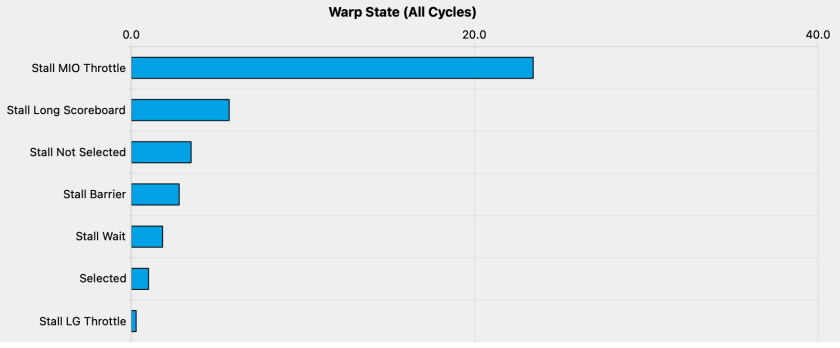
This kernel achieves ~2200 GFLOPS, a 50% improvement over the previous version. We're still far away from hitting the ~30 TFLOPs that the GPU can provide.

If we look at the mix of executed instructions, most of them are memory loads:



That's not good, given that a memory load is bound to have a higher latency than a simple FMA, and given that we know our kernel should be compute bound.

We see this effect when looking at the profiler's sampling of warp states. This quantifies how many cycles were spent in each state per executed instruction:



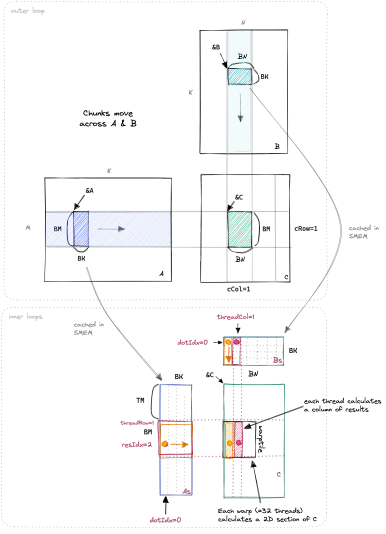
Stall MIO Throttle

Warp was stalled waiting for the MIO (memory input/output) instruction queue to be not full. This stall reason is high in cases of extreme utilization of MIO pipelines (shared memory instructions).

So how can we make our kernel issue less SMEM instructions? One way is to have each thread compute more than one output element, which allows us to perform more of work in registers and relying less on SMEM.

1D Blocktiling

This kernel works like last kernel, but adds a new inner loop, for calculating multiple C entries per thread.



1D Blocktiling

The major part of the implementation of this kernel:

```
// allocate thread-local cache for results in registerfile
float threadResults[TM] = {0.0};

// outer loop over block tiles
for (uint bkIdx = 0; bkIdx < K; bkIdx += BK) {
  // populate the SMEM caches
  As[innerRowA * BK + innerColA] = A[innerRowA * K + innerColA];
  Bs[innerRowB * BN + innerColB] = B[innerRowB * N + innerColB];
  __syncthreads();

  // advance blocktile
  A += BK;
  B += BK * N;

  // calculate per-thread results
  for (uint dotIdx = 0; dotIdx < BK; ++dotIdx) {
    // we make the dotproduct loop the outside loop, which facilitates
    // reuse of the Bs entry, which we can cache in a tmp var.
    float tmpB = Bs[dotIdx * BN + threadCol];
    for (uint resIdx = 0; resIdx < TM; ++resIdx) {
      threadResults[resIdx] +=
          As[(threadRow * TM + resIdx) * BK + dotIdx] * tmpB;
    }
  }
  __syncthreads();
}

// write out the results
for (uint resIdx = 0; resIdx < TM; ++resIdx) {
  C[(threadRow * TM + resIdx) * N + threadCol] =
      alpha * threadResults[resIdx] +
      beta * C[(threadRow * TM + resIdx) * N + threadCol];
}
```

This kernel achieves ∼8600 GFLOPs, 2.2× faster than our previous kernel.

1D Blocktiling

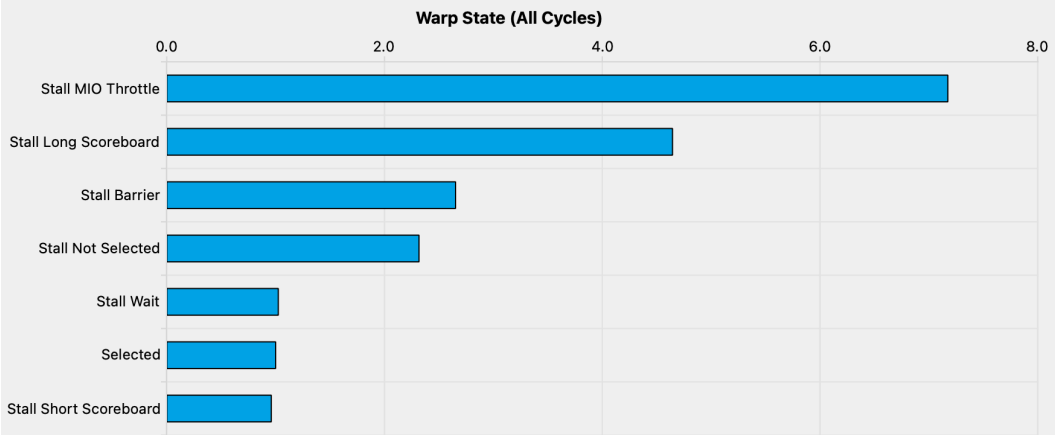
Let's calculate how many memory accesses each thread performed in our previous kernel, where each thread calculated one result:

- GMEM: K/32 iterations of outer loop * 2 loads
- SMEM: K/32 iterations of outer loop * BLOCKSIZE (=32) * 2 loads
- Memory accesses per result: K/16 GMEM, K*2 SMEM

And for our new kernel, where each thread calculates eight results:

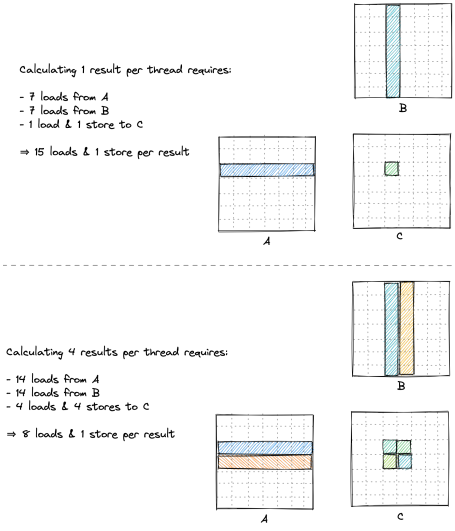
- GMEM: K/8 iterations of outer loop * 2 loads
- SMEM: K/8 iterations of outer loop * BK(=8) * (1 + TM(=8))
- Memory accesses per result: K/32 GMEM, K*9/8 SMEM

As expected, we now spend much fewer cycles per instruction stalling due to memory pressure:



Our current kernel still suffers from the same stalling-for-memory problem as later kernel, just to a lesser extent. So we'll just apply the same optimization again: computing even more results per thread. The main reason this makes our kernel run faster is that it increases arithmetic intensity.

1D Blocktiling



In conclusion, all our kernels perform the same number of FLOPs, but we can reduce the number of GMEM accesses by calculating more results per thread.

2D Blocktiling

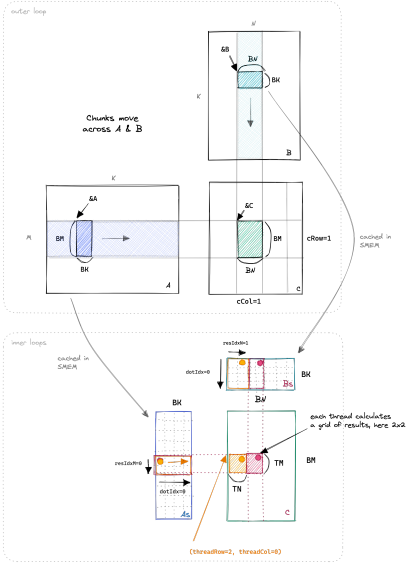
The basic idea for this kernel will be to compute a grid of 8*8 elements of C per thread. The first stage of the kernel is for all threads to work together to populate the SMEM cache. We'll have each thread load multiple elements. This code looks like so:

```
// populate the SMEM caches
  for (uint loadOffset = 0; loadOffset < BM; loadOffset += strideA) {
    As[(innerRowA + loadOffset) * BK + innerColA] =
        A[(innerRowA + loadOffset) * K + innerColA];
  }
  for (uint loadOffset = 0; loadOffset < BK; loadOffset += strideB) {
    Bs[(innerRowB + loadOffset) * BN + innerColB] =
        B[(innerRowB + loadOffset) * N + innerColB];
  }
  __syncthreads();
```

Now that the SMEM cache is populated, we have each thread multiply its relevant SMEM entries and accumulate the result into local registers.

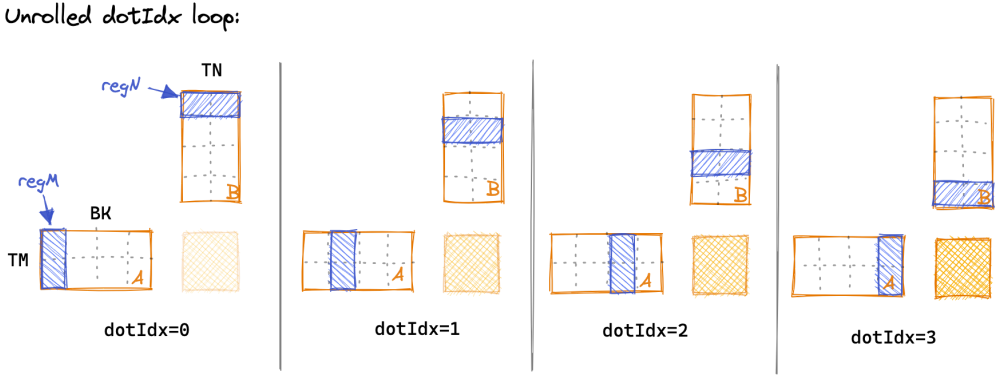
2D Blocktiling

Below I illustrated the (unchanged) outer loop along the input matrices, and the three inner loops for the dot product and the TN and TM dimension:



2D Blocktiling

In the inner loop, we can reduce the number of SMEM accesses by making dotIdx the outer loop, and explicitly loading the values we need for the two inner loops into registers. Below is a drawing of the dotIdx loop across time, to visualize which SMEM entries get loaded into thread-local registers at each step:



2D Blocktiling

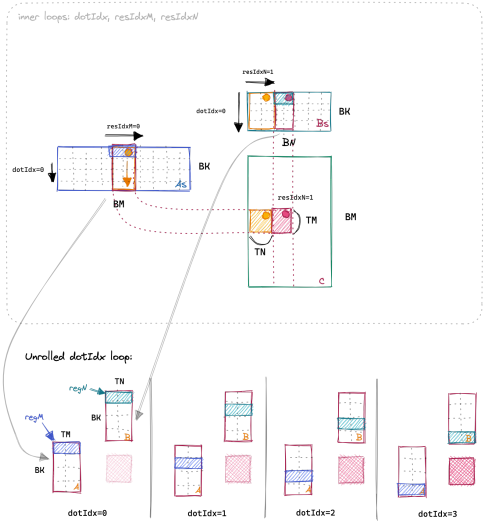
Resulting performance: 16TFLOPs, another 2× improvement. Let's repeat the memory access calculation. We're now calculating TM×TN = 8×8 = 64 results per thread.

- GMEM: K/8 (outer loop iters) × 2 (A+B) × 1024/256 (sizeSMEM/numThreads) loads
- SMEM: K/8 (outer loop iters) × 8 (dotIdx) × 2 (A+B) × 8 loads
- Memory accesses per result: K/64 GMEM, K/4 SMEM

Slowly performance is reaching acceptable levels, however, warp stalls due to memory pipeline congestion are still too frequent. For the next kernel we'll take two measures to try to improve that: Transposing As to enable auto-vectorization of SMEM loads, and promising the compiler alignment on the GMEM accesses.

Vectorize SMEM and GMEM Accesses

The first optimization that mentioned earlier is to transpose As. This will allow us to load from As using vectorized SMEM loads (LDS.128 in SASS):



Vectorize SMEM and GMEM Accesses

Looking at the assembly we see that loading As into the registers, which used to be a 32b LDS load, is now also a 128b LDS.128 load, just like it had already been for Bs. This gives us a 500GFLOPs speedup, or ∼3%.

Next, we'll vectorize all loads and stores from/to GMEM using vectordatatypes, namely float4.

```
float4 tmp =
      reinterpret_cast<float4 *>(&A[innerRowA * K + innerColA * 4])[0];
  As[(innerColA * 4 + 0) * BM + innerRowA] = tmp.x;
  As[(innerColA * 4 + 1) * BM + innerRowA] = tmp.y;
  As[(innerColA * 4 + 2) * BM + innerRowA] = tmp.z;
  As[(innerColA * 4 + 3) * BM + innerRowA] = tmp.w;

  reinterpret_cast<float4 *>(&Bs[innerRowB * BN + innerColB * 4])[0] =
      reinterpret_cast<float4 *>(&B[innerRowB * N + innerColB * 4])[0];
  __syncthreads();
```

This kernel achieves 19TFLOPs. The profiler still shows a bunch of problem areas and optimization opportunities: We're running into shared-memory bank conflicts (which cuBLAS avoids), our occupancy is higher than necessary, and we haven't implemented any double buffering.

Autotuning

We've accumulated a total of five template parameters:

- BM, BN and BK, which specify how much data we cache from GMEM into SMEM.
- TM and TN, which specify how much data we cache from SMEM into the registers.
- The kernel implementation was correct for the ∼400 different hyperparameter settings that remained.

It turns out that the optimal parameters vary quite a bit depending on the GPU model.

- On A6000, BM=BN=128, BK=16, TM=TN=8 increased performance by 5%, from 19 to 20 TFLOPs.
- On A100 SMX4 40GB, that same configuration reached 12 TFLOPs, 6% worse than the optimal setting found by the autotuner (BM=BN=64, BK=16, TM=TN=4), which reached 12.6 TFLOPs.

Warptiling

We'll now add another hierarchy of tiling, in between our blocktiling and threadtiling loops: warptiling. Warptiling is somewhat confusing initially since unlike blocks and threads, warps don't show up anywhere in the CUDA code explicitly. They are a hardware feature that has no direct analog in the scalar CUDA-software world.

We can calculate a given thread's warpId as warpId=threadIdx.x % warpSize, where warpSize is a built-in variable that is equal to 32.

Warptiling

Warps are relevant for performance since (among other reasons):

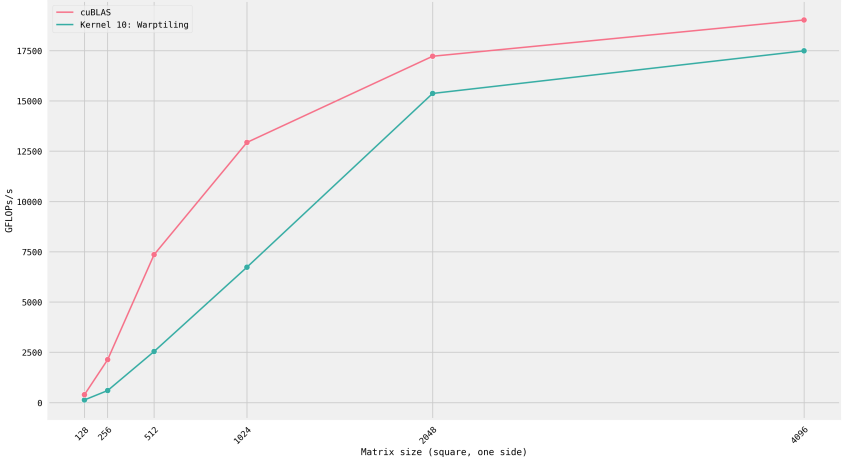
- Warps are the unit of scheduling that is mapped to the warp-schedulers that are part of the SM.
- Shared-memory bank conflicts (I'll cover those in a future post) happen only between threads that are in the same warp.
- There's a register cache on recent GPUs, and tighter threadtiling gives us more register cache locality.

Warptiling is elegant since we now make explicit all levels of parallelism:

- Blocktiling: Different blocks can execute in parallel on different SMs.
- Warptiling: Different warps can execute in parallel on different warp schedulers, and concurrently on the same warp scheduler.
- Threadtiling: (a very limited amount of) instructions can execute in parallel on the same CUDA cores (= instruction-level parallelism aka ILP)

Warptiling

After autotuning the parameters, performance improves from 19.7 TFLOPs to 21.7 TFLOPs on an A100. Here's a plot that compares our warptiling kernel against cuBLAS across increasing matrix sizes:



Warptiling

At dimensions 2048 and 4096, our measured FLOPs are only a few percentage points slower than cuBLAS. However, for smaller matrices, we're doing poorly in comparison to Nvidia's library!

This happens because cuBLAS contains not one single implementation of SGEMM, but hundreds of them. At runtime, based on the dimensions, cuBLAS will pick which kernel to run.

Matrix size	Name	Duration
128	`ampere_sgemm_32x32_sliced1x4_nn`	15.295 µs
256	`ampere_sgemm_64x32_sliced1x4_nn` *followed by* `splitKreduce_kernel`	12.416 µs + 6.912 µs
512	`ampere_sgemm_32x32_sliced1x4_nn`	41.728 µs
1024	`ampere_sgemm_128x64_nn`	165.953 µs
2048	`ampere_sgemm_128x64_nn`	1.247 ms
4096	`ampere_sgemm_128x64_nn`	9.290 ms

At dimension 256 it calls two kernels: a matmul kernel followed by a reduction kernel. So if we were trying to write a high- performance library that works for all shapes and sizes we would have specializations for different shapes, and at runtime dispatch to the one that's the best fit.