

Code optimization for GPUs

Jiří Filipovič

spring 2023

GPU Parallelism

Parallel algorithms need to be designed w.r.t. the parallelism available in the HW

- GPU: array of SIMT multiprocessors working using shared memory

Decomposition for GPU

- coarse-grained decomposition of the problem into the parts that don't need intensive communication
- fine-grained decomposition similar to vectorization (but SIMT is more flexible)

SIMT

A multiprocessor of G80 has one unit executing an instruction

- all 8 SPs have to execute the same instruction
- new instruction is executed every 4 cycles
- 32 threads (so called *warp*) need to execute the same instruction, warp size is fixed for all existing CUDA hardware

How about code branching?

- if different parts of a warp perform different instructions, they are serialized
- decreases performance—should be avoided

The multiprocessor is thus (nearly) MIMD (Multiple-Instruction Multiple-Thread) from programmer's perspective and SIMT (Single-Instruction Multiple-Thread) from performance perspective.

Thread Properties

GPU threads are very lightweight compared to CPU threads.

- their run time can be very short (even tens of instructions)
- there should be many of them
- they should not use large amount of resources

Threads are aggregated into blocks

- all threads of the block always run on the same multiprocessor (multiple blocks can run at one multiprocessor)
- having sufficient number of blocks is substantial to achieve good scalability

Number of threads and thread blocks per multiprocessor is limited.

Memory Latency Masking

Memory has latency

- global memory has high latency (hundreds of cycles)
- registers and shared memory have read-after-write latency

Memory latency hiding is different from CPU

- no instructions are executed out of order (but ILP can be exploited by forcing finalization of load instruction just before loaded data are needed)
- no or limited cache

When a warp waits for data from memory, another warp may be executed

- allows memory latency hiding
- requires execution of more threads than the number of GPU cores
- thread execution scheduling and switching is implemented directly in HW without overhead

Global Memory Access Optimization

Performance of global memory becomes a bottleneck easily

- global memory bandwidth is low relatively to arithmetic performance of GPU (GT200 \geq 24 FLOPS/float, GF100 \geq 30, GK110 \geq 62, GM200 \geq 73, GP100 \geq 53, GV100 \geq 67, TU102 \geq 76, GA100 \geq 50, AD102 \geq 72, GH100 \geq 20, but \geq 295 with 32-bit tensor ops)
- 400–600 cycles latency

The throughput can be significantly worse with bad parallel access pattern

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

Coalesced Memory Access (C. C. < 2.0)

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

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

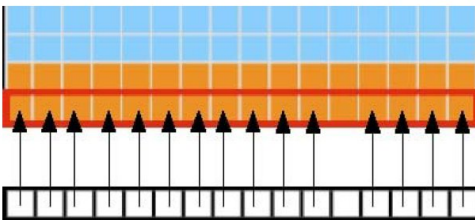
Coalesced Memory Access (C. C. < 2.0)

GPUs with $c. c. \geq 1.2$ 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

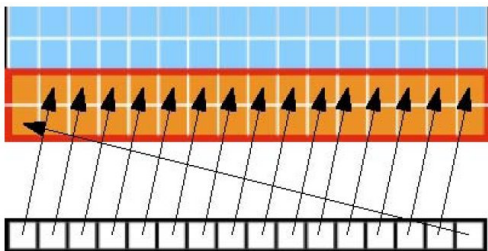
Coalesced Memory Access (C. C. < 2.0)

Threads are aligned, element block is contiguous, order is not permuted – coalesced access on all GPUs



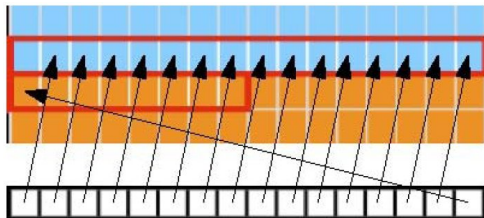
Unaligned Memory Access (C. C. < 2.0)

Threads **are not** aligned, contiguous elements accessed, order is not permuted – one transaction on GPUs with c. c. ≥ 1.2



Unaligned Memory Access (C. C. < 2.0)

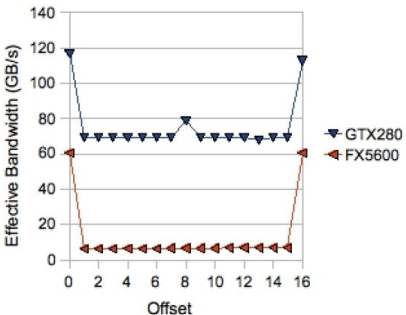
Similar case may result in a need for two transactions



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

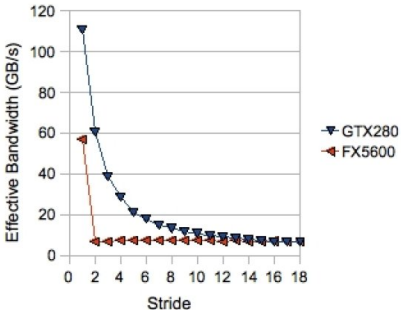
Older GPUs perform smallest possible transfer (32 B) for each element, thus reducing performance to 1/8

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. = 2.x)

Fermi has L1 and L2 cache

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

What are the advantages?

- more efficient programs with unpredictable data locality
- more efficient when shared memory is not used from some reason
- 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)

HW Organization of Shared Memory

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

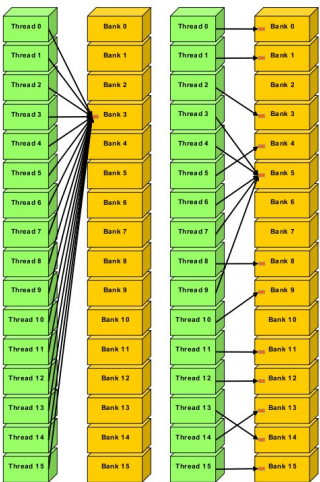
- c. c. 1.x 16 banks, c. c. ≥ 2.0 32 banks
- memory space mapped in an interleaved way with 32 b shift or 64 b shift (c.c. 3.x)
- to use full memory performance, we have to access data in different banks
- broadcast implemented – if all threads access the same data

Bank Conflict

Bank conflict

- occurs when some threads in warp/half-warp access data in the same memory bank with several exceptions
 - threads access exactly the same data
 - threads access different half-words of 64 b word (c.c. 3.x)
- when occurs, memory access gets serialized
- performance drop is proportional to number of parallel operations that the memory has to perform to serve a request

Broadcast



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, for $c.c. \geq 3.0$ no conflict if $c = 2$ and 32 b numbers are accessed

```
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];
```

CPU-GPU memory transfers

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

Matrix Transposition

From theoretical perspective:

- a trivial problem
- a 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 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 modification (copy instead of transpose matrices):

```
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.

Removing Interleaving

The matrix can be processed per tiles

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

Removing Interleaving

The matrix can be processed per tiles

- we read the tile into the 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 tiles should be used?

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

Tiled 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 tile size and 32×8 thread block size – **75.1 GB/s**

Performance

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

- that's significantly better but still less than simple copying

Performance

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

- that's significantly better but still less than 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 tile size and 32×8 thread block size – **75.1 GB/s**

- that's significantly better but still less than 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 creating **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 creating **16-way bank conflict**.

A solution is padding:

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


Loops

Small loops have significant overhead

- jumps
- conditions
- control variable updates
- significant part of instructions may be pointer arithmetics
- low ILP

Loop unrolling is an option

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

Vector Reduction

Let v be the vector of size n . We want to compute $x = \sum_{i=1}^n v_i$.

Vector Reduction

Let v be the vector of size n . We want to compute $x = \sum_{i=1}^n v_i$.
C code (not very reasonable for floats)

```
int x = 0;
for (int i = 0; i < n; i++)
    x += v[i];
```

There is flow dependency across iterations.

Vector Reduction

Let v be the vector of size n . We want to compute $x = \sum_{i=1}^n v_i$.
C code (not very reasonable for floats)

```
int x = 0;
for (int i = 0; i < n; i++)
    x += v[i];
```

There is flow dependency across iterations.

- we cannot compute completely parallel
- addition is (at least in theory :-)) associative
- so, we do not need to add numbers in sequential order

Parallel Algorithm

The sequential algorithm performs seven steps:

$$(((((((v_1 + v_2) + v_3) + v_4) + v_5) + v_6) + v_7) + v_8$$

Parallel Algorithm

The sequential algorithm performs seven steps:

$$(((((((v_1 + v_2) + v_3) + v_4) + v_5) + v_6) + v_7) + v_8)$$

Addition is associative... so let's reorder brackets:

$$((v_1 + v_2) + (v_3 + v_4)) + ((v_5 + v_6) + (v_7 + v_8))$$

We can work in parallel now:

- four additions in the first step
- two additions in the second step
- one addition in the third step

In summary, we perform $n - 1$ additions in $\log_2 n$ parallel steps!

Parallel Algorithm

We have found the parallel algorithm

- the same number of additions as the serial algorithm
- in logarithmic time (if we have enough cores)

We add results of previous additions

- flow-dependency across threads
- we need global barrier

Naive Approach

The simplest scheme of the algorithm:

- for even i , $i < n$ perform $v[i] += v[i+1]$
- repeat for $n \neq 2$ until $n > 1$

The performance is not ideal

- $2n$ numbers loaded from global memory
- n numbers stored to global memory
- $\log_2 n$ kernel invocations

We have three memory accesses to one arithmetics operation and considerable kernel invocation overhead.

Exploiting Data Locality

We can add more than pairs during single kernel call.

- each block bx loads m numbers into shared memory
- it reduces the input (in shared memory in $\log_2 m$ steps)
- it stores only one number containing $\sum_{i=m \cdot bx}^{m \cdot bx + m} v_i$

Reduces both memory transfers and number of kernel invocations

- number of loads: $n + \frac{n}{m} + \frac{n}{m^2} + \dots + \frac{n}{m^{\log_m n}} = (n - 1) \frac{m}{m - 1}$
- approximately $n + \frac{n}{m}$ numbers read, $\frac{n}{m}$ written
- $\log_m n$ kernel invocations

Implementation 1

```

__global__ void reduce1(int *v){
    extern __shared__ int sv[];

    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sv[tid] = v[i];
    __syncthreads();

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

    if (tid == 0)
        v[blockIdx.x] = sv[0];
}

```

Performance

Beware modulo operation.

High degree of divergence

- during the first iteration, only half of threads is working
- during the second iteration, only quarter of threads is working
- etc.

Performance on GTX 280: 3.77 GB/s (0.94 MElem/s).

Implementation 2

We will modify indexation

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

Performance: 8.33 GB/s (2.08 MElem/s).

The code is free of modulo and divergence, but generates shared memory bank conflicts.

Implementation 3

So we can try another indexing...

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

No divergence and no conflicts.

Performance 16.34 GB/s (4.08 MElem/s).

Half of threads do not compute...

Implementation 4

We can add numbers during loading them from global memory.

```
unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;  
sv[tid] = v[i] + v[i+blockDim.x];
```

Performance 27.16 GB/s (6.79 MElem/s).

There is no problem with data access, but the performance is still low – we will focus to instructions.

Implementation 5

The number of active threads decreases during computation in shared memory.

- in the last six iterations, only the last warp is active
- the warp is synchronized implicitly on GPUs with c.c. < 7.0 , so we do not need `__syncthreads()`
 - we need volatile variable in this case
- condition `if(tid < s)` does not spare any computation

So we can unroll the last warp...

Implementation 5

```

float mySum = 0;

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

if (tid < 32){
    volatile float *s = sv;
    s[tid] = mySum = mySum + s[tid + 32]; // __syncthreads();
    s[tid] = mySum = mySum + s[tid + 16]; // __syncthreads();
    s[tid] = mySum = mySum + s[tid + 8]; // __syncthreads();
    s[tid] = mySum = mySum + s[tid + 4]; // __syncthreads();
    s[tid] = mySum = mySum + s[tid + 2]; // __syncthreads();
    s[tid] = mySum = mySum + s[tid + 1];
}

```

We save time in all warps (the last warp is simpler, others exits earlier from the for loop).

Performance: 37.68 GB/s (9.42 MElem/s).

Implementation 5

For c.c. 3.0 or greater, we can use warp shuffle:

```
if (tid < 32){
    mySum += sdata[tid + 32];
    for (int offset = warpSize/2; offset > 0; offset /= 2)
        mySum += __shfl_down_sync(mySum, offset);
}
```

This is safe for all GPUs.

Implementation 6

Can we unroll the for loop?

If we know the number of iterations, we can unroll it

- the number of iterations depends on the block size

Can we implement it generically?

- algorithm uses blocks of size 2^n
- the block size is upper-bound
- if we know the block size during compilation, we can use a template

```
template <unsigned int blockSize>
__global__ void reduce6(int *v)
```

Implementation 6

Conditions using `blockSize` are evaluated during compilation:

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

Performance: 50.64 GB/s (12.66 MElem/s).

Implementation 7

Can we implement faster algorithm?

Let's reconsider the complexity:

- $\log n$ parallel steps
- $n - 1$ additions
- time complexity for p threads running in parallel (using p processors): $\mathcal{O}\left(\frac{n}{p} + \log n\right)$

Cost of parallel computation

- defined as number of processors multiplied by time complexity
- if we assign one thread to one data element, we get $p = n$
- and the cost is $\mathcal{O}(n \cdot \log n)$
- which is not efficient

Implementation 7

Decreasing the cost

- we use $\mathcal{O}\left(\frac{n}{\log n}\right)$ threads
- each thread performs $\mathcal{O}(\log n)$ sequential steps
- after that, it performs $\mathcal{O}(\log n)$ parallel steps
- time complexity is the same
- the cost is $\mathcal{O}(n)$

What it means in practice?

- we reduce overhead of the computation (e.g., integer arithmetics)
- advantage if we have much more threads that is needed to saturate GPU

Implementation 7

We modify loading into shared memory

```

unsigned int gridSize = blockSize*2*gridDim.x;
sv[tid] = 0;

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

```

Performance: 77.21 GB/s (19.3 MElem/s).

Implementation 7

We modify loading into shared memory

```

unsigned int gridSize = blockSize*2*gridDim.x;
sv[tid] = 0;

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

```

Performance: 77.21 GB/s (19.3 MElem/s).

You can find those implementations in CUDA SDK.

Problem Choice

Before we start with code acceleration, we should consider carefully, if it is meaningful.

The accelerated code should be

- critical for application performance (profile... and profile on real data)
- large enough (usually not ideal for relatively simple but latency critical application)
- parallelizable (problematic, e.g., in simulation of a small system evolving for a long time)
- sufficient number of flops to memory transfers (consider slow PCI-E)

Parallelization

Vector addition

- simple to formulate data-parallel code
- no synchronization

Reduction

- may seem sequential at a glance
- but can be computed in $\log n$ steps
- needs some thinking about how to parallelize

Code divergence

Code divergence

- hurts performance when divergence occurs within the warp
- it can be easy to remove
 - reduction
- but also hard to remove
 - graph processing, many independent automata
 - sometimes different algorithm needs to be formulated

