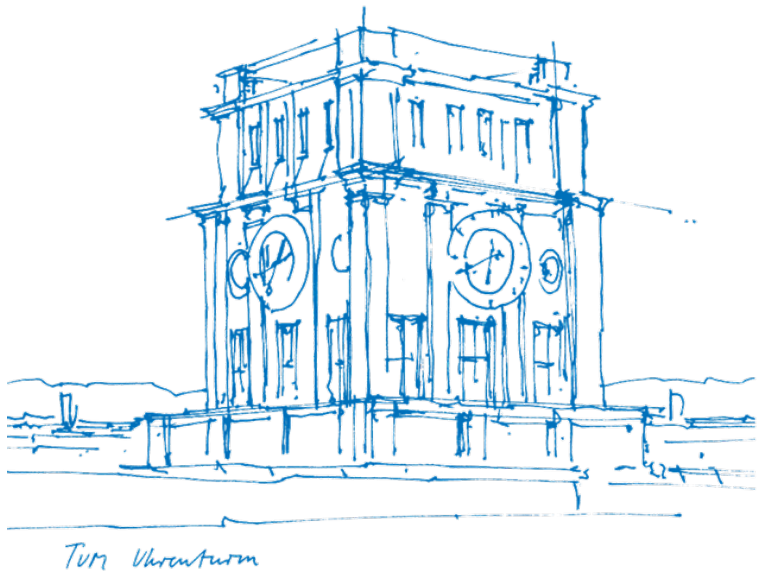
Dense Linear Algebra

HPC - Algorithms and Applications

Alexander Pöppl
Technical University of Munich
Chair of Scientific Computing

23.11.2016



Last Tutorial

CUDA Architecture

- thread hierarchy: threads, warps, blocks, grids
- memory hierarchy: local memory, shared memory, global memory, host memory
- GPU: CUDA cores and load/store units with registers, SM with warp scheduler and L1 cache/shared memory, GPU with L2 Cache, giga thread engine and global memory

Last Tutorial

CUDA API

- host code – executed on CPU
  - memory operations
  - grid and block size definition
  - kernel calls
- device code – executed on GPU
  - execution of lightweight kernels
  - memory-based hierarchical communication between kernels

Dense Matrix Multiplication

Simple Implementation

- Only one block
- Max. Matrix size: 32×32 (or 16×16 on Fermi)

Separation into blocks

- arbitrary matrix size
- still access to global memory only

Using Tiles

- load data into fast shared memory
- computation on shared tile

H1.1 - Make it run

- Trouble with the cluster/CUDA? Contact or visit me!
- My office: LRZ Room E.2.032
  - Enter LRZ main entrance, go down stairs on the left
  - Follow signs to LRZ Building 2 through hallway with black walls and colored windows.
  - Go up to second floor and to the end of the corridor.
  - I am the office on the other side from the tea kitchen.
- Contact me before visiting if you want to be sure I'm here

H1.2 - Tiled Matrix Multiplication

Remarks

- Take care when placing `__syncthreads()` operations. There needs to be a one after transferring tile data from global to local memory and another one after the dot product loop.
- Non-multiples of the tile size may need to be taken care of too.
- Take care to use the right indices.

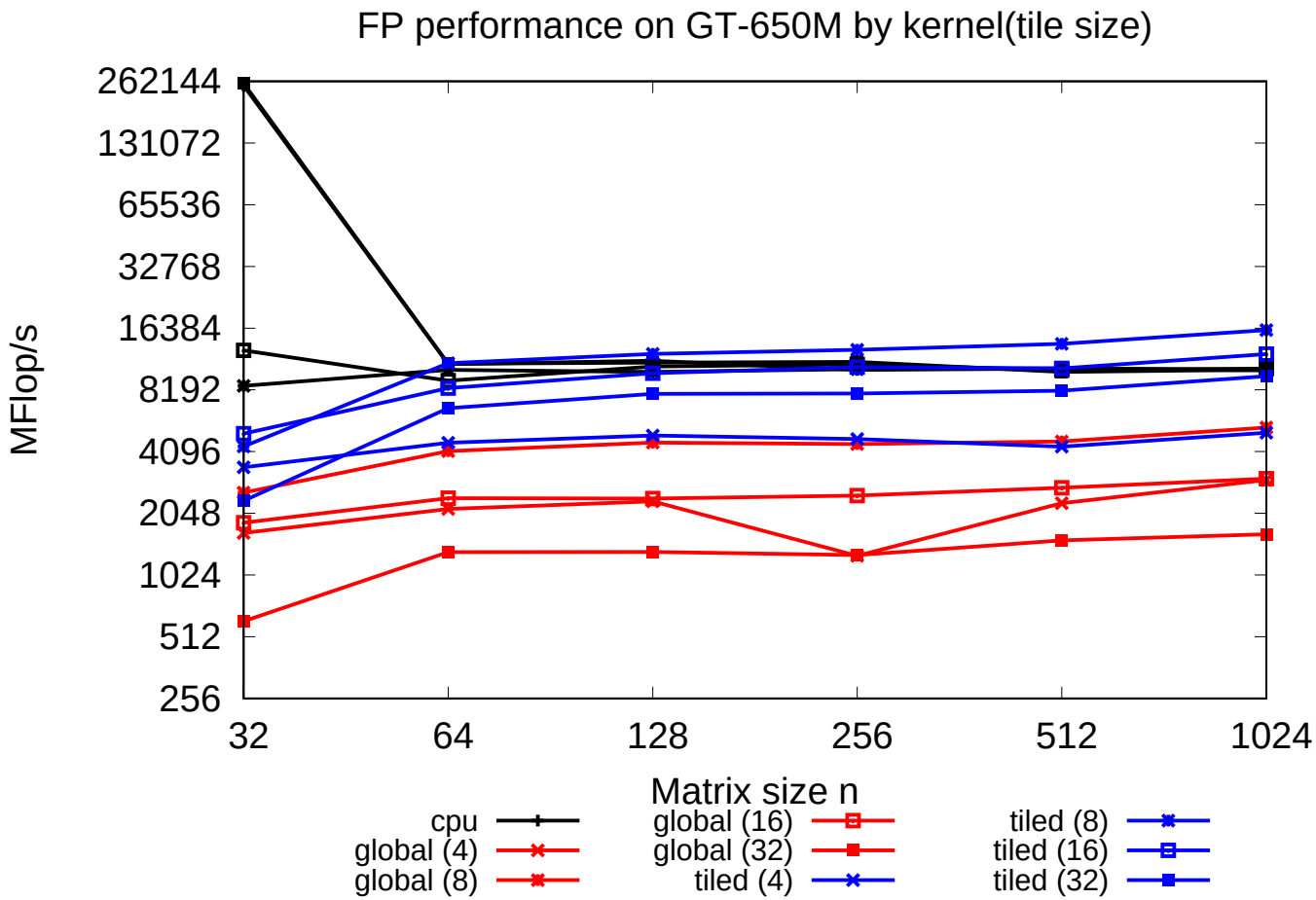
H1.2a - Tiled Matrix Multiplication

```
__global__ void matrixMultKernel(float* Ad, float* Bd,
                                 float* Cd, int n) {

  __shared__ float Ads[TILE_SIZE][TILE_SIZE];
  __shared__ float Bds[TILE_SIZE][TILE_SIZE];

  /* (cont.) */
```

H1.2a - Tiled Matrix Multiplication

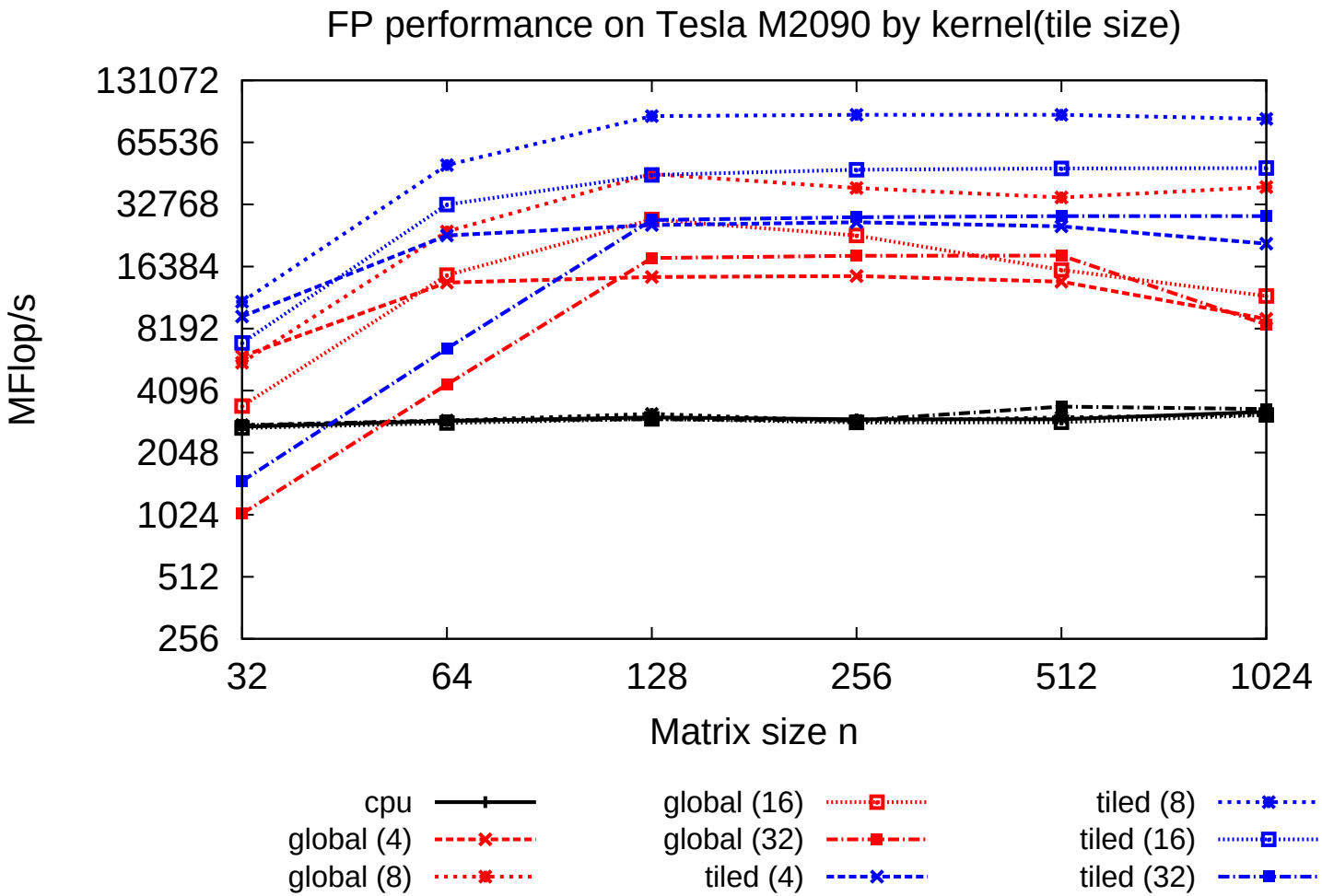
```
    int tx = threadIdx.x; int ty = threadIdx.y;
    int i = blockIdx.x * TILE_SIZE + tx;
    int k = blockIdx.y * TILE_SIZE + ty;

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

H1.2b - Performance measurements



H1.2b, H1.3b - Performance measurements

What are your results?



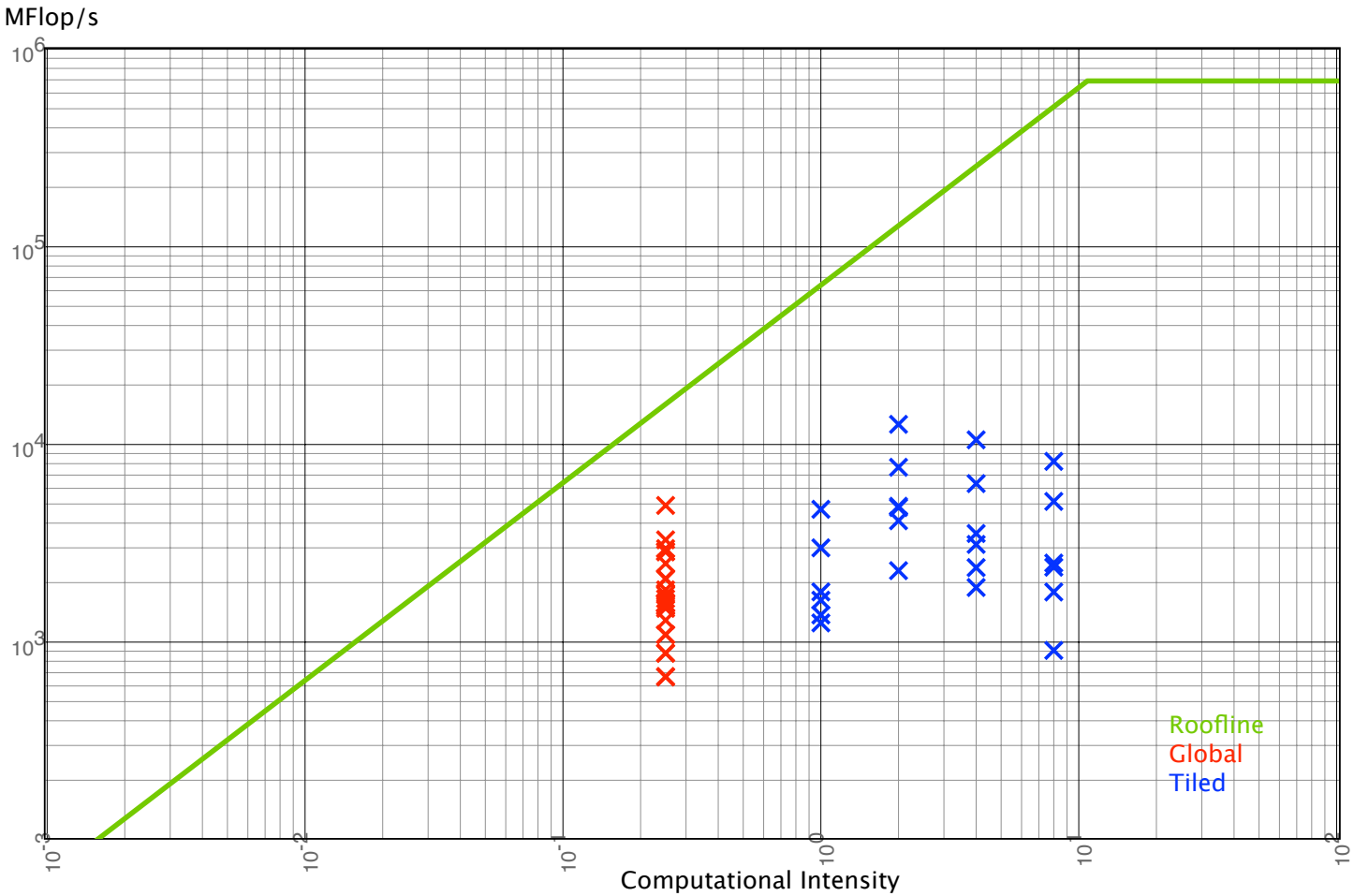
Roofline model

A *roofline model* (Williams, Waterman, Patterson '08) is a helpful tool to model performance issues for an HPC code. It relates floating point performance, memory performance and operational intensity (operations per memory fetch) in a 2D graph.

Task:

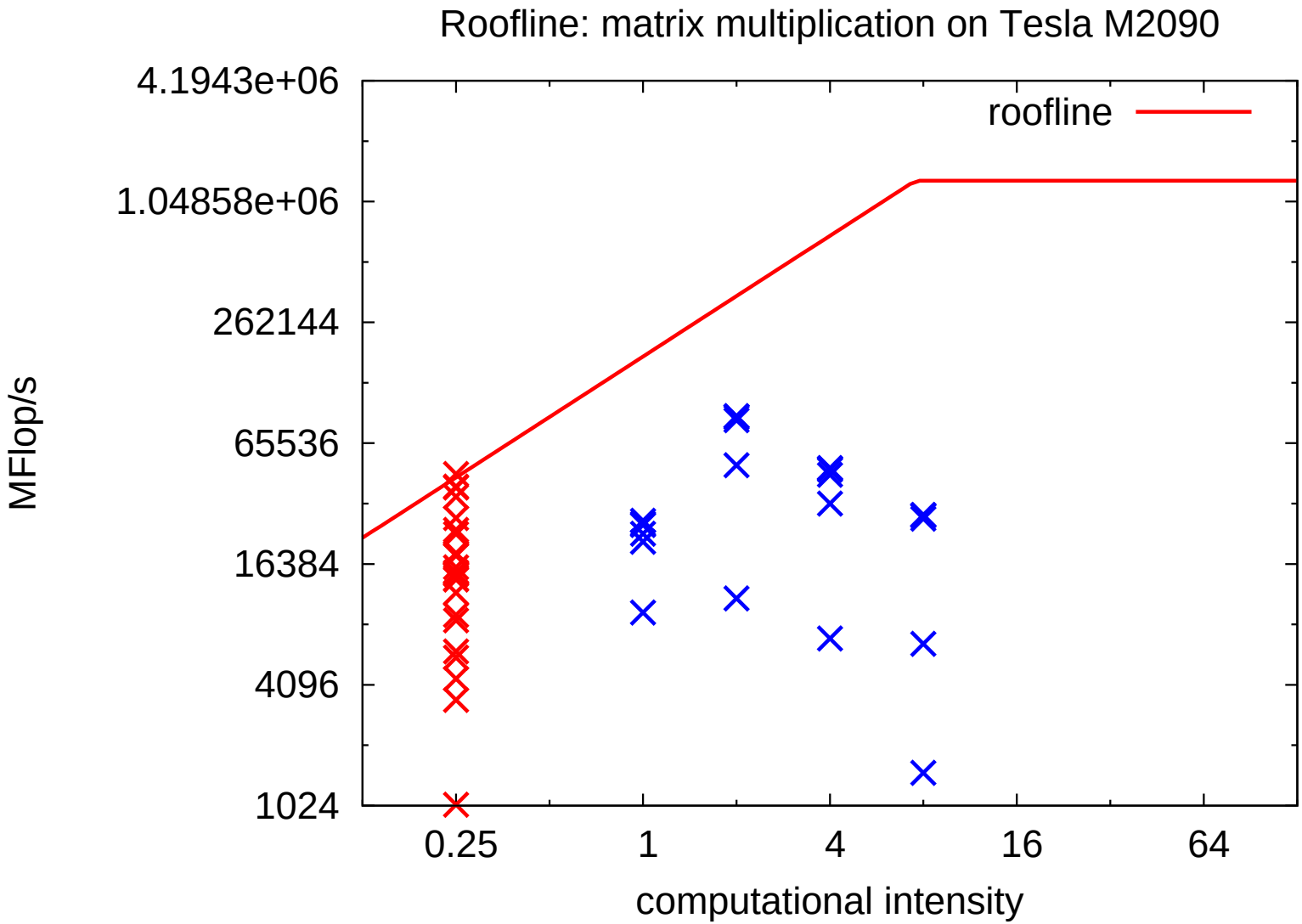
- Draw a roofline model for the NVidia M2090 (peak performance: 1331 GFlop/s, 177.0 GB/s).
- Find computational intensity of basic and tiled matrix multiplication ($\texttt{TILE\_SIZE} = 4, 8, 16, 32$) and insert points with performance measurements.

Roofline model



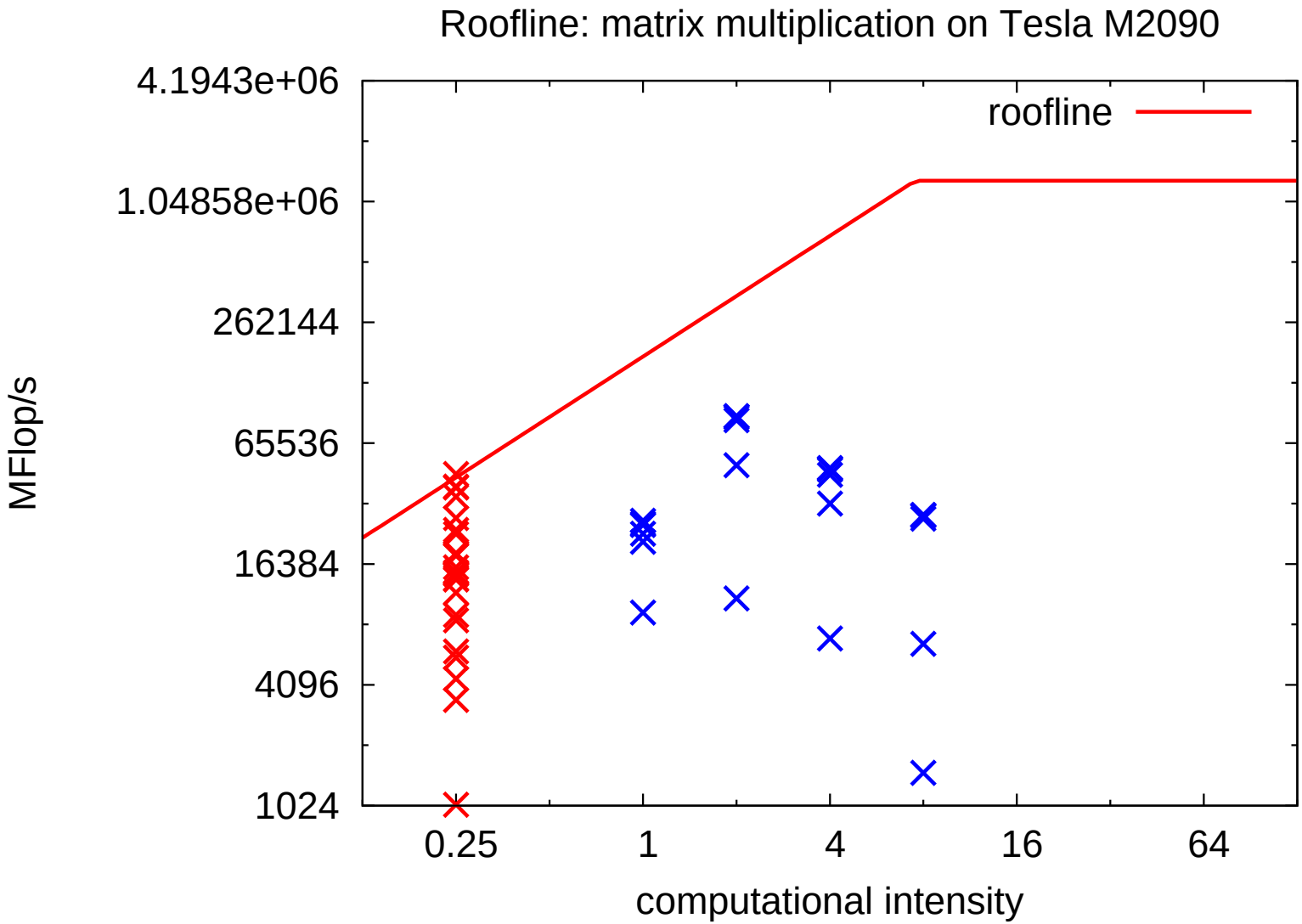
Roofline model for matrix multiplication on GT-650M

Roofline model



What did we miss?

Roofline model



What did we miss? L1 Cache!

Updated Performance Estimate

- m-loop: each thread loads one matrix element from global memory
- j-loop: shared memory $\rightarrow$ no further loads in `TILE_SIZE` iterations
- we have reduced the memory transfer from global memory to `1/TILE_SIZE` of the global kernel
- for `TILE_SIZE = 32`: new performance limit at 1331 GFlop/s

$\rightarrow$ we've eliminated a major bottleneck, but apparently hit another ...

Coalesced Memory Access

Global Memory Architecture

- DRAM (i.e., global memory): multiple contiguous memory slots are read simultaneously (compare: cache lines)

Warp Serialize

- access to global memory can be merged, if nearby memory locations are accessed concurrently
- all threads in a warp access memory concurrently – if data is located in consecutive memory: merge memory calls

A single (half-)warp can only execute a single instruction per cycle $\Rightarrow$ Serialization, *uncoalesced* access.

Coalesced Memory Access

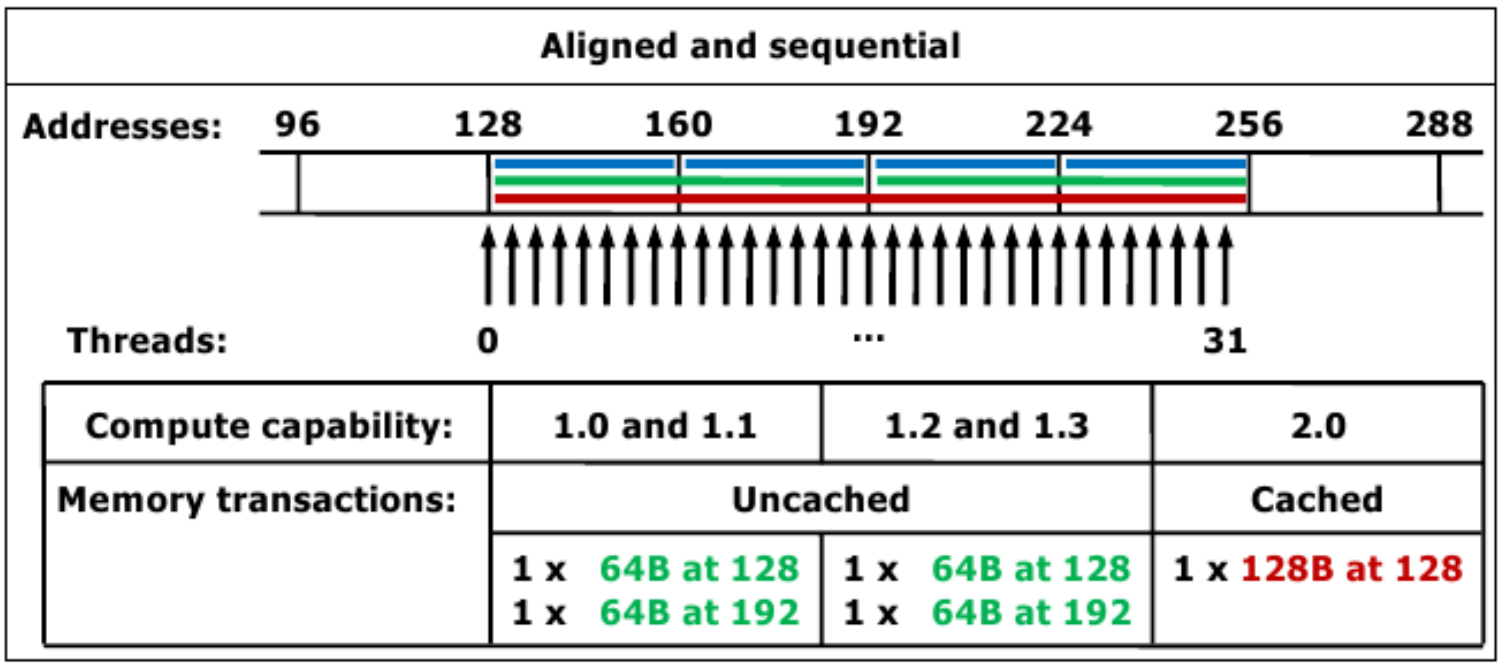
- necessary for maximal bandwidth
- neighboring threads in a warp should access neighboring memory addresses
- not all threads have to participate
- have a valid starting address
- strides are allowed but the speed is reduced significantly
- have to be in order
- no double accesses

exact compute pattern is depending on the compute capability

Compute Capability

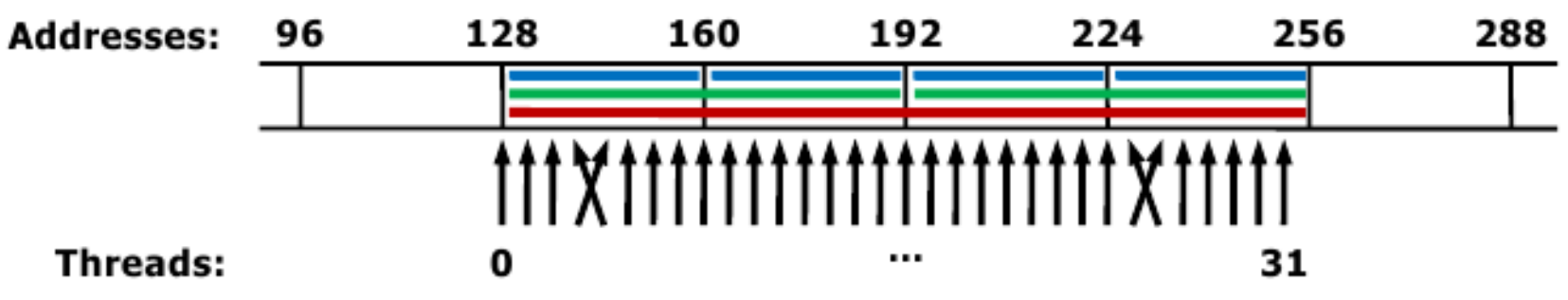
Different Compute Capabilities of devices (1.x to 5.x)

- different technical specifications (caches,number of SMs ...)
- different treatment of
  - global memory
  - shared memory



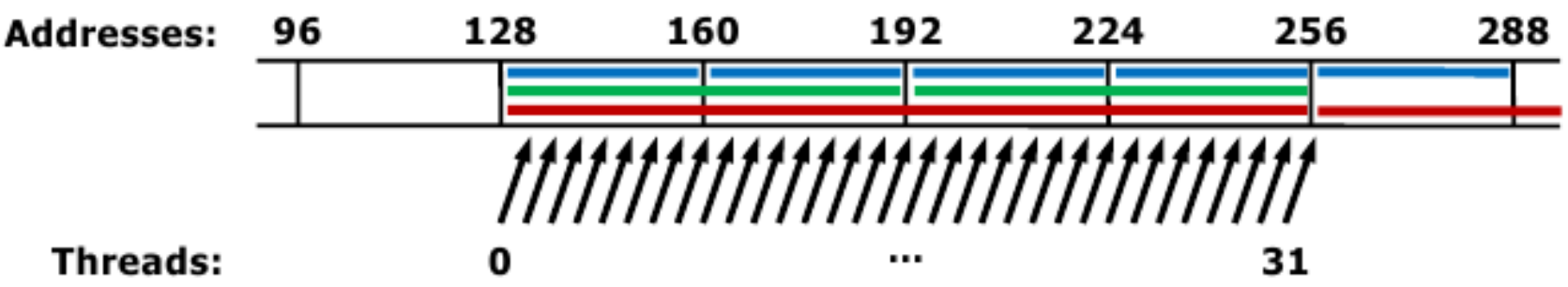
Compute capability:	1.0 and 1.1	1.2 and 1.3	2.0
Memory transactions:	Uncached		Cached
	1 x 64B at 128 1 x 64B at 192	1 x 64B at 128 1 x 64B at 192	1 x 128B at 128

Aligned and non-sequential



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

Misaligned and sequential



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

Global memory access:

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

- 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

Shared memory access:

```
for(int j=0; j<TILE_SIZE; j++)
Celem += Ads[tx][j]*Bds[j][ty];
```

- Matrix layout in CUDA is *row-wise* (same as C/C++)
- stride-`TILE_SIZE` access to `Ads` by `tx`
- stride-1 access to `Bds` by `ty`

Warps and Coalesced Access

Question: How are threads ordered in a block?

Warps and Coalesced Access

Question: How are threads ordered in a block?

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)`, i.e., 64 threads (2 warps)
  - threads (0,0), (7,0), (0,3), (7,3) are in warp 0
    threads (0,4), (7,4), (0,7), (7,7) are in warp 1
- 3D thread block: x, then y, then z

Coalesced Access in Matrix Multiplication

Task:

- Examine our tiled matrix multiplication, are global and shared memory access coalesced?
- If not, what has to be changed?

Matrix Multiplication with Tiling

Original algorithm:

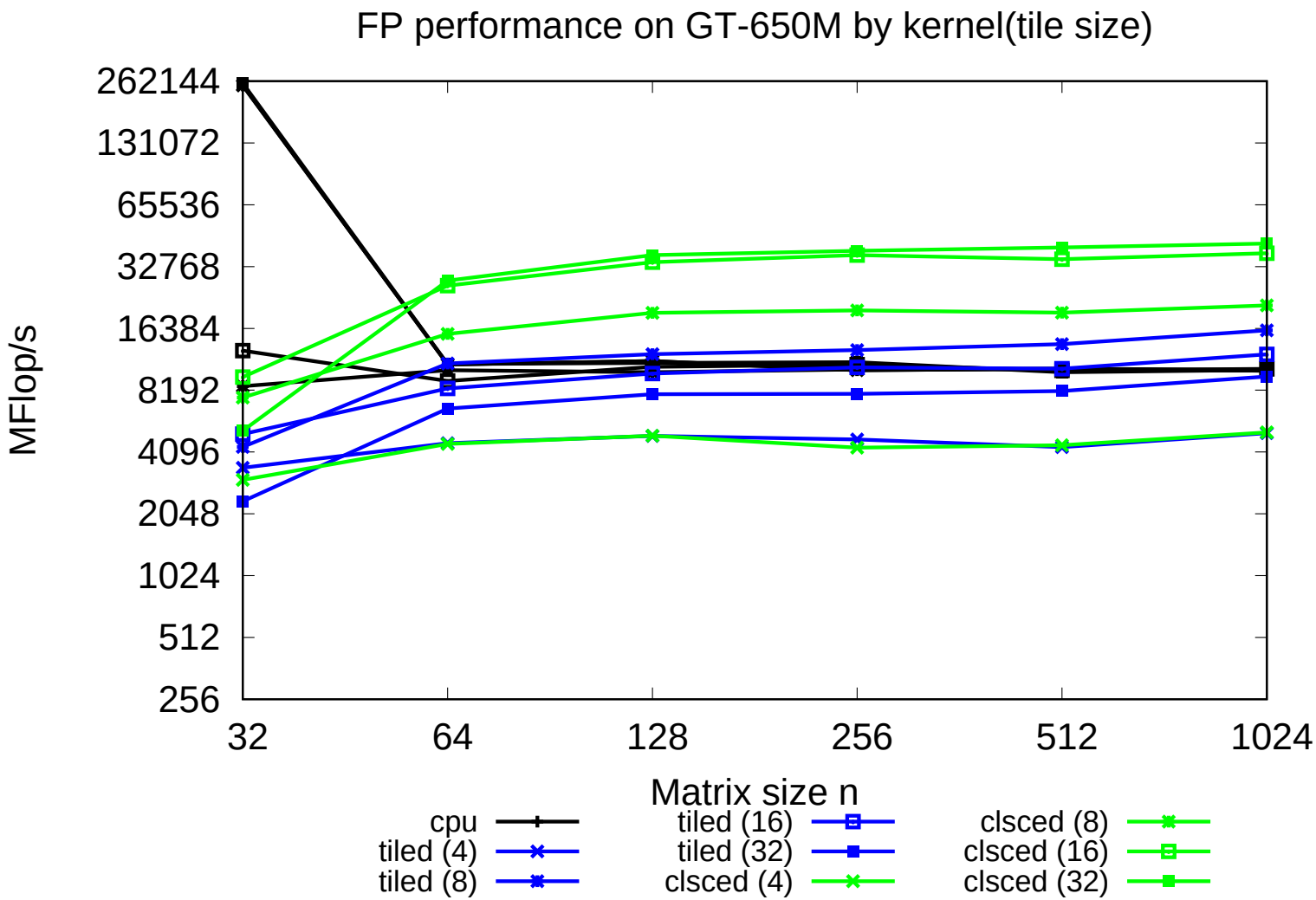
```
int i = blockIdx.x * TILE_SIZE + tx;
int k = blockIdx.y * TILE_SIZE + ty;
float Celem = 0;
for(int m=0; m < n/TILE_SIZE; m++) {
    Ads[tx][ty] = Ad[ i*n + m*TILE_SIZE+ty];
    Bds[tx][ty] = Bd[ (m*TILE_SIZE+tx)*n + k];
    __syncthreads();
    for(int j=0; j<TILE_SIZE; j++)
    Celem += Ads[tx][j]*Bds[j][ty];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

Matrix Multiplication with Coalesced Access

Switch x and y $\Rightarrow$ stride-1 read access to `Ad`, `Bd`, `Bds`, stride-1 write access to `Ads`, `Bds`, `Cd`:

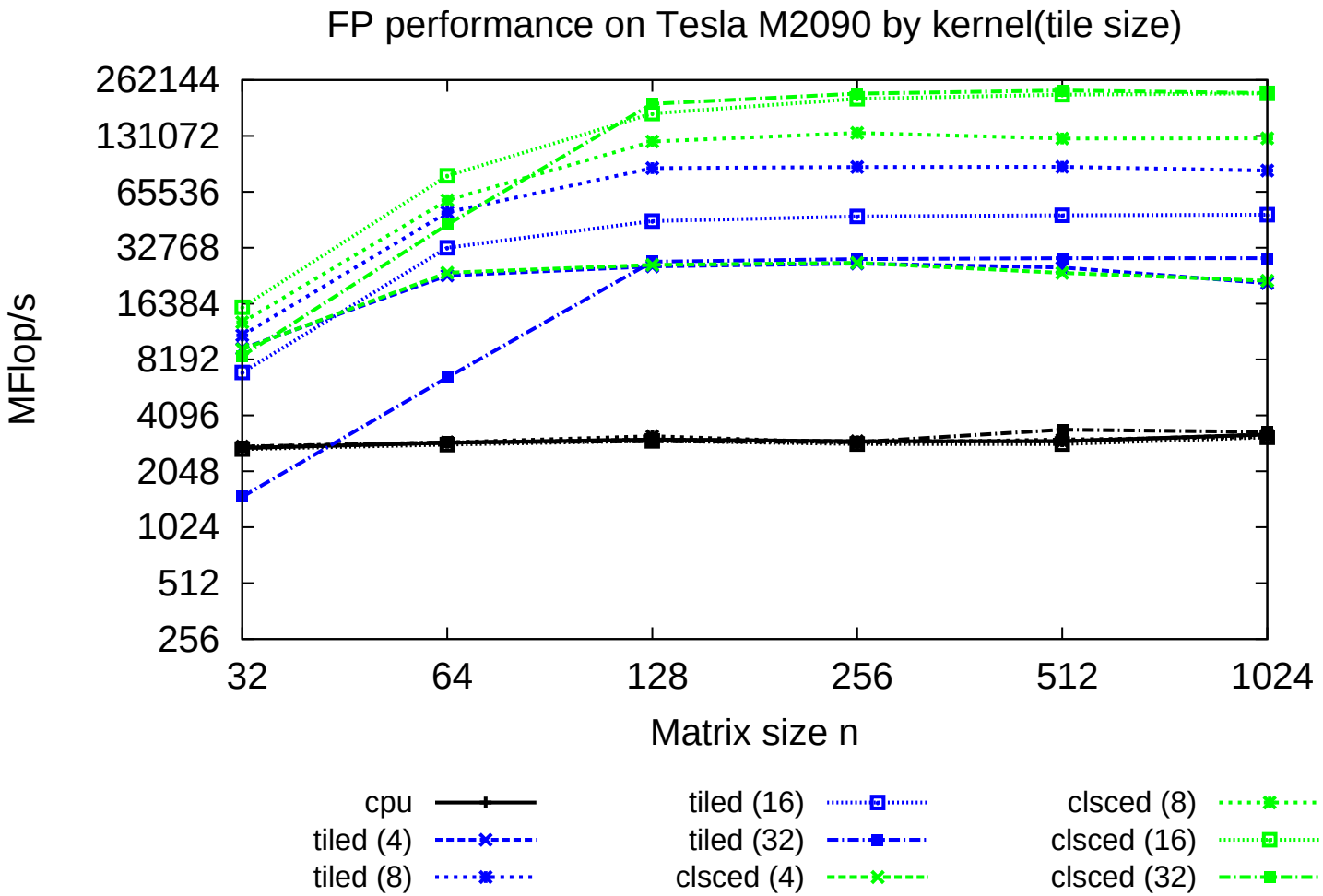
```
int i = blockIdx.y * TILE_SIZE + ty;
int k = blockIdx.x * TILE_SIZE + tx;
float Celem = 0;
for(int m=0; m < n/TILE_SIZE; m++) {
    Ads[ty][tx] = Ad[ i*n + m*TILE_SIZE+tx];
    Bds[ty][tx] = Bd[ (m*TILE_SIZE+ty)*n + k];
    __syncthreads();
    for(int j=0; j<TILE_SIZE; j++)
    Celem += Ads[ty][j]*Bds[j][tx];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

Performance measurements (v2)



Performance measurements (v2)

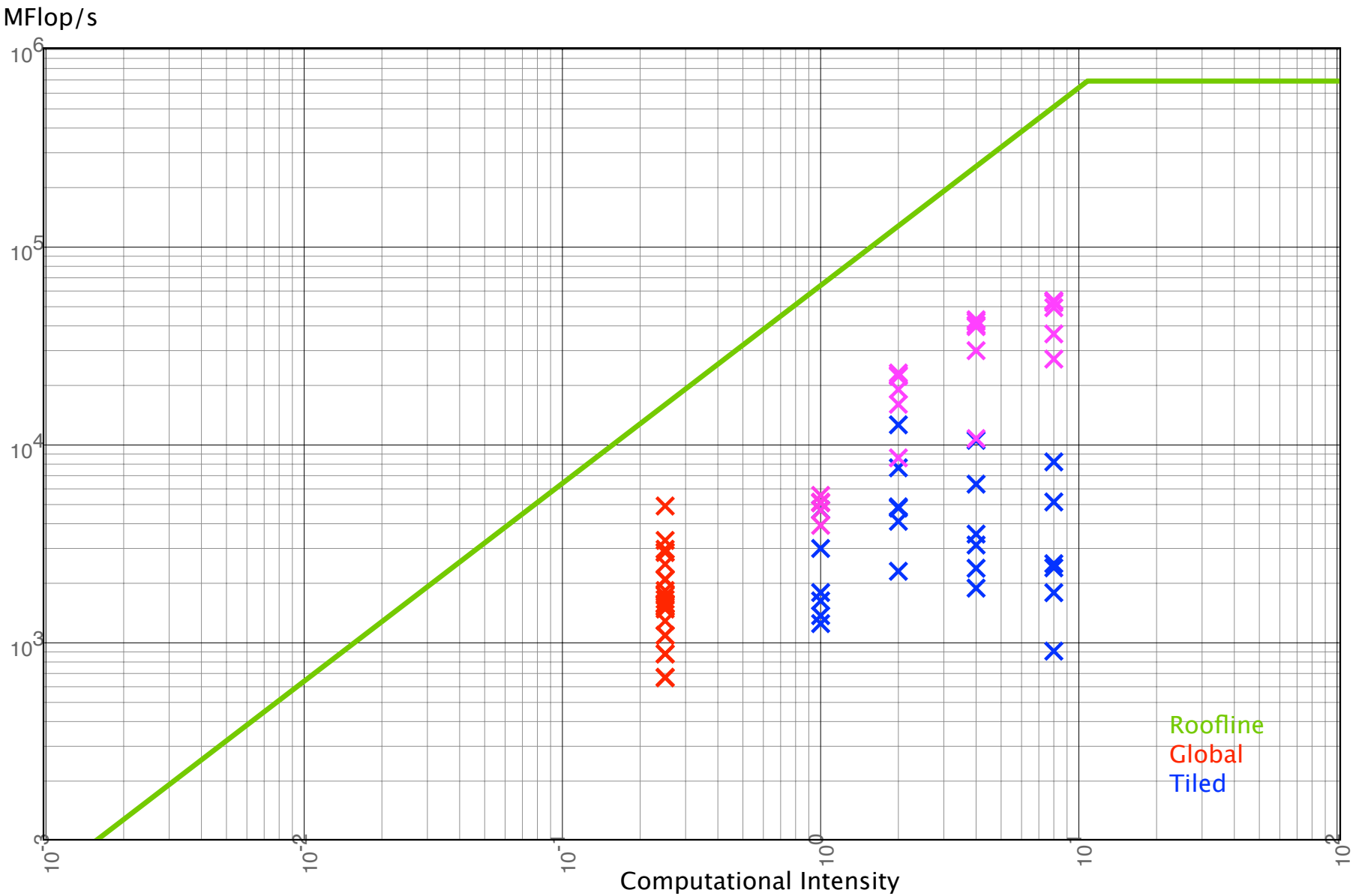
Let's test the change!



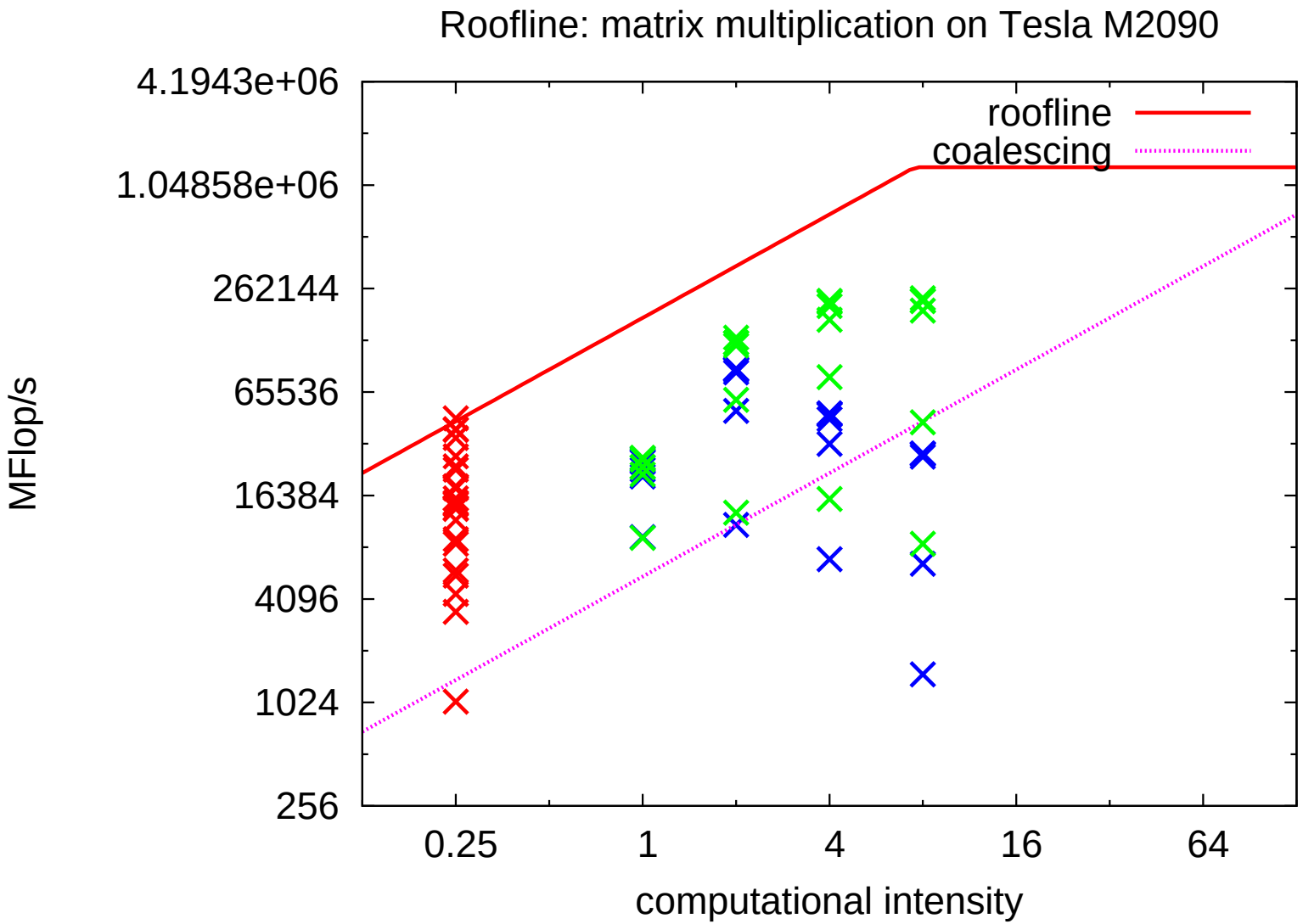
Roofline model

Task:

- Add a ceiling for coalescing to the roofline model (CUDA cc 1.2: $\leq$ 128 B per fetch).
- Insert points with the performance measurements of the coalesced kernel.


MFlop/s
10^6
10^5
10^4
10^3
10^2
10^-3
10^-2
10^-1
10^0
10^1
10^2
Computational Intensity
Roofline
Global
Tiled

Roofline model (v2)



Memory Latency for Tile Transfers

Recapitulate tiled matrix multiplication:

- tiles of 32×32 matrix elements
  $\rightarrow 32^2 = 1024$ threads per tile (also per thread block)
- thus: 32 warps (32 threads each) on 8 slots
- 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();
```

- due to memory latency global memory access takes multiple cycles
- warp scheduler underloaded when threads read data
- meanwhile other warps could do some 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

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

$\rightarrow$ Homework!

CUDA Profiler

Kernel analysis

Compute	branches, warp serialization
Bandwidth	bandwidth limitations, load/store sizes
Latency	occupancy