GPGPU Programming

Donato D'Ambrosio

Department of Mathematics and Computer Science
Cubo 30B, University of Calabria, Rende 87036, Italy
mailto: donato.dambrosio@unical.it
homepage: http://www.mat.unical.it/~donato

Academic Year 2020/21

Table of contents

Performance Considerations

Performance Considerations

Performance Considerations

Performance Considerations

- Limited hardware resources, such as computing power and memory, can become *bottlenecks* if the demands for these resources are too high. An application can be:
  - compute-bound: if the bottleneck is the computing power;
  - memory-bound: if the bottleneck is the memory.
- Often, performance can be greatly improved by finding a better trade-off between computational and memory resources.
- Often, the application must be re-designed.

Global Memory Bandwidth

- Here, we will further discuss memory coalescing techniques that can more effectively move data from the global memory into shared memories and registers.
- Accessing global memory (DRAM) requires 10s of nanoseconds, which is much more than the SMs' clock speed.
- Nevertheless, each time a location is accessed, a range of consecutive locations, aka bursts (including the requested location) are also accessed in a very fast way.
- Accessing bursts results in higher transfer rate than accessing locations in a random order.

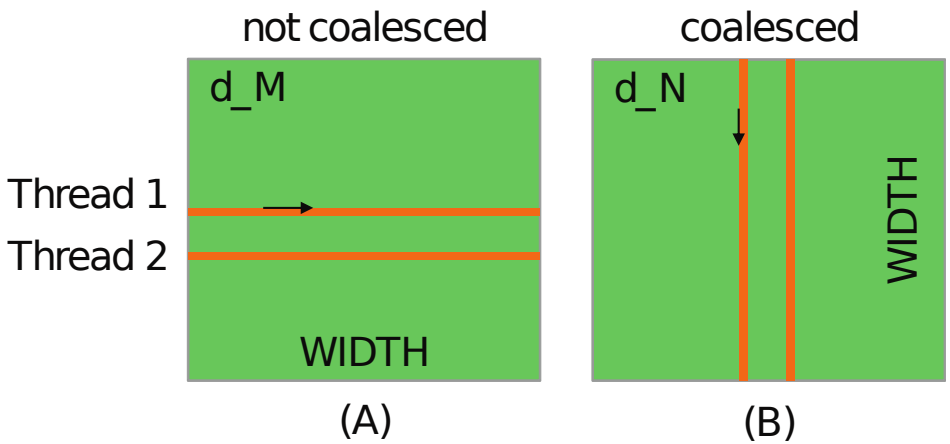
Global Memory Bandwidth

- When all threads in a warp execute a load instruction (that is executed simultaneously), the hardware detects whether they access consecutive global memory locations. As a consequence, the most favorable access pattern is achieved when all threads in a warp access consecutive locations.
- In this case, the hardware combines, or *coalesces*, all these accesses into a single (or more if needed) consolidated access request(s) to consecutive DRAM locations.
- WARNING: Data coalescence can be counter-intuitive if we think in terms of sequential execution instead of parallel execution of threads inside warps.

Global Memory Bandwidth

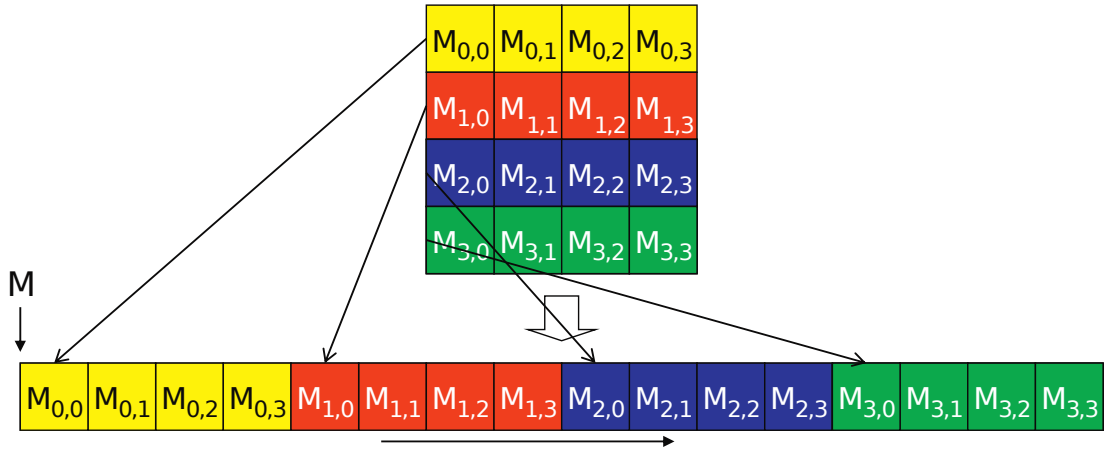
- Let us consider the non-optimized matrix multiplication example:

```
__global__ void MatrixMulKernel(float* M, float* N, float* P,
int Width) {
  int Row = blockIdx.y*blockDim.y+threadIdx.y;
  int Col = blockIdx.x*blockDim.x+threadIdx.x;
  if ((Row < Width) && (Col < Width)) {
    float Pvalue = 0;
    for (int k = 0; k < Width; ++k) {
      Pvalue += M[Row*Width+k] * N[k*Width+Col];
    }
    P[Row*Width+Col] = Pvalue;
  }
}
```



Global Memory Bandwidth

- Since matrices are stored in row-major order...



  ...the coalesced accesses occur in the case B, where a column of N is read (this may be counter-intuitive at a first sight).

- The reason is because the SM process the threads in a warp in parallel, in lockstep.
- As a consequence, at each execution step, `k` is the same for all the threads (since they execute the same instruction), while `Row` and `Col` vary depending on blocks and threads indices.

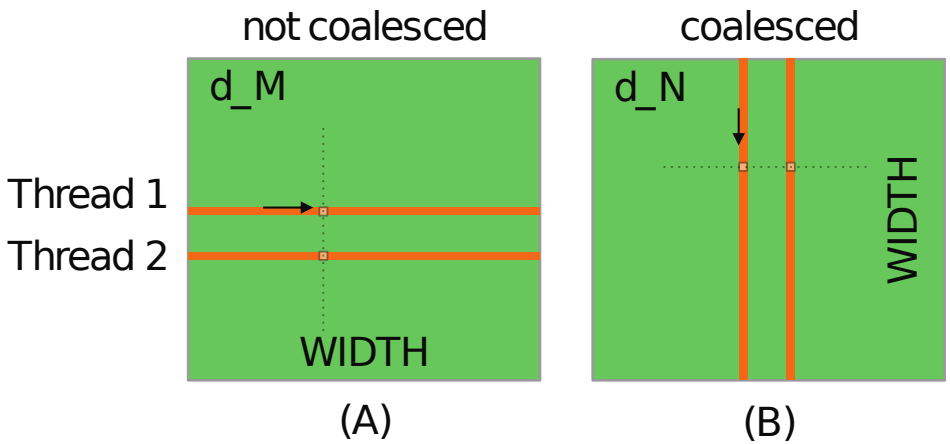
Global Memory Bandwidth

- In fact, If we look at the code with more awareness

```
int Row = blockIdx.y*blockDim.y+threadIdx.y;
int Col = blockIdx.x*blockDim.x+threadIdx.x;
// ...
  for (int k = 0; k < Width; ++k)
    Pvalue += M[Row*Width+k] * N[k*Width+Col];
```

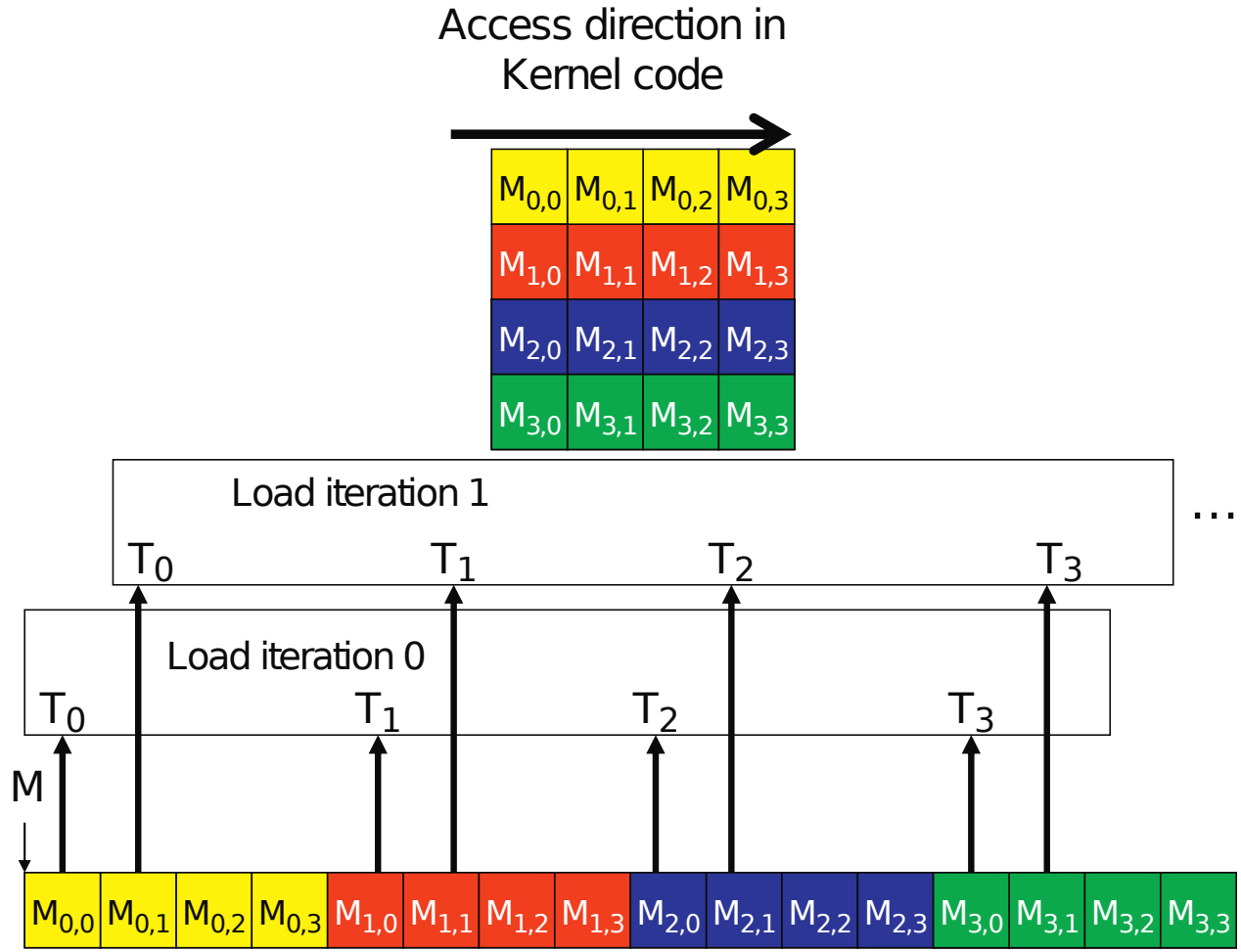
we notice that:

- `Row*Width+k` results in non-adjacent locations, since is `Row` that varies, each thread thus accessing an element on a different row.
- `k*Width+Col` results in adjacent accesses, since `Col` varies.



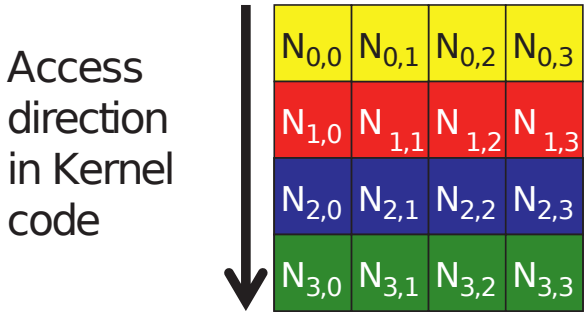
Global Memory Bandwidth

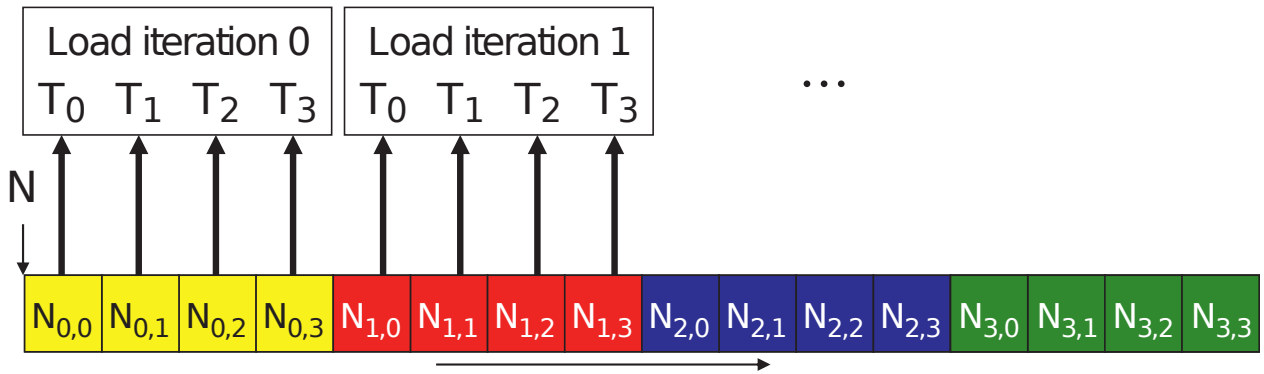
- If we assume 4x4 blocks and the warp size is 4, the `Row*Width+k` index of `M` results in the following access pattern:



Global Memory Bandwidth

- Conversely, the access pattern for the `k*Width+Col` index of `N` results in:

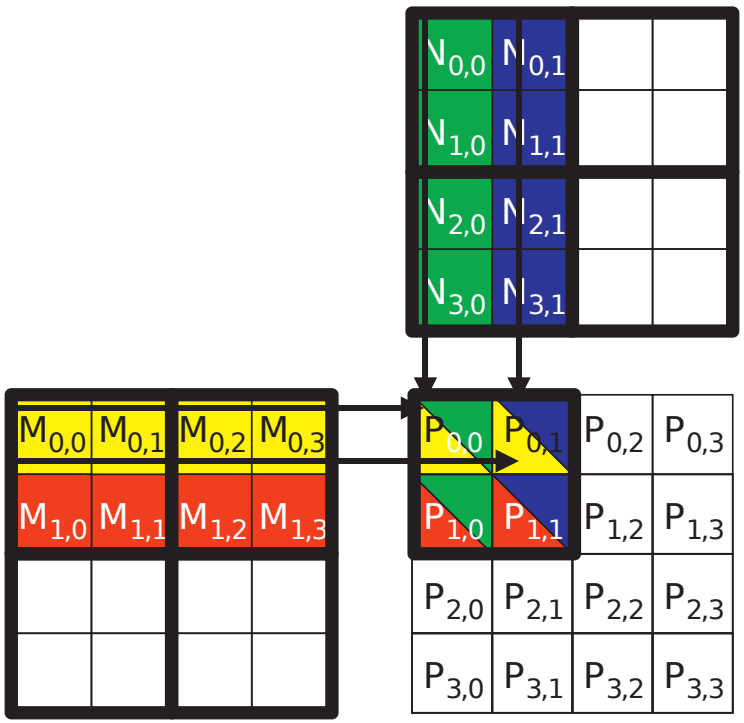




- All these accesses are coalesced into consolidated accesses for improved DRAM bandwidth utilization.

Global Memory Bandwidth

- If an algorithm requires to iterate along the row direction (resulting in non-coalesced accesses), shared memory can be used that does not require coalescing to achieve high data access rate.
- Nevertheless, coalesced access should be exploited in the copy phase, as done in the tiled matrix multiplication algorithm.



	Phase 1			Phase 2		
$thread_{0,0}$	**$M_{0,0}$** ↓ $Mds_{0,0}$	**$N_{0,0}$** ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{0,1}$*$Nds_{1,0}$	**$M_{0,2}$** ↓ $Mds_{0,0}$	**$N_{2,0}$** ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{0,1}$*$Nds_{1,0}$
$thread_{0,1}$	**$M_{0,1}$** ↓ $Mds_{0,1}$	**$N_{0,1}$** ↓ $Nds_{1,0}$	$PValue_{0,1}$ += $Mds_{0,0}$*$Nds_{0,1}$ + $Mds_{0,1}$*$Nds_{1,1}$	**$M_{0,3}$** ↓ $Mds_{0,1}$	**$N_{2,1}$** ↓ $Nds_{0,1}$	$PValue_{0,1}$ += $Mds_{0,0}$*$Nds_{0,1}$ + $Mds_{0,1}$*$Nds_{1,1}$
$thread_{1,0}$	**$M_{1,0}$** ↓ $Mds_{1,0}$	**$N_{1,0}$** ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{1,0}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{1,0}$	**$M_{1,2}$** ↓ $Mds_{1,0}$	**$N_{3,0}$** ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{1,0}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{1,0}$
$thread_{1,1}$	**$M_{1,1}$** ↓ $Mds_{1,1}$	**$N_{1,1}$** ↓ $Nds_{1,1}$	$PValue_{1,1}$ += $Mds_{1,0}$*$Nds_{0,1}$ + $Mds_{1,1}$*$Nds_{1,1}$	**$M_{1,3}$** ↓ $Mds_{1,1}$	**$N_{3,1}$** ↓ $Nds_{1,1}$	$PValue_{1,1}$+= $Mds_{1,0}$*$Nds_{0,1}$ + $Mds_{1,1}$*$Nds_{1,1}$

time

Global Memory Bandwidth

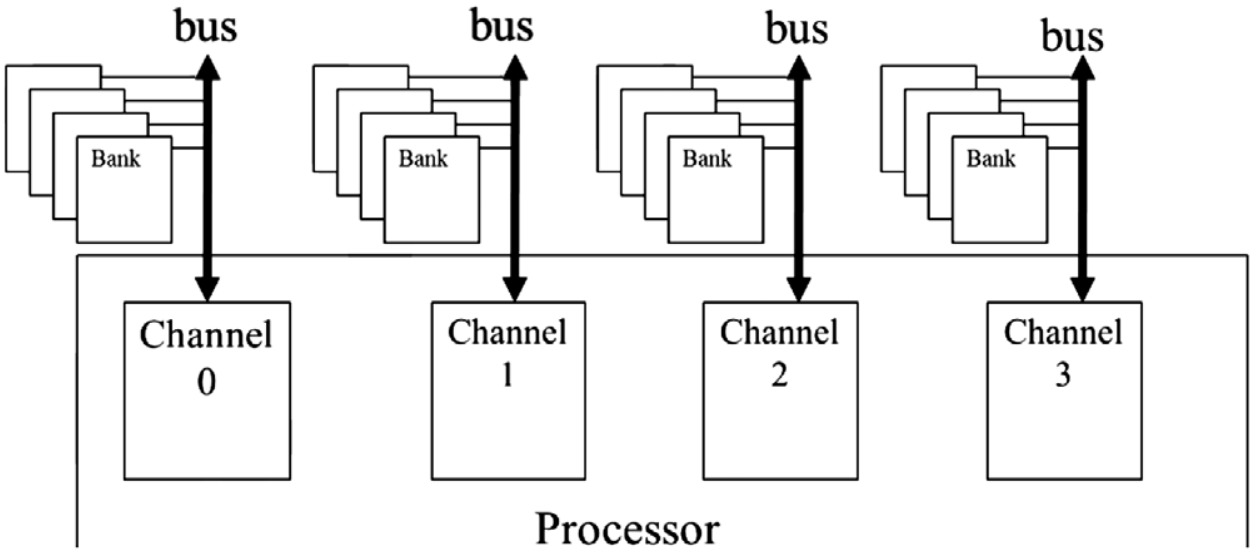
```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width
    ) {
  __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
  __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

  int bx = blockIdx.x;  int by = blockIdx.y;
  int tx = threadIdx.x; int ty = threadIdx.y;
  int Row = by * TILE_WIDTH + ty;
  int Col = bx * TILE_WIDTH + tx;

  float Pvalue = 0;
  for (int ph = 0; ph < Width/TILE_WIDTH; ++ph) {
    Mds[ty][tx] = M[Row*Width + ph*TILE_WIDTH + tx];    //coalesced
    Nds[ty][tx] = N[(ph*TILE_WIDTH + ty)*Width + Col]; //coalesced
    __syncthreads();
    for (int k = 0; k < Width; ++k)
      Pvalue += Mds[ty][k] * Nds[k][tx];
    __syncthreads();
  }
  P[Row*Width+Col] = Pvalue;
}
```

More on Memory Parallelism

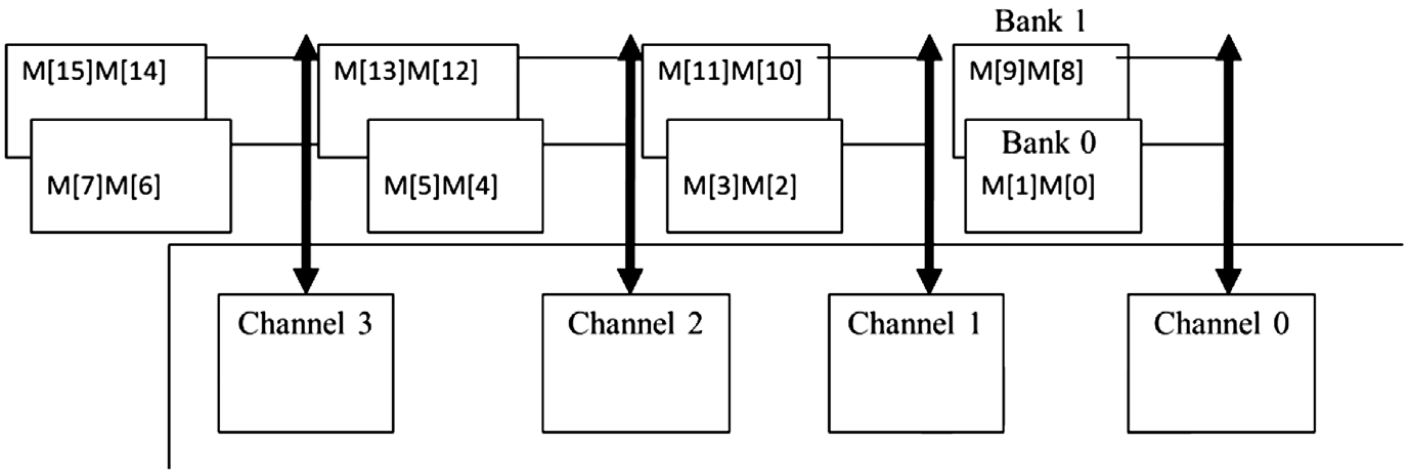
- Bursting alone is not sufficient to realize the level of DRAM access bandwidth required by modern processors.
- A processor contains one or more memory controllers, called channels, with a bus that connects a set of DRAM banks to the processor.



- In real systems, a processor typically has one to eight channels and each channel is connected to a large number of banks.

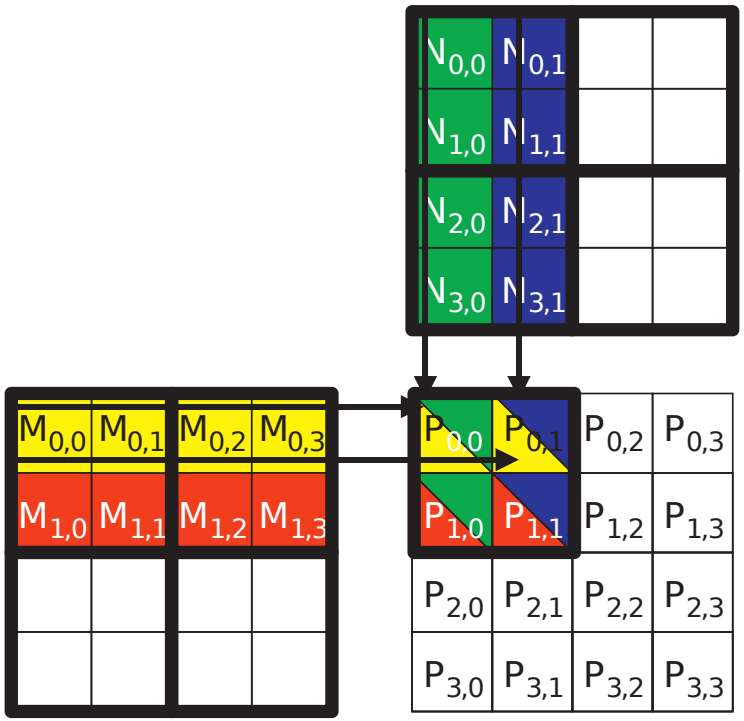
More on Memory Parallelism

- There is an important connection between the parallel execution of threads and the parallel organization of the DRAM system.
- In order to achieve the optimal memory bandwidth, there must be a sufficient number of threads making evenly distributed simultaneous memory accesses.
- In fact, data is stored in memory with an *interleaved* pattern, as in this example, where we assumed burst size of two elements (eight bytes).



More on Memory Parallelism

- Consider the tiled matrix multiplication algorithm' access pattern.

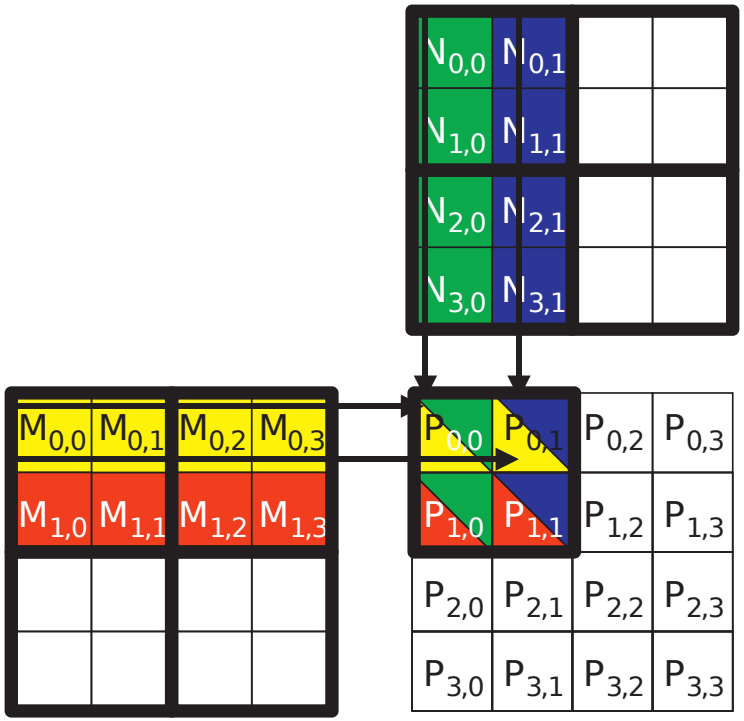


Tiles loaded by	Block 0,0	Block 0,1	Block 1,0	Block 1,1
Phase 0 (2D index)	M[0][0],M[0][1], M[1][0], M[1][1]	M[0][0], M[0][1], M[1][0], M[1][1]	M[2][0], M[2][1], M[3][0], M[3][1]	M[2][0], M[2][1], M[3][0], M[3][1]
Phase 0 (linearized index)	M[0], M[1], M[4], M[5]	M[0], M[1], M[4], M[5]	M[8], M[9], M[12], M[13]	M[8], M[9], M[12], M[13]
Phase 1 (2D index)	M[0][2],M[0][3], M[1][2], M[1][3]	M[0][2],M[0][3], M[1][2], M[1][3]	M[2][2], M[2][3], M[3][2], M[3][3]	M[2][2], M[2][3], M[3][2], M[3][3]
Phase 1 (linearized index)	M[2], M[3], M[6], M[7]	M[2], M[3], M[6], M[7]	M[10], M[11], M[14], M[15]	M[10], M[11], M[14], M[15]

- Note that each block makes two coalesced accesses to two banks using two different channels (e.g., 0 and 2 for Block0,0). In this manner, the algorithm maximizes the data transfer bandwidth of each channel.

More on Memory Parallelism

- Note that Block0,0 and Block0,1 load the same elements of M. Most devices have caches that combine these accesses into one if the execution timing of these blocks are sufficiently close.



Tiles loaded by	Block 0,0	Block 0,1	Block 1,0	Block 1,1
Phase 0 (2D index)	M[0][0],M[0][1], M[1][0], M[1][1]	M[0][0], M[0][1], M[1][0], M[1][1]	M[2][0], M[2][1], M[3][0], M[3][1]	M[2][0], M[2][1], M[3][0], M[3][1]
Phase 0 (linearized index)	M[0], M[1], M[4], M[5]	M[0], M[1], M[4], M[5]	M[8], M[9], M[12], M[13]	M[8], M[9], M[12], M[13]
Phase 1 (2D index)	M[0][2],M[0][3], M[1][2], M[1][3]	M[0][2],M[0][3], M[1][2], M[1][3]	M[2][2], M[2][3], M[3][2], M[3][3]	M[2][2], M[2][3], M[3][2], M[3][3]
Phase 1 (linearized index)	M[2], M[3], M[6], M[7]	M[2], M[3], M[6], M[7]	M[10], M[11], M[14], M[15]	M[10], M[11], M[14], M[15]

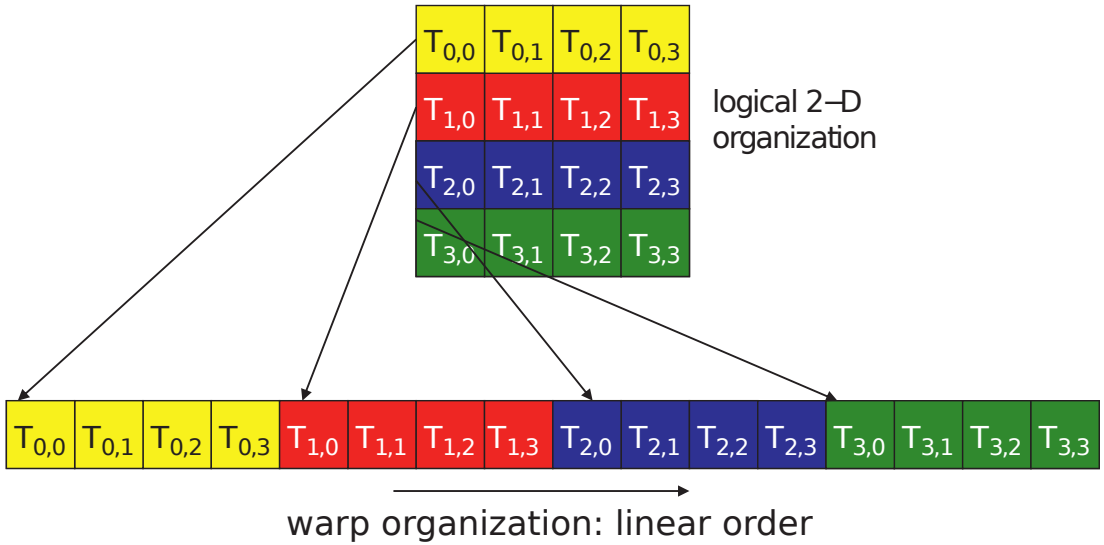
- Algorithms accessing data in parallel better exploit the memory bandwidth and maximize performance.

Warps and SIMD Hardware

- We now turn our attention to aspects of the thread execution that can limit performanance.
- Launching a CUDA kernel generates a grid of threads that is organized as a two-level hierarchy:
  1. A grid of a one-, two-, or three-dimensional array of blocks.
  2. Each block, in turn, consists of a one-, two-, or three-dimensional array of threads.
- Bocks can execute in any order relative to each other, which allows for transparent scalability across different devices.
- Threads within a block are partitioned for execution in warps. All CUDA devices use warps consisting of 32 threads. However, this value could change in the future.

Warps and SIMD Hardware

- Thread blocks are partitioned into warps based on thread indices.
  1. One-dimensional: in order, according to `threadIdx.x`;
  2. Two-dimensional: linearized in row-majour order with `threadIdx.y` as top-level indix;



- Three-dimensional: linearized in slice-majour and then row-majour order, thus with `threadIdx.z` as top-level index.
- If a block size is not a multiple of 32 in one or more dimensions, the last warp will be padded with extra threads to fill up the 32-thread positions.

Warps and SIMD Hardware

- The SIMD hardware executes all threads of a warp as a bundle in lockstep.
- The lockstep execution works well when all the threads follow the same execution path (aka control flow).
- For example, for an `if-else`, the execution works well when either all threads execute the `if` part or all execute the `else` part.
  - When threads within a warp take different control flow paths, the SIMD hardware will take multiple passes through these divergent paths. One pass executes those threads that follow the if part and another pass executes those that follow the else part.
  - During each pass, the threads that follow the other path are not allowed to take effect. These passes are sequential to each other, thus will add to the execution time.
- Divergence can arise in other constructs. For example, if threads execute a `for` loop which a variable number of steps per thread (e.g., six, seven, or eight), the threads that do not need to further iterate must wait for the others!

Warps and SIMD Hardware

- Let us consider the vector addition kernel to discuss the impact of control divergence:

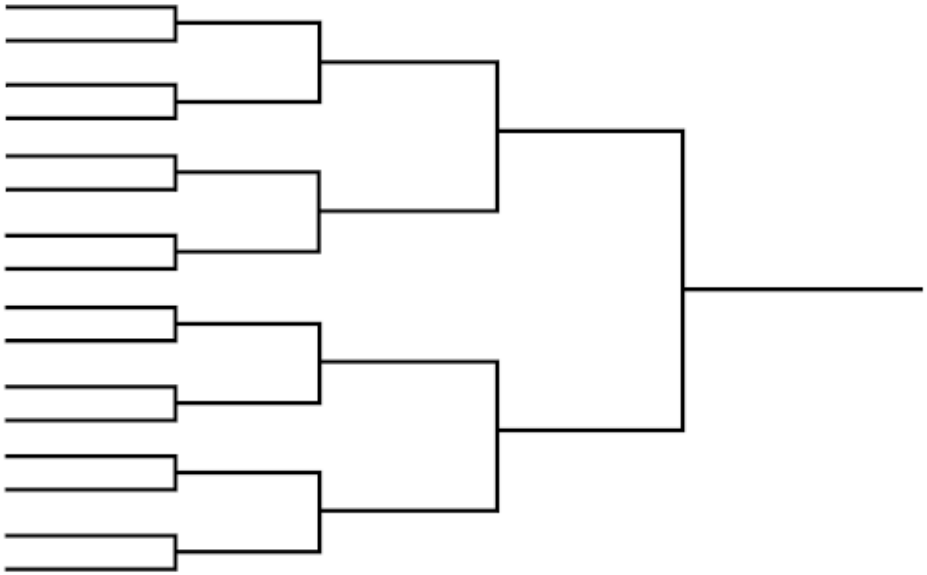
```
__global__ void vecAddKernel(float* A, float* B, float* C, int n)
{
  int i = blockDim.x*blockIdx.x + threadIdx.x;

  if (i < n) // to be sure to "intercept" data
    C[i] = A[i] + B[i];
}
```

- The impact of control divergence depends on the size of the vectors being processed.
  - If `n==100`, 1 of the 4 warps will have control divergence, which can have significant impact on performance.
  - If `n=1000`, only 1 out of the 32 warps (about the 3% of the total) will have control divergence.
  - If `n=10,000`, only 1 of the 313 warps will have control divergence!
- Divergence also arise in those algorithms with a variable number of threads per computational step...

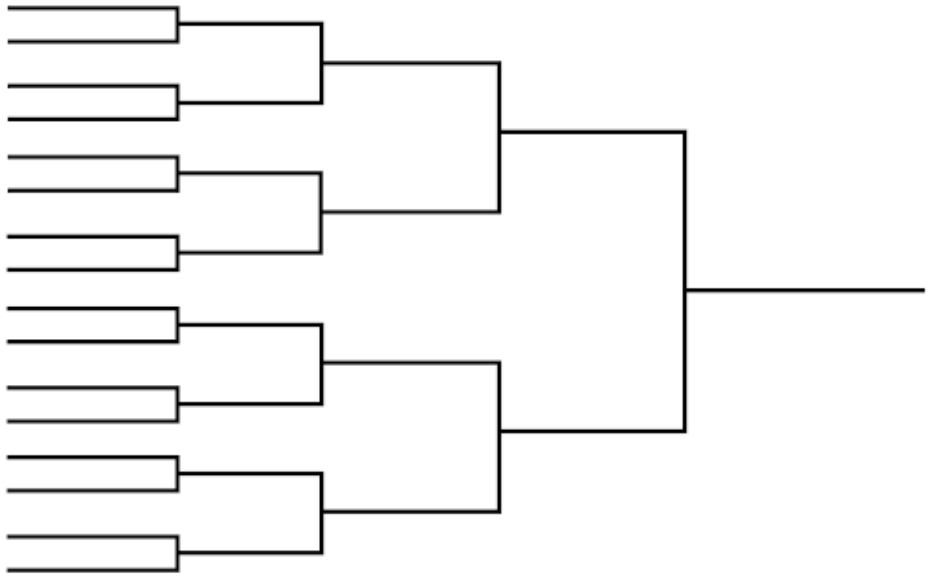
Warps and SIMD Hardware

- Let us consider a reduction algorithm, which typically resembles the structure of a tournament. The tournament reduction is done in multiple rounds:
  - The teams are divided into pairs.
  - During the first round, all pairs play in parallel.
  - Winners of the first round advance to the second round, whose winners advance to the third round, etc.



Warps and SIMD Hardware

- With 16 teams, 8 winners will emerge from the first round, 4 from the second round, 2 from the third round, and the winner from the fourth round.

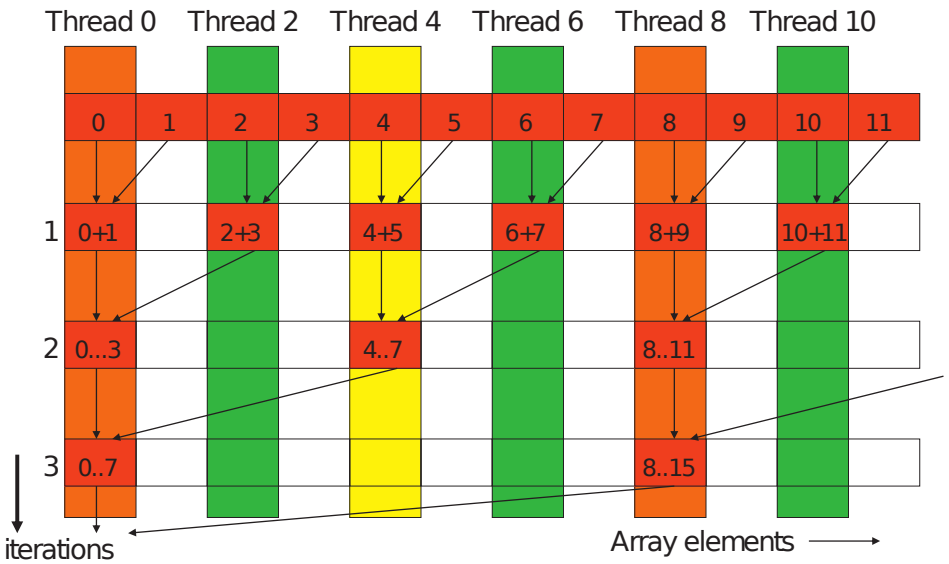


- With 1024 teams, it takes only 10 rounds to determine the winner, under the condition to have enough courts to hold the 512 games in parallel during the first round, 256 games in the second round, 128 games in the third round, and so on.

Warps and SIMD Hardware

- Here is a kernel function that performs parallel sum reduction, where `X` is the array in global memory.

```
__shared__ float partialSum[SIZE];
partialSum[threadIdx.x] = X[blockIdx.x*blockDim.x+threadIdx.x];
unsigned int t = threadIdx.x;
for (int stride = 1; stride < blockDim.x; stride *= 2) {
  __syncthreads();
  if (t % (2*stride) == 0)
    partialSum[t] += partialSum[t+stride];
}
```



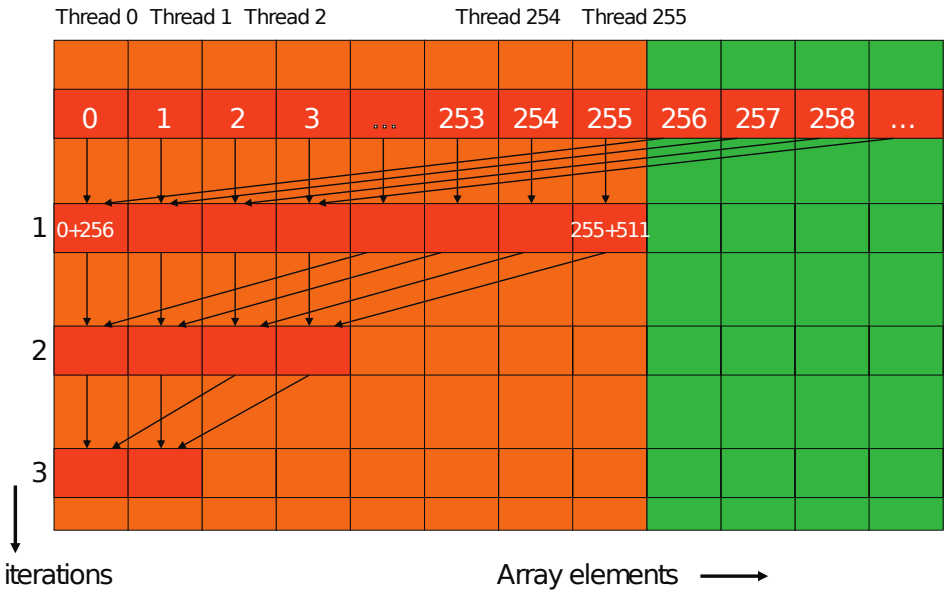
Warps and SIMD Hardware

- During the first iteration of the loop, only those threads whose `threadIdx.x` are even will execute the add statement. One pass will be needed to execute these threads and one additional pass will be needed to execute those that do not execute the `partialSum[t] += partialSum[t+stride];` statement.
- In each successive iteration, fewer threads will execute the `partialSum[t] += partialSum[t+stride];` statement but two passes will be still needed to execute all the threads during each iteration.
- This divergence can be reduced with a slight change to the algorithm.

Warps and SIMD Hardware

- Here is an improved parallel reduction algorithm.

```
__shared__ float partialSum[SIZE];
partialSum[threadIdx.x] = X[blockIdx.x*blockDim.x+threadIdx.x];
unsigned int t = threadIdx.x;
for (int stride = blockDim.x/2; stride >= 1; stride = stride>>1) {
  __syncthreads();
  if (t < stride)
    partialSum[t] += partialSum[t+stride];
}
```



Warps and SIMD Hardware

- Instead of adding neighbor elements in the first round, now we add elements that are half a section away from each other.
- During the first iteration, all threads whose `threadIdx.x` values are less than half of the size of the section execute the `partialSum[t] += partialSum[t+stride];` statement.
- For a section of 512 elements, Threads 0 through 255 execute the add-statement during the first iteration while threads 256 through 511 do not.
- The pair-wise sums are stored in elements 0 through 255 after the first iteration. Since the warps consists of 32 threads with consecutive `threadIdx.x` values, all threads in warp 0 through warp 7 execute the add-statement, whereas warp 8 through warp 15 all skip the add-statement.
- Since all threads in each warp take the same path, there is no thread divergence!

Warps and SIMD Hardware

- Actually, the improved algorithm does not completely eliminate the divergence caused by the if-statement.
- In fact, starting with the 5th iteration, the number of threads that execute the add-statement will fall below 32: the final five iterations will have only 16, 8, 4, 2, and 1 thread(s) performing the addition, resulting in a divergent behaviour.
- However, the number of divergence iterations is reduced from 10 to 5.
- The difference between the two algorithms is small but has very significant performance impact.

Dynamic Partitioning of Resources

- As we already know, CUDA dynamically partitions the limited SM resources (registers, shared memory, thread block slots, and thread slots) and assigns them to threads to support their execution.
- For instance, Fermi devices have 1536 thread slots. These slots are partitioned and assigned to thread blocks during runtime:
  - If each block consists of 512 threads, the 1536 slots are partitioned and assigned to 3 blocks, meaning that each SM can accommodate up to three thread blocks.
  - If each block contains of 256 threads, the 1536 thread slots are partitioned and assigned to 6 thread blocks.
- The ability to dynamically partition the thread slots among thread blocks makes SMs versatile. They can either execute many thread blocks each having few threads, or *vice versa*.

Dynamic Partitioning of Resources

- WARNING: Dynamic partitioning of resources can lead to underutilization of resources.
- For example, if we set each block to have 128 threads, the 1536 thread slots can be partitioned and assigned to 12 blocks. However, since there are only 8 block slots in each SM (due to the hardware design), only 8 blocks will be allowed, and therefore only 128x8=1024 thread slots will be utilized.
  - To fully utilize both the block slots and thread slots, one needs at least 256 threads in each block.
- In addition, by dynamically partitioning registers among blocks, the SM can accommodate more blocks if they require few registers, and fewer blocks if they require more registers...

Dynamic Partitioning of Resources

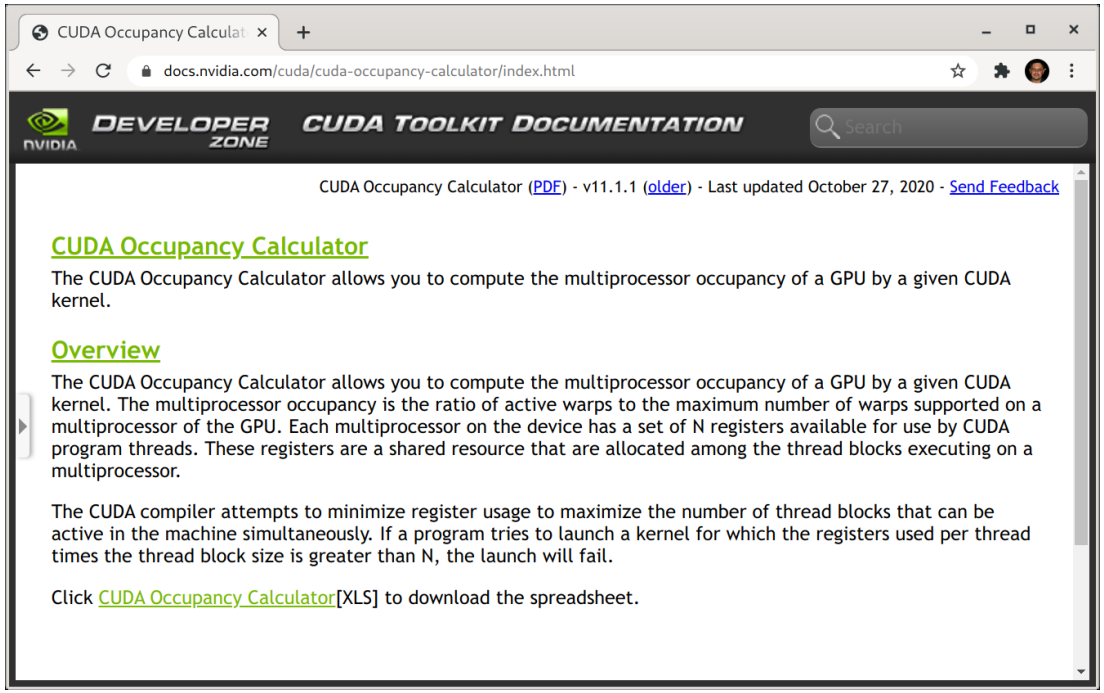
- Let us suppose that each SM has 16,384 registers.
- If the kernel uses 10 registers and the block size is 256 (e.g., for a block of 16x16 threads), the threads per block are 2,560, and the registers required by 6 blocks are 6x2,560=15,360 < 16,384 (i.e., the SM registers). Adding another block would exceed the limit.
- Adding 2 more automatic variables would require 12 registers per thread, resulting in 12*256=3,072 threads per block; the registers required by 6 blocks become 6x3,072=18,432 > 16,384.
- In the latter case, CUDA reduces the number of blocks to respect the registers limit. In the example, the number of blocks is decreased to 5, so that the required registers are 5x3,072=15,360 < 16,384. The side effect is that the number of running threads drops from 1536 to 1280, resulting in an under-utilization of the SM. This phenomenon is known as *performance cliff*.

Dynamic Partitioning of Resources

- Shared memory is another resource that is dynamically partitioned at run-time.
- Unfortunately, large shared memory usage can reduce the number of thread blocks running on an SM.
- Reduced thread parallelism can negatively affect the utilization of the memory access bandwidth of the DRAM system.
- The reduced memory access throughput, in turn, can further reduce the thread execution throughput. This is a pitfall that can result in disappointing performance.
- It is clear that the dynamically partitioned resources interact with each other in a complex manner and can be difficult to determine the best parameters.

Dynamic Partitioning of Resources

- The CUDA Occupancy Calculator spreadsheet `https://docs.nvidia.com/cuda/cuda-occupancy-calculator/index.html` permits to calculate the actual number of threads running on each SM for a device given the usage of resources by a kernel.



Generalizing the Tiled Matrix Multiplication Algorithm

```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P,
    unsigned int j, unsigned int k, unsigned int l) {
  __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
  __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];
  int tx = threadIdx.x; int ty = threadIdx.y;
  int Row = blockIdx.y*TILE_WIDTH+ty, Col = blockIdx.x*TILE_WIDTH+tx;

  float Pvalue = 0
  for (int ph = 0; ph < k/TILE_WIDTH; ++ph) {
    if ((Row < j) && (ph*TILE_WIDTH+tx) < k)
      Mds[ty][tx] = d_M[Row*k + ph*TILE_WIDTH + tx];
      else Mds[ty][tx] = 0;
    if ((ph*TILE_WIDTH+ty) < k && Col < l)
      Nds[ty][tx] = d_N[(ph*TILE_WIDTH + ty)*l + Col];
      else Nds[ty][tx] = 0;
    __syncthreads();
    for (int i = 0; i < TILE_WIDTH; ++i)
      Pvalue += Mds[ty][i] * Nds[i][tx];
    __syncthreads();
  }
  if ((Row<j) && (Col<l)) d_P[Row*l + Col] = Pvalue;
}
```

Dynamic Partitioning of Resources

- The GTX 980 GPU (Maxwell architecture) has:
  - CUDA Compute Capability: 5.2
  - Shared Memory size: 98304 bytes
  - Global Load Caching Mode: L2 only (cg)[1]
- The above kernel has:
  - Threads per block: 512[2]
  - Registers per thread: 13[3]
  - Used Shared Memory per block: 128 bytes[4]

---

[1] See:
`https://docs.nvidia.com/cuda/maxwell-tuning-guide/index.html`
Section "1.4.2.1. Unified L1/Texture Cache".

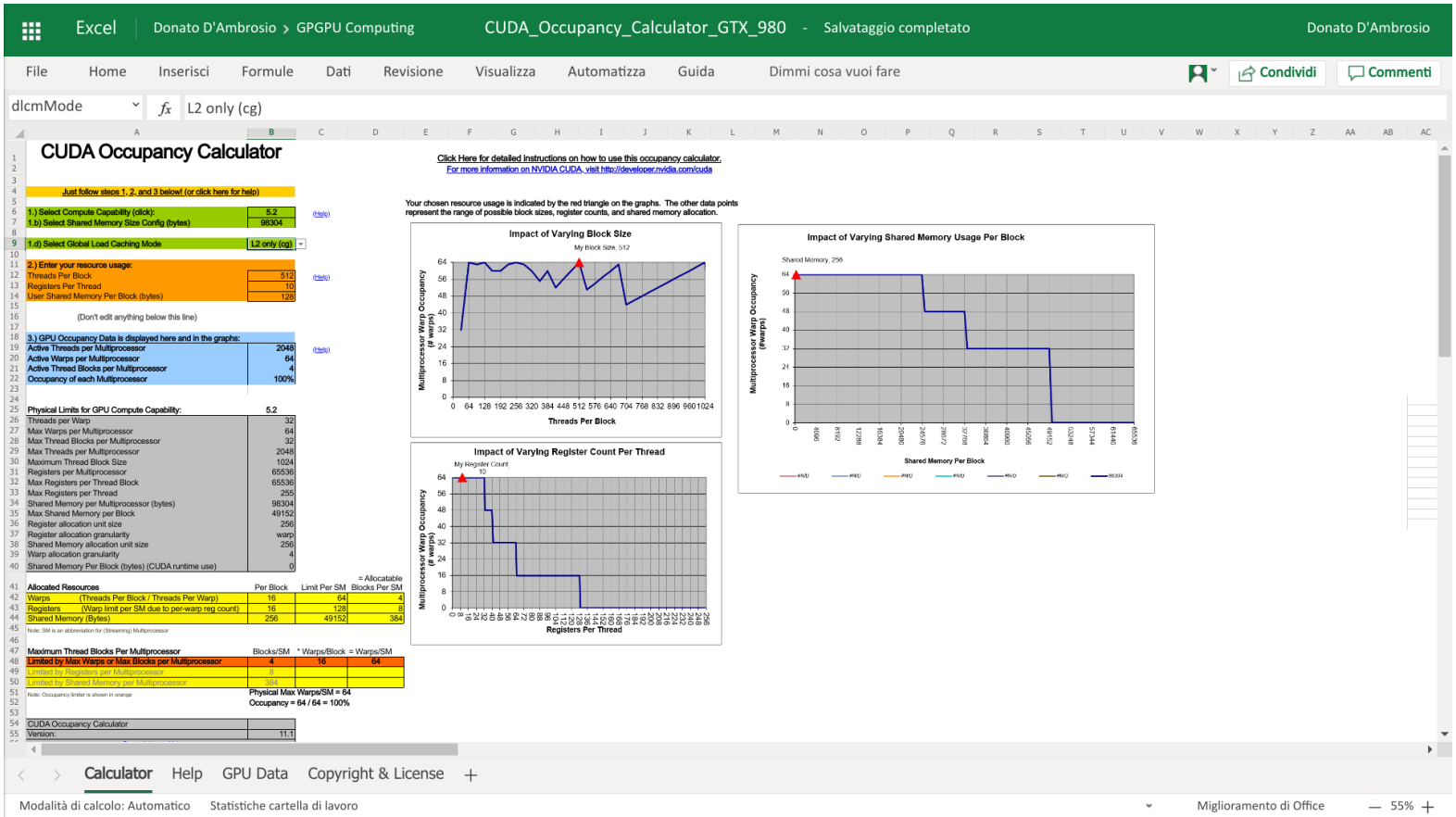
[2] Value defined at kernel invocation.

[3] Number of automatic variables, formal parameters included.

[4] 2*TILE_WIDTH*TILE_WIDTH*sizeof(float).

Dynamic Partitioning of Resources

- The calculator states that the warp occupancy is maximized (see the red triangle).

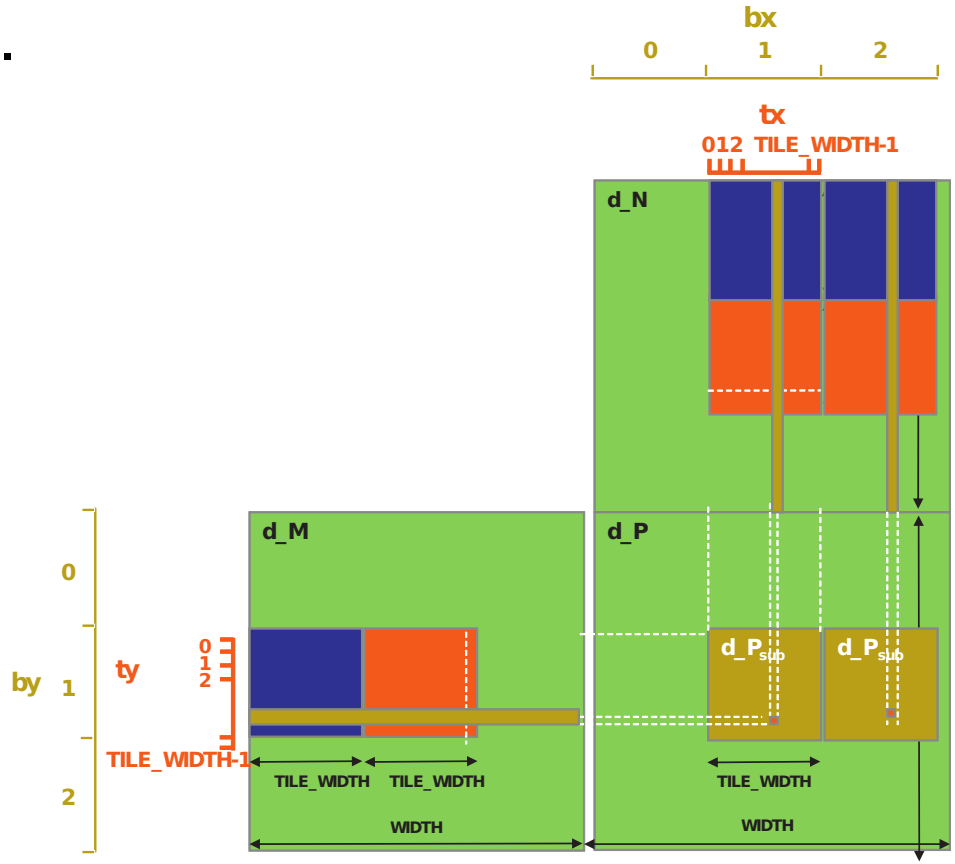


Thread Granularity

- An important algorithmic decision in performance tuning is the granularity of threads.
- Sometimes it is advantageous to put more work into each thread and use fewer threads if the finer granularity threads have redundant work.
- For instance, the matrix multiplication algorithm uses one thread per element of P, requiring a dot-product between one row of `M` and one column of `N`.
- The opportunity of thread granularity adjustment comes from the fact that multiple blocks redundantly load each M tile (the calculation of two P elements in adjacent tiles uses the same M row).
- With the original tiled algorithm, the same M row is redundantly loaded by the two blocks assigned to generate these two P tiles.

Thread Granularity

- The redundancy can be eliminated by merging the two thread blocks into one.



- Each thread now calculates two P elements using the same `Mds` row but different `Nds` columns. This reduces the global memory access by one-quarter.

Thread Granularity

- The potential downside is that the new kernel now uses even more registers and shared memory.
  - As we discussed, the number of blocks that can be running on each SM may decrease.
- For a given matrix size, this also reduces the total number of thread blocks by half, which may result in an insufficient amount of parallelism for matrices of smaller dimensions. In practice, combining up to four adjacent horizontal blocks to compute adjacent horizontal tiles significantly improves the performance of large (2048×2048 or more) matrix multiplication.