GPU Hardware Performance

Jiří Filipovič

Fall 2010

Global Memory Access Optimization

Performance of global memory becomes a bottleneck easily

- global memory bandwdith is low relatively to arithmetic performance of GPU (G200 $\geq$ 24 FLOPS/float, G100 $\geq$ 30)
- 400–600 cycles latency

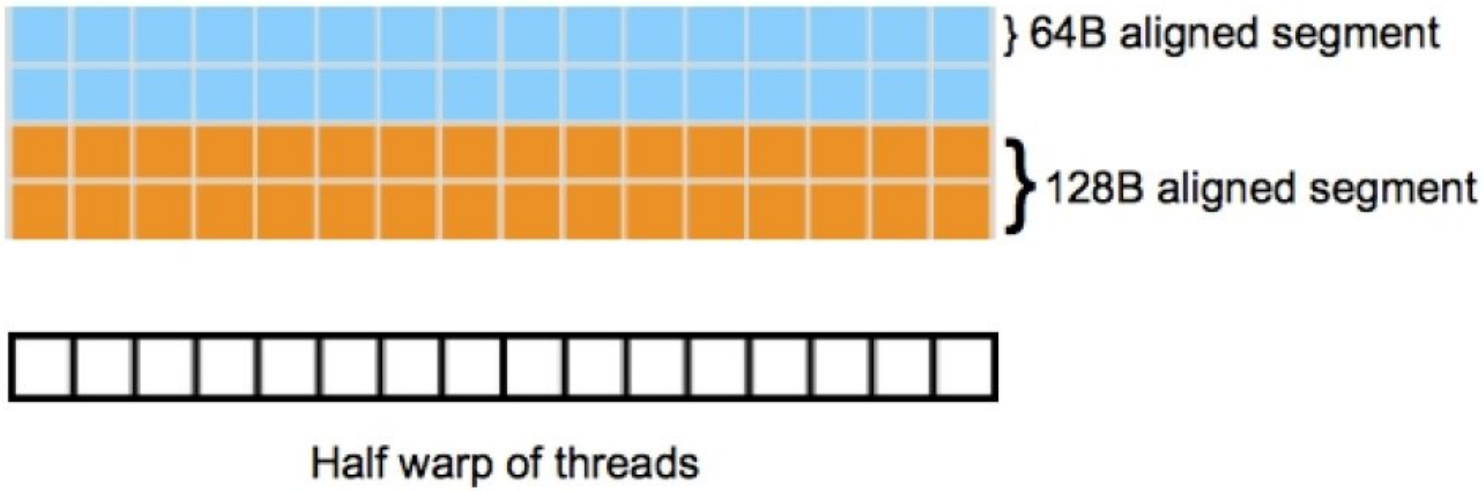
The throughput can be significantly worse with bad parallel access pattern

- the memory has to be accessed continuously (*coalescing*)
- use of just certain subset of memory regions should be avoided (*partition camping*)

Continous Memory Access (C. C. < 2.0)

GPU memory needs to be accessed in larger blocks for efficiency

- global memory is split into 64 B segments
- two of these segments are aggregated into 128 B segments



Continous Memory Access (C. C. < 2.0)

A half of a warp can transfer data using single transaction or one to two transactions when transactions when transferring a 128 B word

- it is necessary to use large words
- one memory transaction can transfer 32 B, 64 B, or 128 B words
- GPUs with c. c. ≤ 1.2
  - the accessed block has to begin at an address dividable by 16$\times$ data size
  - k-th thread has to access k-th block element
  - some threads needn't participate
- if these rules are not obeyed, each element is retrieved using a separate memory transaction

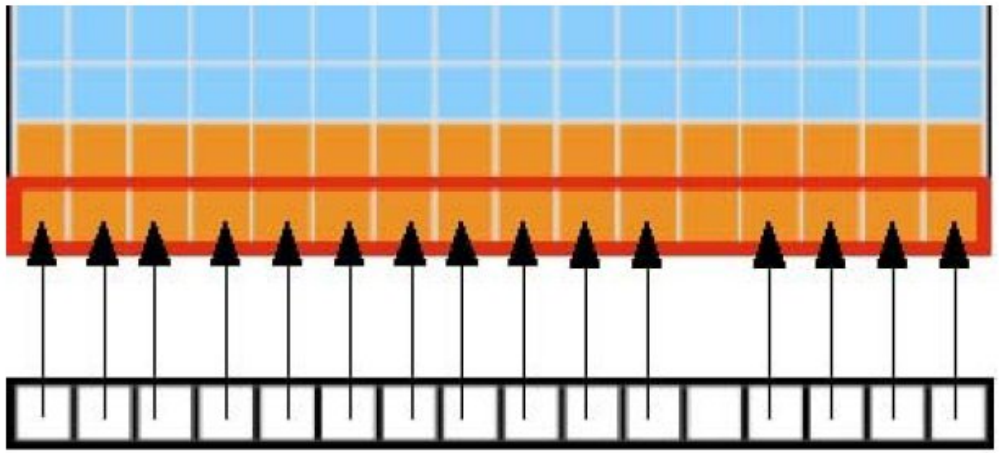
Continous Memory Access (C. C. < 2.0)

GPUs with c. c. ≥ 1.2 are less restrictive

- each transfer is split into 32 B, 64 B, or 128 B transactions in a way to serve all requests with the least number of transactions
- order of threads can be arbitrarily permuted w.r.t. transferred elements

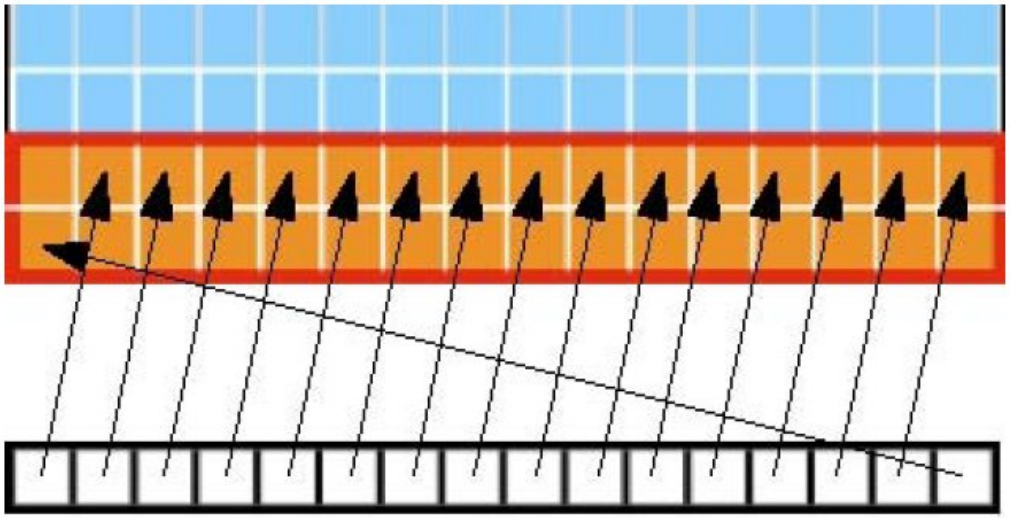
Continous Memory Access (C. C. < 2.0)

Threads are aligned, element block is continous, order is not permuted – continuous access on all GPUs



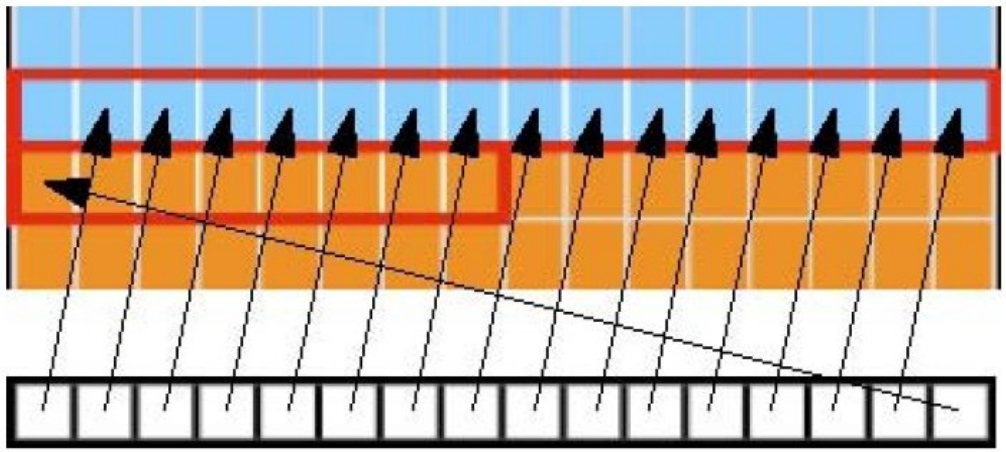
Unaligned Memory Access (C. C. < 2.0)

Threads **are not** aligned, element block is continuous, order is not permuted – one transaction on GPUs with c. c. ≥ 1.2



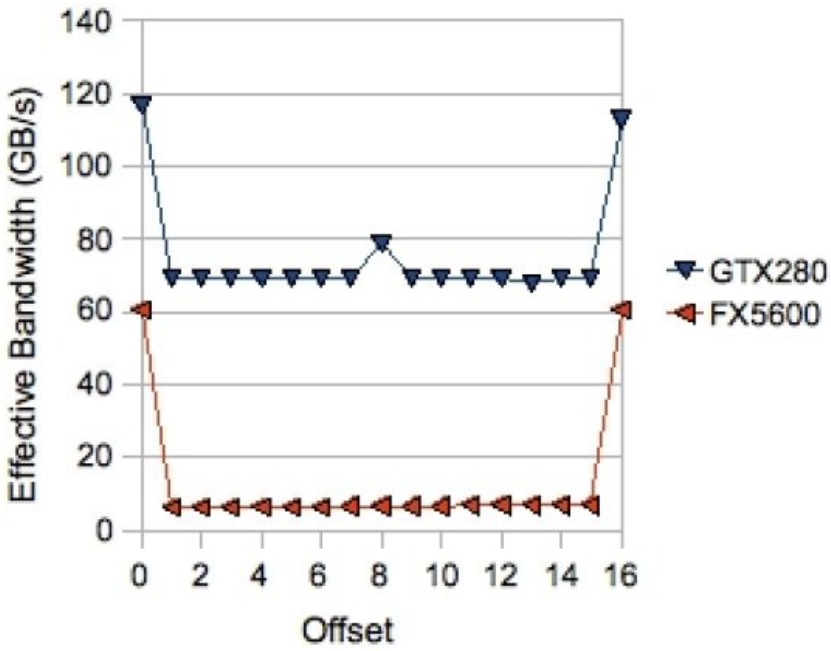
Unaligned Memory Access (C. C. < 2.0)

Similar case may result in a need for two transactions



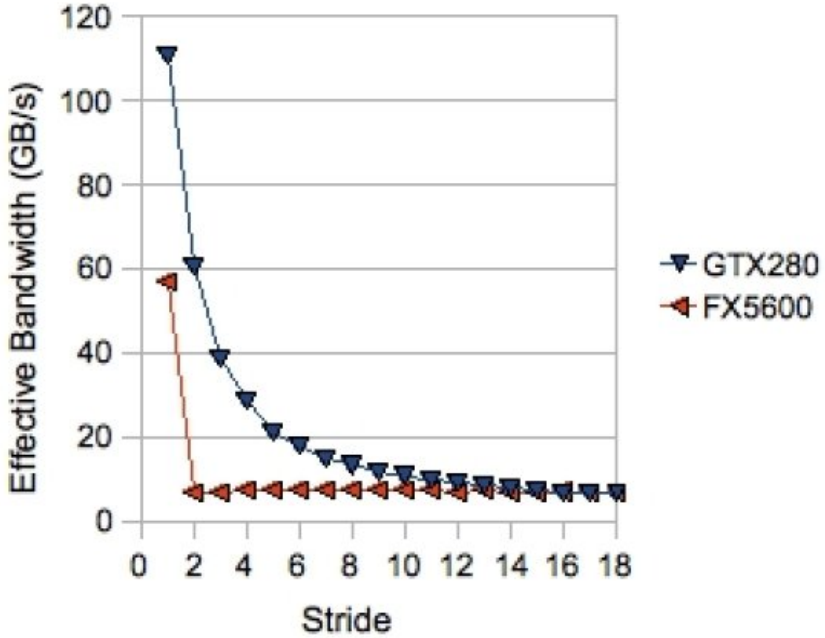
Unaligned Memory Access Performance (C. C. < 2.0)

Older GPUs perform smallest possible transfer (32 B) for each element, thus reducing performance to 1/8
Newer GPUs perform (c. c. ≥ 1.2) two transfers



Interleaved Memory Access Performance (C. C. < 2.0)

The bigger the spaces between elements, the bigger performance drop on GPUs with c. c. ≥ 1.2 – the effect is rather dramatic



Global Memory Access with Fermi (C. C. $\geq$ 2.0)

Fermi has L1 and L2 cache

- L1: 256 B per row, 16 kB or 48 kB per multiprocesor in total
- L2: 32 B per row, 768 kB on GPU in total

What are the advantages?

- more efficient programs with unpredictable data locality
- unaligned access – no slowdown in principle
- interleaved access – data needs to be used before it is flushed from the cache, otherwise the same or bigger problem as with c. c. $<$ 2.0 (L1 cache may be turned of to avoid overfetching)

Partition camping

- relevant for c. c. 1.x
- processors based on G80 have 6 regions, G200 have 8 regions of global memory
- the memory is split into 256 B regions
- even access among the regions is needed for maximum performance
  - among individual blocks
  - block are usually run in order given by their position in the grid
- if only part of regions is used, the resulting condition is called *partition camping*
- generally not as critical as the continuous access
- more tricky, problem size dependent, disguised from fine-grained perspective

HW Organization of Shared Memory

Shared memory is organized into memory banks, which can be accessed in parallel

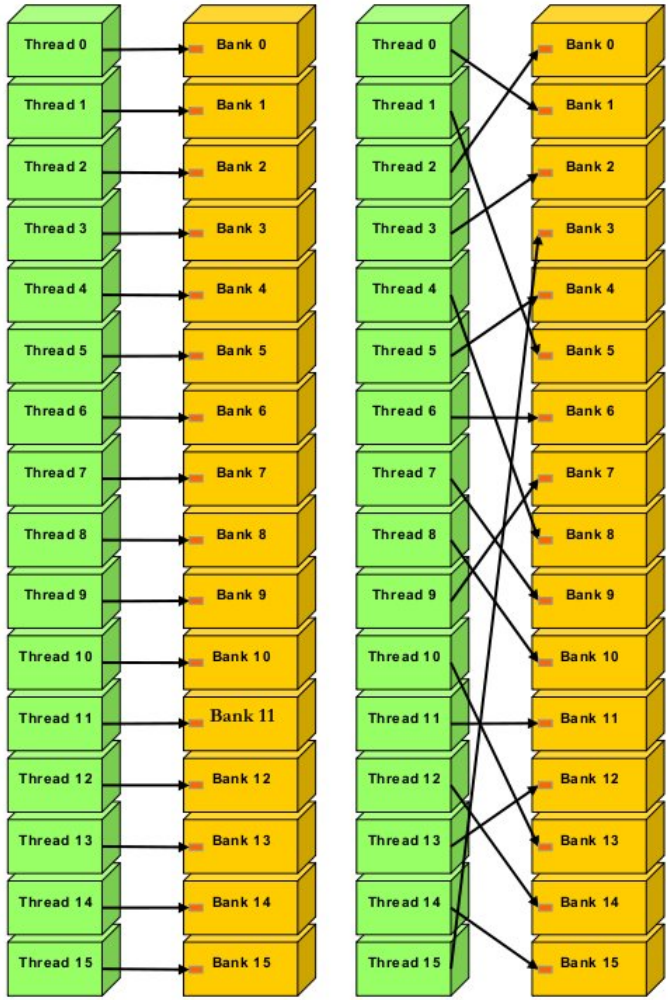
- c. c. 1.x 16 banks, c. c. 2.x 32 banks, memory space mapped in an interleaved way with 32 b shift
- to use full memory performance, we have to access data in different banks
- broadcast implemented – if all threads access the same data

Bank Conflict

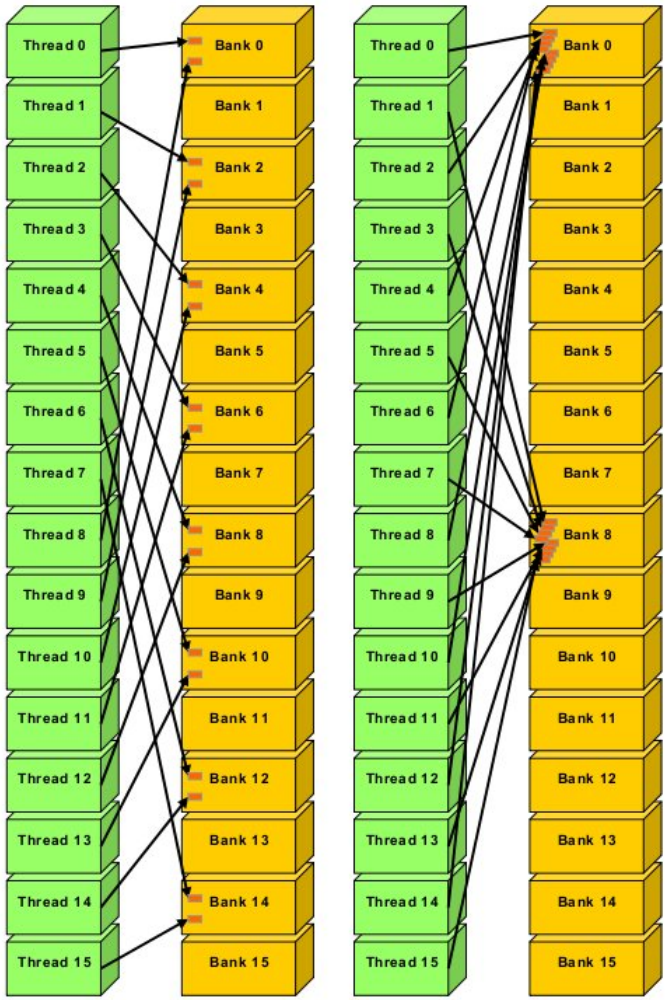
Bank conflict

- occurs when some threads in warp/half-warp access data in the same memory bank (except for when accessing exactly the same data)
- memory access gets serialized
- performance drop is proportional to number of parallel operations that the memory has to perform to serve a request
  - there is a difference if some threads access different data in a single bank and the same data in a single bank

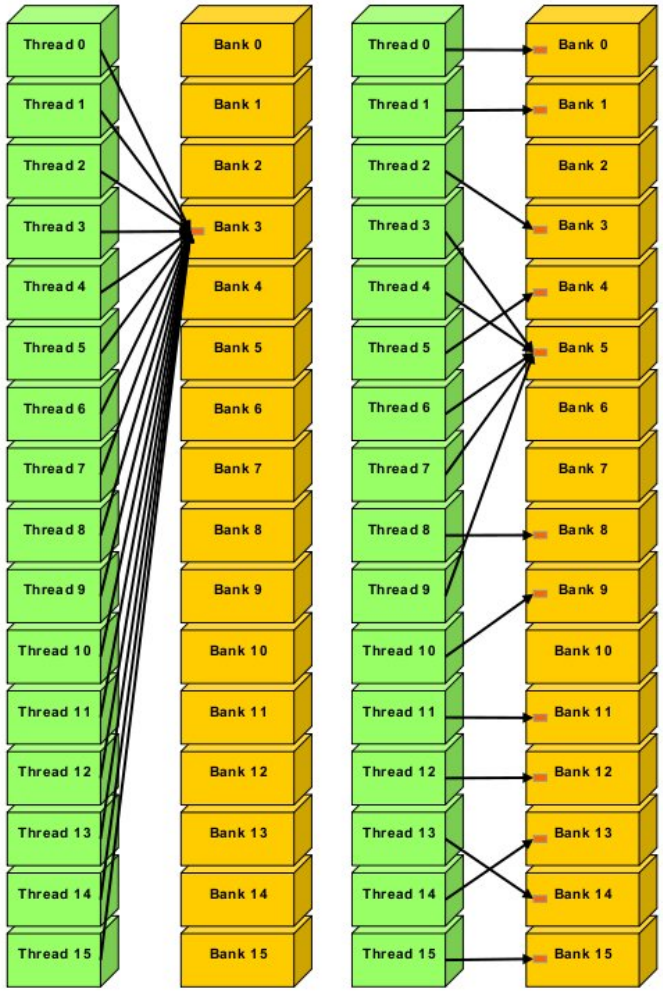
Access without Conflicts



n-Way Conflicts



Broadcast


Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
Thread 6
Thread 7
Thread 8
Thread 9
Thread 10
Thread 11
Thread 12
Thread 13
Thread 14
Thread 15
Bank 0
Bank 1
Bank 2
Bank 3
Bank 4
Bank 5
Bank 6
Bank 7
Bank 8
Bank 9
Bank 10
Bank 11
Bank 12
Bank 13
Bank 14
Bank 15
Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
Thread 6
Thread 7
Thread 8
Thread 9
Thread 10
Thread 11
Thread 12
Thread 13
Thread 14
Thread 15
Bank 0
Bank 1
Bank 2
Bank 3
Bank 4
Bank 5
Bank 6
Bank 7
Bank 8
Bank 9
Bank 10
Bank 11
Bank 12
Bank 13
Bank 14
Bank 15

Access Patterns

Alignment is not needed, bank conflicts not generated

```
int x = s[threadIdx.x + offset];
```

Interleaving does not create conflicts if c is odd

```
int x = s[threadIdx.x * c];
```

Access to the same variable never generates conflicts on c. c. 2.x, while on 1.x only if thread count accessing the variable is multiple of 16

```
int x = s[threadIdx.x % c];
```

Other Memory Types

Transfers between host and GPU memory

- need to be minimized (often at cost of decreasing efficiency of computation on GPU)
- may be accelerated using page-locked memory
- it is more efficient to transfer large blocks at once
- computations and memory transfers should be overlapped

Texture memory

- designed to reduce number of transfers from the global memory
- works well for aligned access
- does not help if latency is the bottleneck
- may simplify addressing or add filtering

Other Memory Types

Constant memory

- as fast as registers if the same value is read
- performance decreases linearly with number of different values read

Registers

- read-after-write latency, disguised if at least 192 threads are running for c. c. 1.x or at least 768 threads are running for c. c. 2.x
- possible bank conflicts even in registers
  - compiler tries to avoid them
  - we can make life easier for the compiler if we set block size to multiple of 64

Matrix Transposition

From theoretical perspective:

- a trivial problem
- trivial parallelization
- trivially limited by the memory throughput (no arithmetic ops done)

```
__global__ void mtran(float *odata, float* idata, int n){
  int x = blockIdx.x * blockDim.x + threadIdx.x;
  int y = blockIdx.y * blockDim.y + threadIdx.y;
  odata[x*n + y] = idata[y*n + x];
}
```

Performance

When running the code on GeForce GTX 280 with large enough matrix 4000×4000, the throughput will be $5.3\,\mathrm{GB/s}$
Where's the problem?

Performance

When running the code on GeForce GTX 280 with large enough matrix 4000×4000, the throughput will be 5.3 GB/s
Where's the problem? Access to `odata` is interleaved After transposition modification:

```
odata[y*n + x] = idata[y*n + x];
```

the throughput is 112.4 GB/s. If `idata` is accessed in an interleaved way too, the resulting throughput would be 2.7 GB/s.

On Removing Interleaving

The matrix can be processed per block

- we read the block into he shared memory row-wise
- we will store its transposition into the global memory row-wise
- thus having both reading and writing without interleaving

On Removing Interleaving

The matrix can be processed per block

- we read the block into he shared memory row-wise
- we will store its transposition into the global memory row-wise
- thus having both reading and writing without interleaving

What size of blocks should be used?

- lets consider square blocks
- for aligned reading, the row size has to be multiple of 16
- we can consider block sizes of 16×16, 32×32, and 48×48 because of shared memory size limitations
- best size can be determined experimentally

Block Transposition

```
__global__ void mtran_coalesced(float *odata, float *idata, int n)
  __shared__ float tile[TILE_DIM][TILE_DIM];

  int x = blockIdx.x * TILE_DIM + threadIdx.x;
  int y = blockIdx.y * TILE_DIM + threadIdx.y;
  int index_in = x + y*n;
  x = blockIdx.y * TILE_DIM + threadIdx.x;
  y = blockIdx.x * TILE_DIM + threadIdx.y;
  int index_out = x + y*n;

  for (int i = 0; i < TILE_DIM; i += BLOCK_ROWS)
    tile[threadIdx.y+i][threadIdx.x] = idata[index_in+i*n];

  __syncthreads();

  for (int i = 0; i < TILE_DIM; i += BLOCK_ROWS)
    odata[index_out+i*n] = tile[threadIdx.x][threadIdx.y+i];
}
```

Performance

The highest performance was measured for 32×32 block size and 32×8 thread block size – 75.1 GB/s

Performance

The highest performance was measured for 32×32 block size and 32×8 thread block size – 75.1 GB/s

- that's significantly better but still far from simple copying

Performance

The highest performance was measured for 32×32 block size and 32×8 thread block size – 75.1 GB/s

- that's significantly better but still far from simple copying
- the kernel is more complex, contains synchronization
  - we need to figure out whether we got the maximum or there's still a problem somewhere

Performance

The highest performance was measured for 32×32 block size and 32×8 thread block size – 75.1 GB/s

- that's significantly better but still far from simple copying
- the kernel is more complex, contains synchronization
  - we need to figure out whether we got the maximum or there's still a problem somewhere
- if we only copy within the blocks, we get 94.9GB/s
  - something is still sub-optimal

Shared Memory

When reading from the global memory, we write into the shared memory row-wise

```
tile[threadIdx.y+i][threadIdx.x] = idata[index_in+i*n];
```

Shared Memory

When reading from the global memory, we write into the shared memory row-wise

```
tile[threadIdx.y+i][threadIdx.x] = idata[index_in+i*n];
```

When writing to the global memory, we read from the shared memory column-wise

```
odata[index_out+i*n] = tile[threadIdx.x][threadIdx.y+i];
```

That's reading with interleaving which is multiple of 16, the whole column is in a single memory bank – thus creaing 16-way bank conflict

Shared Memory

When reading from the global memory, we write into the shared memory row-wise

```
tile[threadIdx.y+i][threadIdx.x] = idata[index_in+i*n];
```

When writing to the global memory, we read from the shared memory column-wise

```
odata[index_out+i*n] = tile[threadIdx.x][threadIdx.y+i];
```

That's reading with interleaving which is multiple of 16, the whole column is in a single memory bank – thus creaing 16-way bank conflict
A solution is padding:

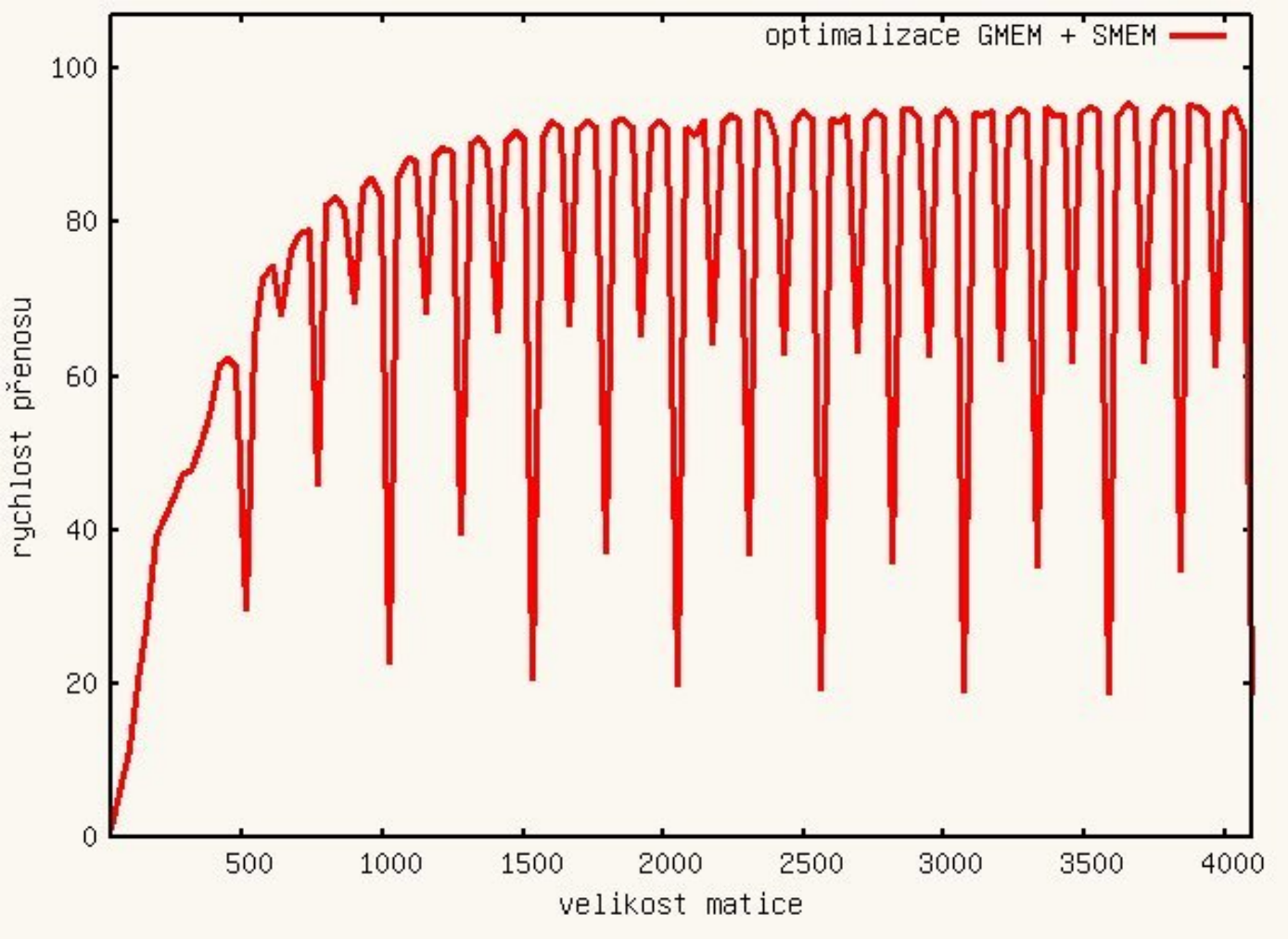
```
__shared__ float tile[TILE_DIM][TILE_DIM + 1];
```

Performance

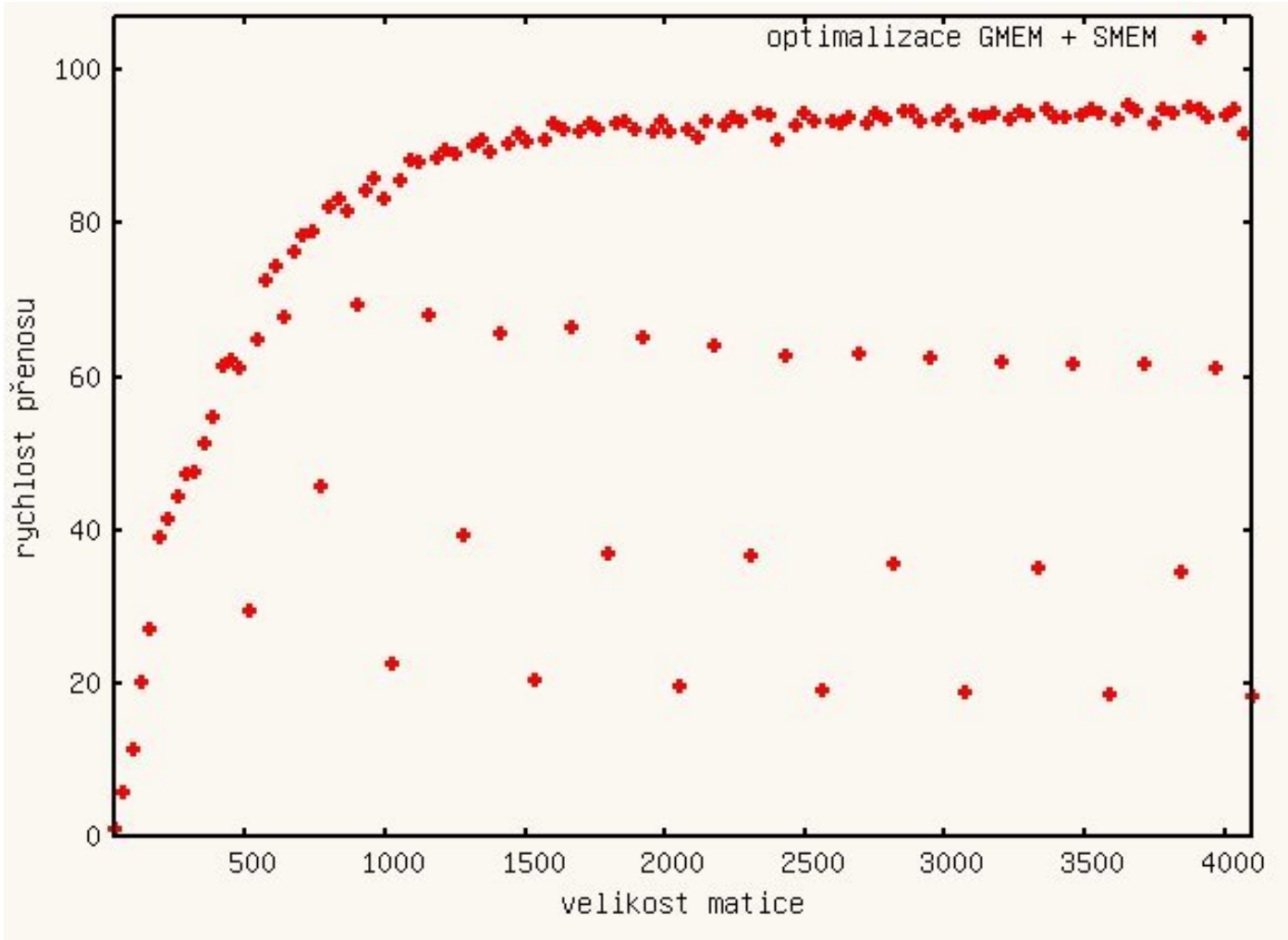
Now our implementations shows 93.4 GB/s.

- as good as simple copying
- it seems we can't do much better for given matrix
- beware of different input data sizes (see partition camping)

Performance



Performance



Performance Drops

The performance drops for some size and the behavior is regular

Performance Drops

The performance drops for some size and the behavior is regular

- for matrices sized multiple of 512, we only get 19 GB/s
- for other matrices sized multiple of 256, we only get 35 GB/s
- for other matrices sized multiple of 128, we only get 62 GB/s

Performance Drops

One memory region has width of 2 blocks (256 B / 4 B per float, 32 floats in a block). If we analyze block placement w.r.t. matrix size, we learn that

- with multiple of 512 size, the blocks are in the same columns in the same region
- with multiple of 256 size, each column is at most in two regions
- with multiple of 128, each column is at most in four regions

We have discovered partition camping.

How to Remove Partition Camping?

We can pad "fake data" and avoid bad matrix sizes.

- it makes further work on the algorithm more complicated
- it occupies more memory

How to Remove Partition Camping?

We can pad "fake data" and avoid bad matrix sizes.

- it makes further work on the algorithm more complicated
- it occupies more memory

We can change the mapping of thread blocks id's on matrix blocks

- diagonal mapping ensures access to different regions

```
int blockIdx_y = blockIdx.x;
int blockIdx_x = (blockIdx.x+blockIdx.y) % gridDim.x;
```

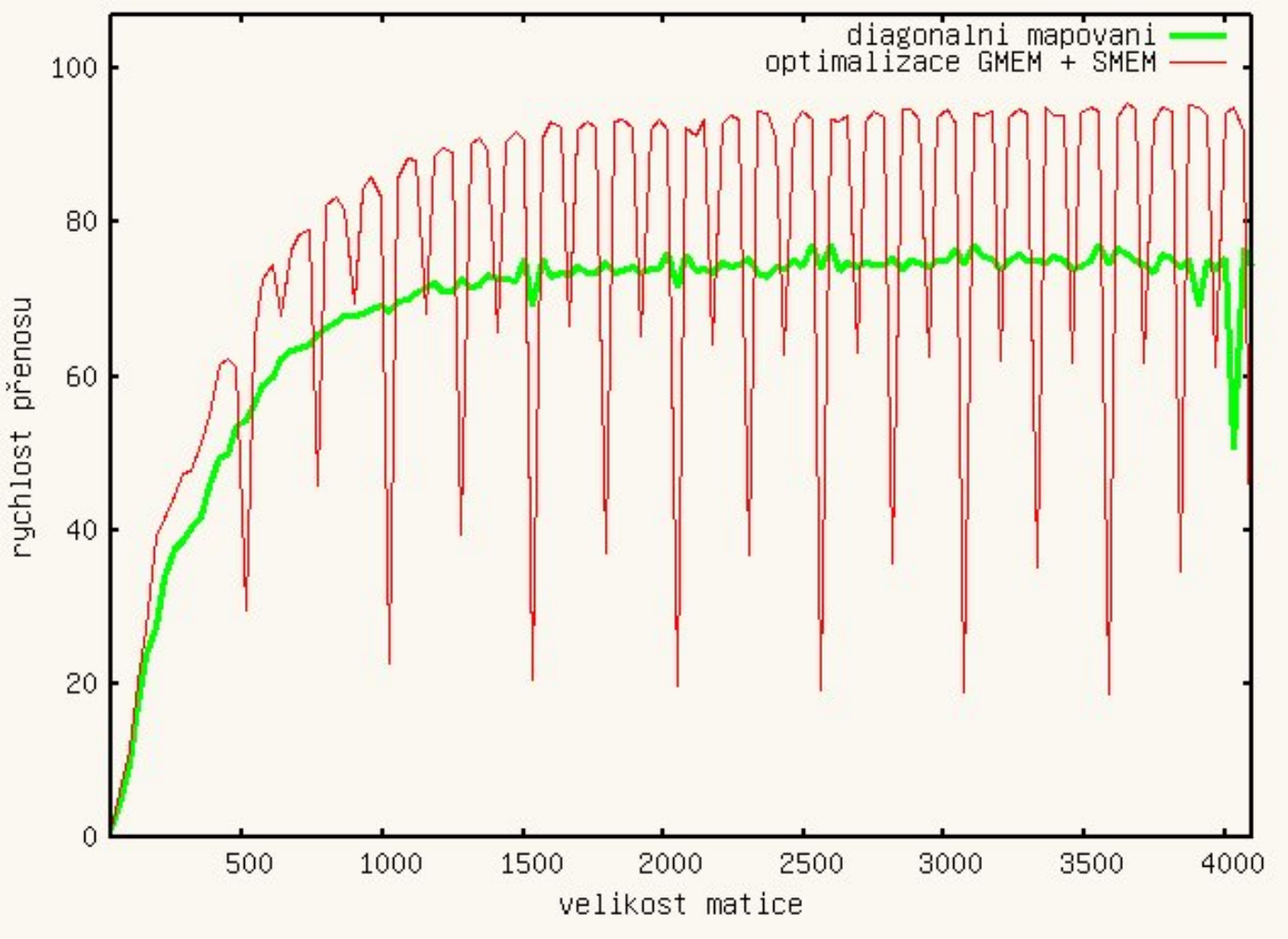
Performance

New implementation gives 80 GB/s

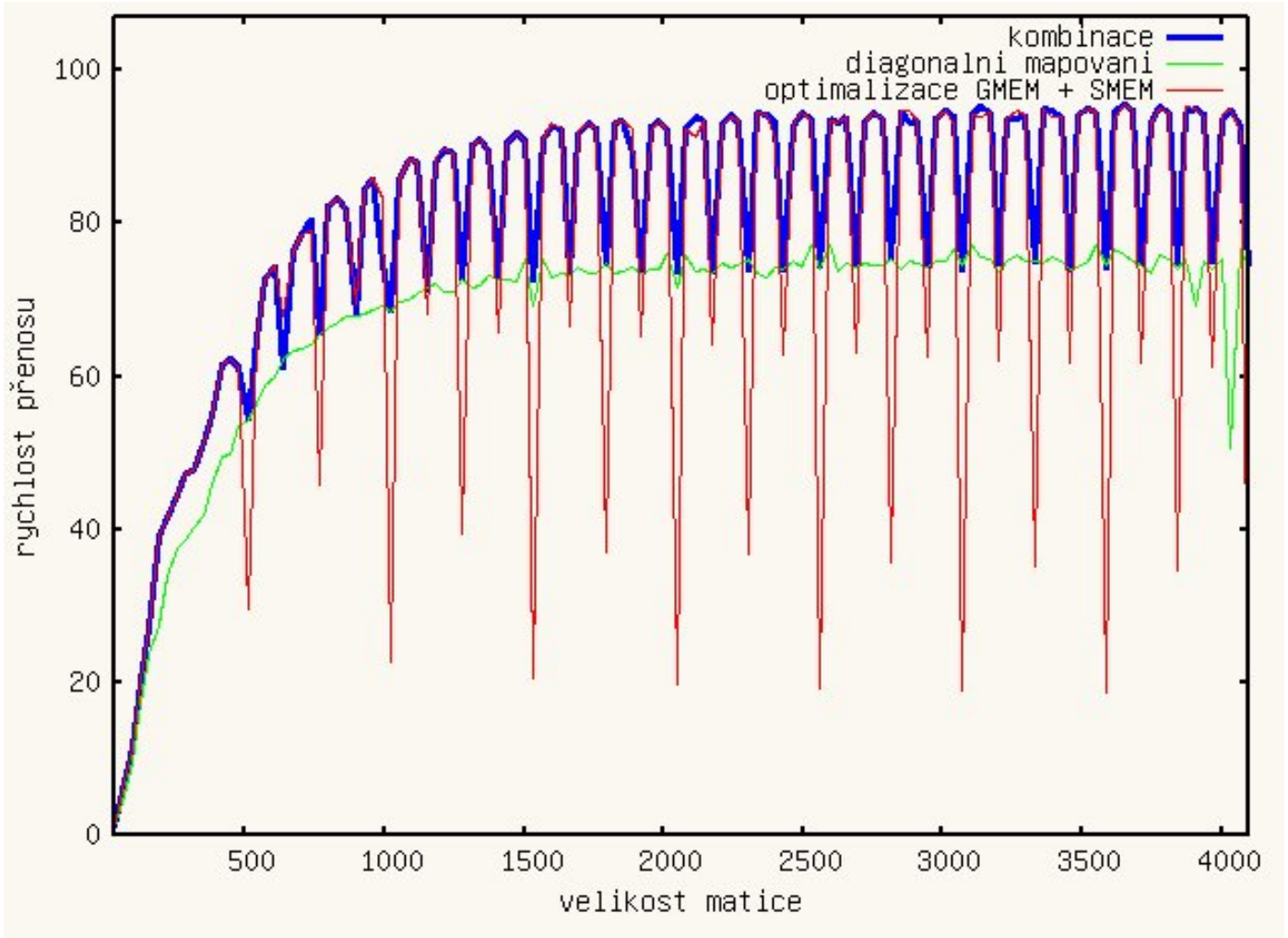
- performance doesn't drop where we saw it previously
- for matrix size of multiple of 128 still worse then the original implementation
  - the algorithm is more complex
- we can use it only for the problematic data sizes

For given problem, there may not be (and often there is not) an ideal algorithm for the whole input data size range. It is necessary to benchmark as not all the problems are easily revealed just by looking at the code.

Performance



Performance



Performance Summary

All optimizations were only toward better accommodation of HW properties

- still we got 17.6× speedup
- when creating an algorithm, it is necessary to understand HW limitations
- otherwise we wouldn't have to develop specifically for GPUs – developing a good sequential algorithm would have been just fine...

Optimizations Effects

Beware of optimization effects

- if we took 4096×4096 matrices instead of 4000×4000, the memory bank conflict removal would have been just marginal
- after removing partition camping, the memory bank conflicts would have appeared
- thus it makes sense to go from more general/substantial optimizations to the less general ones
- if some (provably correct) optimization does not result in performance increase, we need to analyze, what the algorithm is limited by

Processing of Instructions

Processing of instructions on a multiprocessor (c. c. 1.x)

- there are 8 SP cores and 2 SFU cores
- if the SP and SPU instruction processing is not overlapped, the multiprocessor can process up to 8 instructions per cycle
  - one warp is thus done in 4 or more cycles
- some instructions are significantly slowe
- instruction processing knowledge helps us to design optimal code

Floating Point Operations

GPU is graphical HW primarily

- graphical operations mostly use floating point numbers
- efficiently implemented in GPUs
- newer GPUs (c. c. ≥ 1.3) can work in double precision while older ones in single precision only
- some arithmetic operations are used very frequently in graphics
  - GPU implements them in HW
  - HW implementation provides lower precision (not in issue for lots of applications)
  - differentiated using "__" prefix

Aritmetic Operations

Floating point operations (throughput on an MP)

- addition, multiplication 8 (1.x), 32 (2.0), 48 (2.1)
- multiplication and addition may be combined into a single MAD instruction for c. c. 1.x
  - lower precision
  - 1 cycle speed on SP
  - *__fadd_rn()* and *__fmul_rn()* may be used to enforce avoiding MAD instruction during compilation
- MAD is replaced by FMAD for c. c. 2.x (same speed, higher precision)
- 64b versions 1/8 (1.3), 1/2 (2.0), 1/12 (2.1)
- inverse value 2 (1.x), 4 (2.0) a 8 (2.1)
- division is relatively slower (by 1.23 on average for c. c. 1.x)
  - faster variant *__fdividef(x, y)* 1.6 (c. c. 1.x)
- inverted square root 2 (1.x), 4 (2.0) a 8 (2.1)
- type conversion 8 (c.c. 1.x), 16 (c.c. 2.x)

Aritmetic Operations

Floating point operations

- *__sinf(x)*, *__cosf(x)*, *__expf(x)* 2 (c.c. 1.x), 4 (c.c. 2.0), 8 (c.c. 1.2)
- *sinf(x)*, *cosf(x)*, *expf(x)* more precise but an order of magnitude slower
- other operations with different speed and precision trade-offs are implemented, see CUDA manual

Integer operations

- addition as for the floating point ops
- multiplication on c. c. 1.x 2 instructions on an MP
  - *__mul24(x, y)* a *__umul24(x, y)* 8 instructions
- multiplication on c. c. 2.x is as fast as floating point ops, 24-bit version is slow
- division and modulo is very slow, but if n is power of 2, we can utilize
  - i/n is equivalent to $i >> log2(n)$
  - $i\%n$ is equivalent to $i\&(n-1)$

Loops

Small loops have significant overhead

- jumps need to be implemented
- it is necessary to update control variable
- significant part of instructions may be pointer arithmetics

Loop unrolling is an option

- partially may be done by the compiler
- we can do manual unrolling or use *#pragma unroll*

Other Instructions

Other common instructions are done at the basic speed (i.e., correspond to number of SPs)

- comparison
- bit operations
- memory access instructions (given the limitations discussed earlier and memory latency/bandwidth)
  - the offset may be register value + constant
- synchronization (unless we get blocked)

Beware of Shared Memory

If memory bank conflict is avoided, the shared memory is as fast as registers

But beware

- instructions can work with only one operand in the shared memory
- if more than one operands in shared memory are used for one instruction, explicit load/store is necessary
- MAD instructions run slower (c.c. 1.x)
  - $a + s[i]$ 4 cycles per warp
  - $a + a * s[i]$ 5 cycles per warp
  - $a + b * s[i]$ cycles per warp
- these details are not published by nVidia (revealed through measurements)
- may change with future GPU generations, interested only for really critical code

C for CUDA Compilation

Device code can be compiled into PTX assembler and binary files

- PTX is intermediate code, does not correspond directly to GPU instructions
  - easier to read
  - harder to figure out what really happens on GPU
- native GPU code compiler is to be released

Binary files may be disassembled using *decuda* tool

- third party product
- may not work completely reliably
- still quite useful