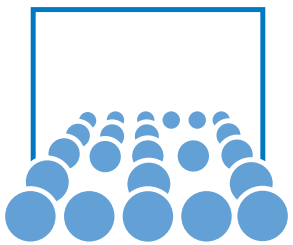
Further Details for Dense Linear Algebra in CUDA

Oliver Meister

November 21st 2012



Last Tutorial

GPU Architecture

- many SMP's which execute warps in parallel
- hierarchy in processing and memory

CUDA API

- host code – executed on CPU
  - problem initialization
  - memory allocation
- device code – executed on GPU
  - execution of lightweight kernels
  - hierarchical communication between kernels

Dense Matrix Multiplication

Simple Implementation

- max. size of 16

Seperation in Threadblocks

- arbitrary matrix size
- still only access to global memory

Using Tiles

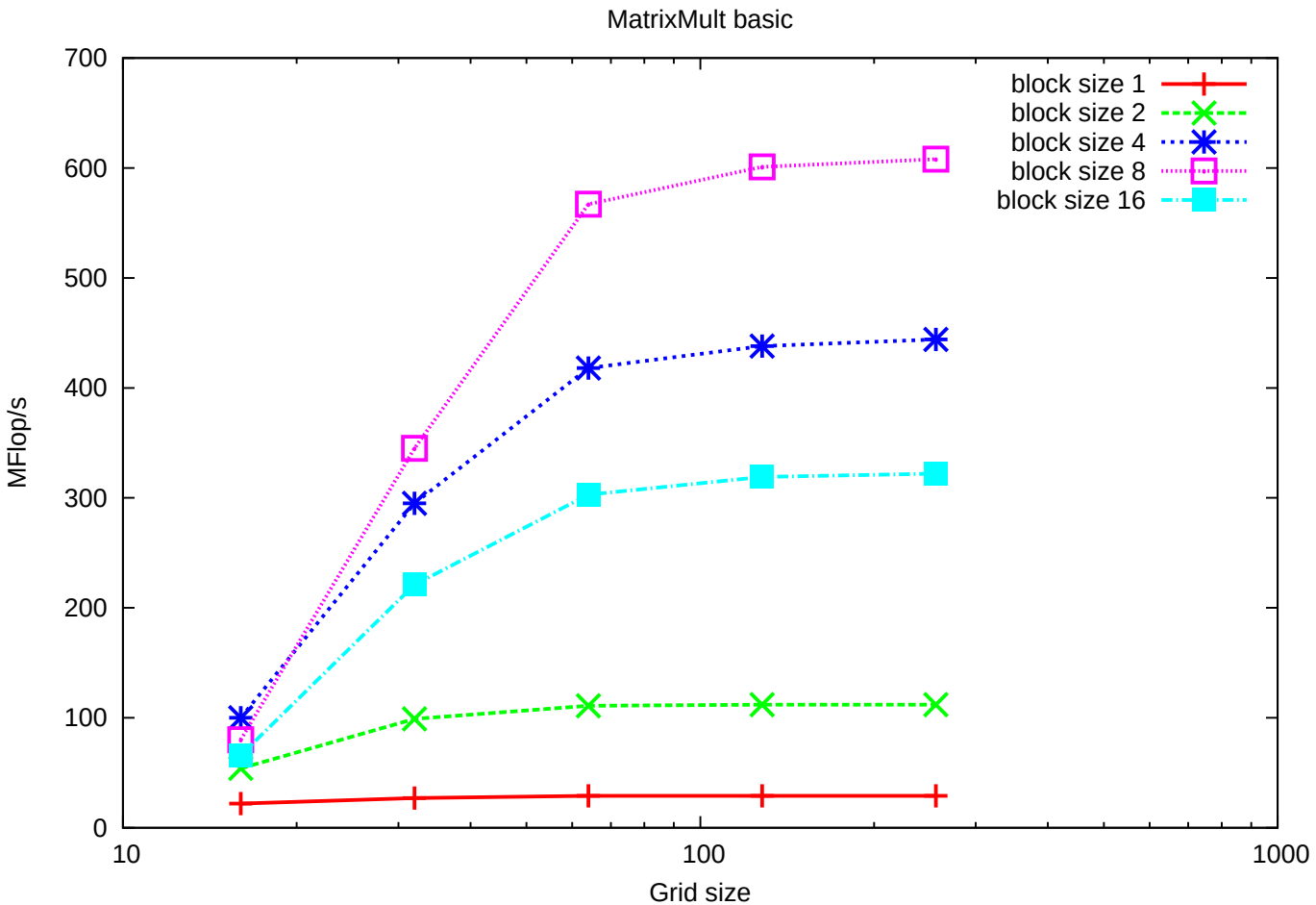
- exploit the fast shared memory
- then do computation

Assignment 1

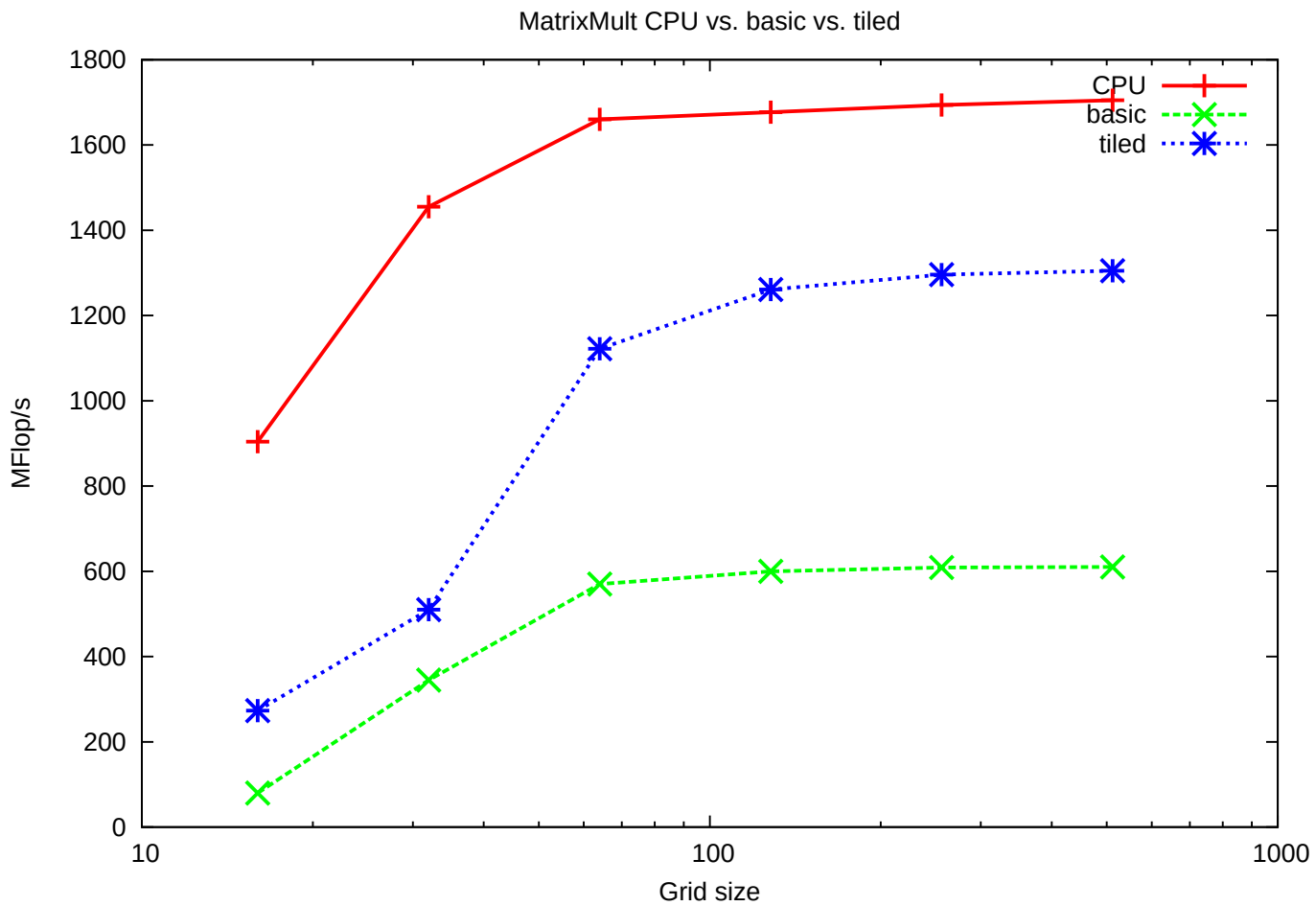
```
__global__ void matrixMultKernel(float* Ad, float* Bd,
                                 float* Cd, int n) {
   int i = threadIdx.x;
   int k = threadIdx.y;
   float Celem = 0;
   for(int j=0; j<n; j++) {
      float Aelem = Ad[i*n+j];
      float Belem = Bd[j*n+k];
      Celem += Aelem*Belem;
   }
   Cd[i*n+k] += Celem;
}
```

Results?

Assignment 2



Assignment 3



CUDA Profiler

Counters (Selection)

Occupancy	number of active warps / max. number of active warps
Branch	number of branches, that might split a warp
Warp serialize	Serialization of a warp due to branch / memory access
gld / gst uncoalesced	global memory operations which serialize the warps
instruction throughput	achieved instruction rate compared to the peak instruction rate

Global Memory Architecture

- DRAM (i.e., global memory) built, such that multiple contiguous memory slots are read together (compare: cache lines)

Warp Serialize

- access to global memory requires serialization
- each thread gets its own piece of memory and not consecutive entries

leads to uncoalesced memory access, since a single warp (max. 16 threads) can only execute a single instruction per cycle. When access to global memory is not coalesced, it is serialized.

Coalesced Memory Access

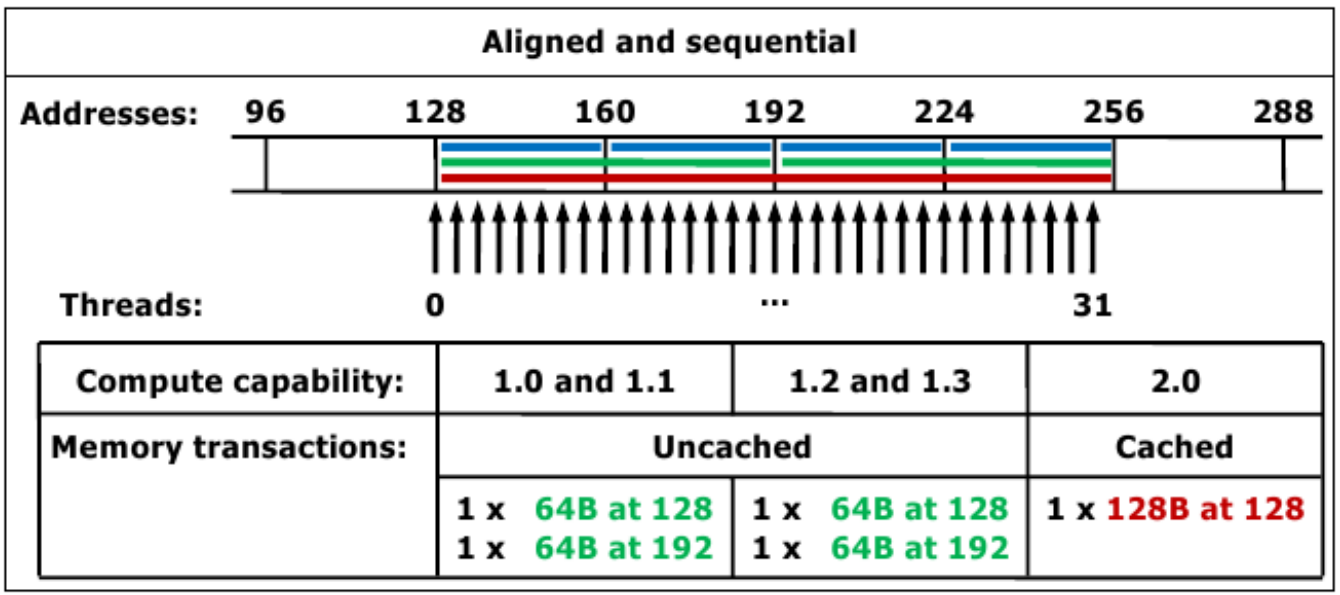
- necessary for maximal bandwidth
- neighboring threads in a warp should access neighboring memory addresses
- not all threads have to participate
- have a valid starting address
- have the right order
- strides are allowed but the speed is reduced significantly
- have to have the right order
- no double accesses

exact compute pattern is depending on the compute capability

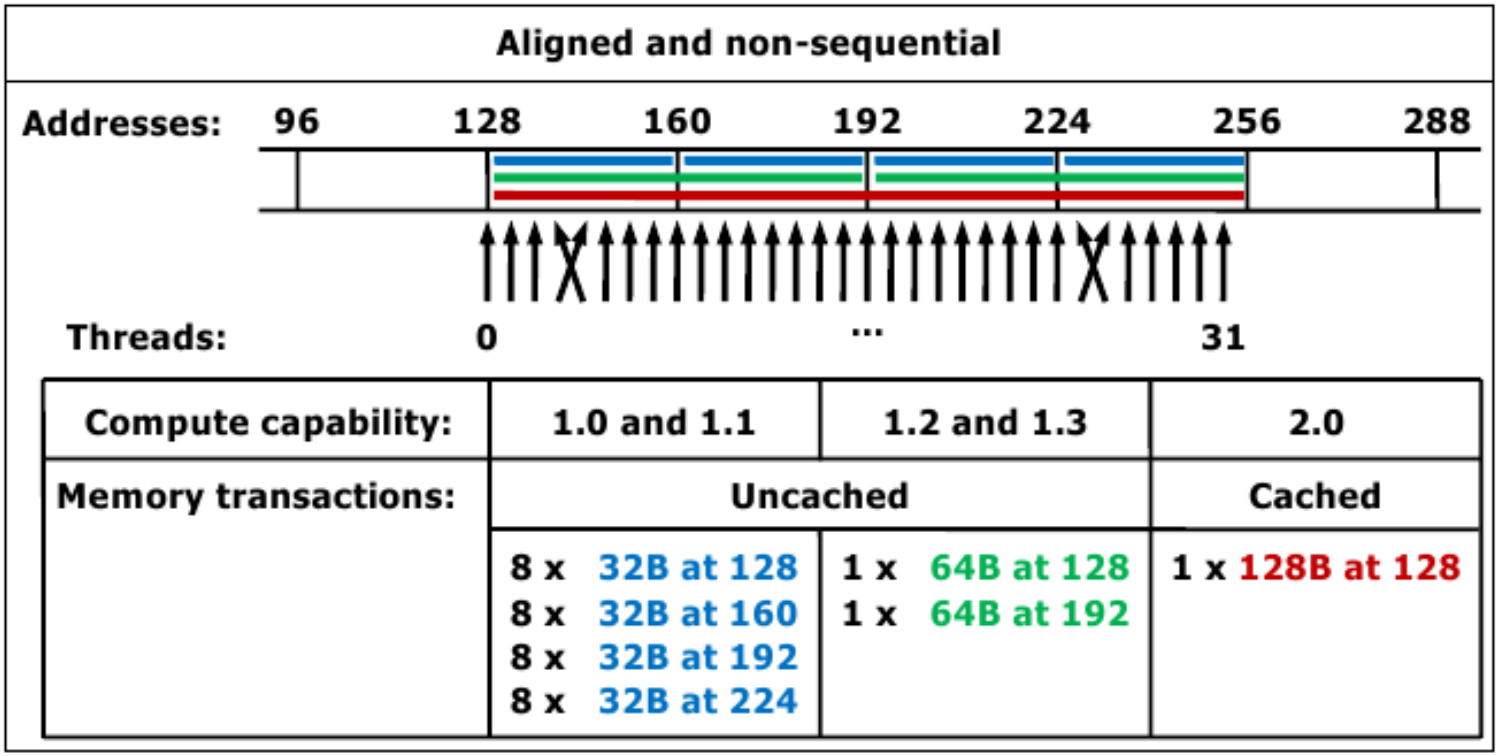
Compute Capability

Different Compute Capabilities of devices (1.x to 2.x)

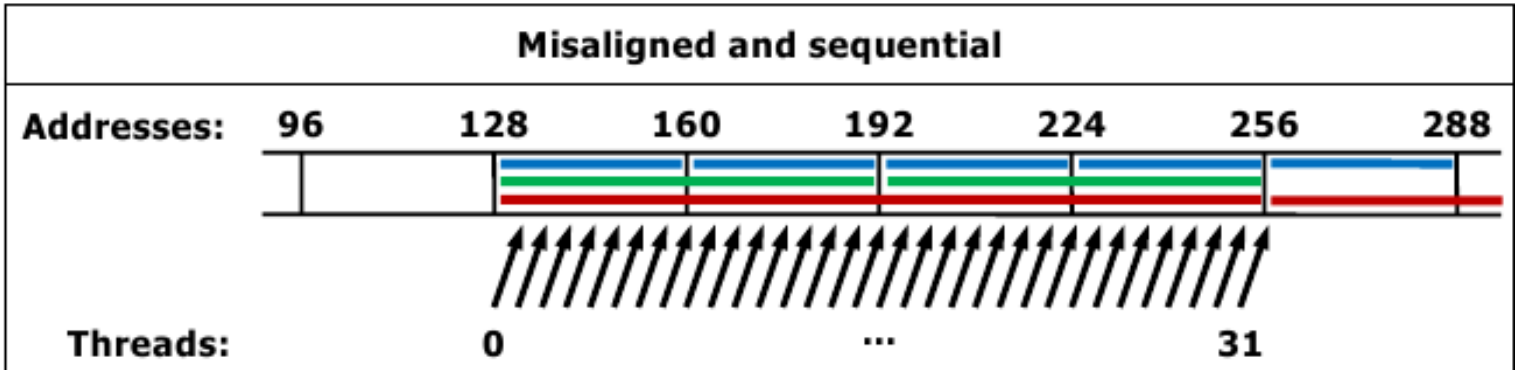
- different technical specifications (caches,number of SM's ...)
- different treatment of
  - global memory
  - shared memory



Compute capability:	1.0 and 1.1	1.2 and 1.3	2.0
Memory transactions:	Uncached		Cached
	1 x 64B at 128 1 x 64B at 192	1 x 64B at 128 1 x 64B at 192	1 x 128B at 128



Compute capability:	1.0 and 1.1	1.2 and 1.3	2.0
Memory transactions:	Uncached		Cached
	8 x 32B at 128 8 x 32B at 160 8 x 32B at 192 8 x 32B at 224	1 x 64B at 128 1 x 64B at 192	1 x 128B at 128



Compute capability:	1.0 and 1.1	1.2 and 1.3	2.0
Memory transactions:	Uncached		Cached
	7 x 32B at 128 8 x 32B at 160 8 x 32B at 192 8 x 32B at 224 1 x 32B at 256	1 x 128B at 128 1 x 64B at 192 1 x 32B at 256	1 x 128B at 128 1 x 128B at 256

Coalesced Access in Matrix Multiplication

Copy matrix tile *m* from global into shared memory:

```
Ads[tx][ty] = Ad[ i*n + m*TILE_SIZE+ty];
Bds[tx][ty] = Bd[ (m*TILE_SIZE+tx)*n + k];
__syncthreads();
```

Do we have coalesced memory access?

Coalesced Access in Matrix Multiplication

Copy matrix tile *m* from global into shared memory:

```
Ads[tx][ty] = Ad[ i*n + m*TILE_SIZE+ty];
Bds[tx][ty] = Bd[ (m*TILE_SIZE+tx)*n + k];
__syncthreads();
```

Do we have coalesced memory access?

- row computation: `i = blockIdx.x * TILE_SIZE + tx`
- column computation: `k = blockIdx.y * TILE_SIZE + ty`

$\Rightarrow$ stride-1 access w.r.t. `ty`, stride-n access w.r.t. `tx`

Coalesced Access in Matrix Multiplication

Copy matrix tile *m* from global into shared memory:

```
Ads[tx][ty] = Ad[ i*n + m*TILE_SIZE+ty];
Bds[tx][ty] = Bd[ (m*TILE_SIZE+tx)*n + k];
__syncthreads();
```

Do we have coalesced memory access?

- row computation: `i = blockIdx.x * TILE_SIZE + tx`
- column computation: `k = blockIdx.y * TILE_SIZE + ty`

$\Rightarrow$ stride-1 access w.r.t. `ty`, stride-n access w.r.t. `tx`

Question: which threads are combined in a warp?

Warps and Coalesced Access

Combination of threads into warps:

- 1D thread block: thread 0, . . . 31 into warp 0; thread 32, . . . 63 into warp 1; etc.
- 2D thread block: x-dimension is “faster-running”; e.g.:
  - `dimBlock(8,8,1)`, i.e., 64 threads (2 warps)
  - then threads (0,0), . . . , (7,3) are in warp 0 and threads (0,4), . . . , (7,7) are in warp 1
- 3D thread block: x, then y, then z

Warps and Coalesced Access

Combination of threads into warps:

- 1D thread block: thread 0, . . . 31 into warp 0; thread 32, . . . 63 into warp 1; etc.
- 2D thread block: x-dimension is “faster-running”; e.g.:
  - `dimBlock(8,8,1)`, i.e., 64 threads (2 warps)
  - then threads (0,0), . . . , (7,3) are in warp 0 and threads (0,4), . . . , (7,7) are in warp 1
- 3D thread block: x, then y, then z

Tile-Copying in Matrix Multiplication:

- threads with consecutive `tx` value in one warp
- leads to stride-n access $\Rightarrow$ **not coalesced**

Matrix Multiplication with Coalesced Access

Switch tx and ty ⇒ stride-1 (coalesced) access to Ads, Bds:

```
int i = blockIdx.y * TILE_SIZE + ty;
int k = blockIdx.x * TILE_SIZE + tx;
float Celem = 0;
for(int m=0; m < n/TILE_SIZE; m++) {
   Ads[ty][tx] = Ad[ i*n + m*TILE_SIZE+tx];
   Bds[ty][tx] = Bd[ (m*TILE_SIZE+ty)*n + k];
   __syncthreads();
   for(int j=0; j<TILE_SIZE; j++)
      Celem += Ads[ty][j]*Bds[j][tx];
   __syncthreads();
};
Cd[i*n+k] += Celem;
```

Check Profiler

- only coalesced memory access
- dependence on tile size
- $\approx 6 \times 10^9$ Flops
- no divergent branches, no serialized warps
- but still reserves in instruction throughput

Assignment

1. Use the cuda profiler and check the performance of the kernels implemented so far. Find the bottleneck by using the appropriate performance counters.
2. Implement the matrix multiplication kernel with coalesced global memory access and check the results with the profiler. Compare the achieved number of active threads/warps to the maximum of the used GPU. Do you exploit all parallelism provided by the device?

Shared Memory Access

- less restrictive than global memory accesses
- organized in banks of 32-bits
- if memory accesses are conflicting, execution is serialized
- broadcasting and random access possible
- exact behavior depending on compute capability

Memory Latency for Tile Transfers

Recapitulate tiled matrix multiplication:

- tiles of 16×16 matrix elements
  $\rightarrow 16^2 = 256$ threads per tile (also per thread block)
- thus: 8 warps (32 threads each)
- examine load operation for matrix tiles

```
Ads[ty][tx] = Ad[ i*n + m*TILE_SIZE+tx];
Bds[ty][tx] = Bd[ (m*TILE_SIZE+ty)*n + k];
__syncthreads();
```

  $\rightarrow$ delay due to memory latency
- all threads in a warp wait for data to arrive
- but another warp can be scheduled to work

Tiled Matrix Multiplication with Prefetching

Include prefetching of blocks to reduce “idle” time for memory transfer:

1. load first tile into register(s)
2. copy register(s) to shared memory
3. barrier
4. load next tile into register(s)
5. compute current tile
6. barrier
7. proceed with 2 (if there are more tiles)
8. compute last tile

Registers

but no massive performance gain

- before prefetching 10 Registers per thread $\rightarrow$ $16 \times 16 \times 10 = 2560$ registers per block
- 8192 registers per SM $\rightarrow$ 3 active blocks

Registers

but no massive performance gain

- before prefetching 10 Registers per thread $\rightarrow$ $16 \times 16 \times 10 = 2560$ registers per block
- 8192 registers per SM $\rightarrow$ 3 active blocks
- now 16 registers $\rightarrow$ $16 \times 16 \times 16 = 4096$ registers per block
- only 8192 registers per SM $\rightarrow$ only 2 active blocks per SM!

Registers

but no massive performance gain

- before prefetching 10 Registers per thread $\rightarrow$ $16 \times 16 \times 10 = 2560$ registers per block
- 8192 registers per SM $\rightarrow$ 3 active blocks
- now 16 registers $\rightarrow$ $16 \times 16 \times 16 = 4096$ registers per block
- only 8192 registers per SM $\rightarrow$ only 2 active blocks per SM!

Number of active threads

- before: 3 active blocks $\rightarrow$ 24 active warps $\rightarrow$ 768 active threads $\rightarrow$ optimal occupancy
- after: 2 active blocks $\rightarrow$ 16 active warps $\rightarrow$ 512 active thread $\rightarrow$ less parallelism, less latency hide

Test it for your system!

Dynamic Partitioning of Resources

Leads to trade-offs for performance:

- always depending on the used hardware
- limited number of threads reduces parallelism (and, thus, achievable performance)
- “performance cliffs”: slight change in set-up might lead to jumps in available blocks
- requires detailed estimates (or lots of testing?) to determine best option

Towards High-Performance Matrix Multiplication

More Options for Optimisation:

- loop unrolling (save loop instructions and address arithmetics)
- thread granularity: compute 1×2 or 1×4 blocks per thread (requires to load Ads or Bds only once)
- how do different optimisations interact with resource limitations (available registers, etc.)

Assignments

3. Implement the prefetching algorithm and use the profiler again to check the performance. Identify the new bottlenecks in the code. Which version has more performance for your GPU?
4. Try to apply loop unrolling and adjust the thread granularity. How do the different adjustments interact with each other? Try to come as close to peak performance as possible!