Code Optimizations

Jiří Filipovič

Fall 2020

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

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 $n /= 2$ untill $n > 1$

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: $n + \frac{n}{m} + \frac{n}{m^2} + .. + \frac{n}{m^{\log_m n}} = (n-1)\frac{m}{m-1}$
- 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];
   s[tid] = mySum = mySum + s[tid + 16];
   s[tid] = mySum = mySum + s[tid + 8];
   s[tid] = mySum = mySum + s[tid + 4];
   s[tid] = mySum = mySum + s[tid + 2];
   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): $\mathcal{O}(\frac{n}{p} + \log n)$

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 $\mathcal{O}(\frac{n}{\log n})$ 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.

Intra-kernel Optimizations

The compiler optimizes each kernel separately, so it may miss some optimization opportunities.

- kernel fusion – gluing code from several kernels into one kernel
- kernel fission – splitting a kernel into several smaller kernels

Kernel fusion

Performance impact of kernel fusion

- reduce kernel execution overhead
- may add more parallelism
- allow more scalar code optimizations: common subexpression elimination, loop fusion, condition fusion
- reduce global memory transfers if kernels are flow-dependent or input-dependent

Correctness

- no flow dependency between thread blocks
- shared memory and registers locality has to be maintained

Kernel fission

Kernel fission reduces resources consumption

- increases occupancy
- may allow to use different algorithm (e.g. if part of the algorithm uses different amount of parallelism or different amount of resources)
- more complicated and divergent codes may be separated (e.g. handling array boundaries)

Correctness

- much easier, we just need to transfer data between new kernels

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)

Problem Choice

Do we optimize running time or power consumption?

- accelerators are usually faster, but also have higher power consumption
- how to deal with hybrid systems (e.g. CPU, GPU and Xeon Phi)
- influences decision what to buy as well as what to use (which resources let in power-saving mode)

Algorithm Design

Parallelization

- we need to parallelize computational problem
- we should be aware about target architecture even in this stage (consider e.g. graph algorithms)

It is difficult to accelerate codes on GPU:

- if threads within the warp access rather random addresses in the memory
- if threads within the warp diverges (by nature of the algorithm)
- if the parallelism is insufficient in certain parts of computation

How to Write Bug-Free Code

Test if API and kernel calls are successful

- otherwise, errors can appear later...

The memory allocation on GPU occurs seldom

- if your modify your code in a way your kernel does not write any result, you got a result from its previous run
- clear output arrays for debugging purposes

Be aware of out-of-bounds shared memory access

- kernel usually runs successfully, but one block interferes with another

Optimization

Start with the most important optimizations and continue with less important (so the effect of less important optimizations is not hidden). In general, this order should work well:

- PCI-E transfers reduction/overlay
- global memory access (bandwidth, latency)
- access to other types of memory
- divergence
- parallelism configuration (block size, amount of serial work per thread)
- instruction optimization

It is good idea to write your code configurable

- block size, number of serial iterations per thread, loop unrolling factor, used algorithm ...
- use macros or templates to ease configutation of the optimizations

Interpretation of Algorithm Performance

Some optimizations may be hidden

- e.g. optimizing instruction cannot help when code is bound by wrong global memory access
- can be reduced by applying more important optimizations earlier
- use the profiler

The optimization space is not continuous

- due to restricted amount of GPU resources
- e.g. improving efficiency of scalar code by using one more register may decrease the performance by restricting GPU occupancy

Performance is data-dependent

- data size: partition camping, underutilized GPU
- data content: sparse data with varying structure

What is real speedup over CPU?

Comparison of a theoretical peak is basic metric

- however, the speedup can be lower
  - insufficient parallelism
  - inappropriate data structures, random access
  - PCI-E bottleneck (especially multi-GPU algorithms)
- however, the speedup can be also higher
  - frequent usage of SFUs
  - complicated vectorization on CPU
  - insufficient scaling on SMP (cache interferences, NUMA)
- different scaling of CPU and GPU with growing problem size

Searching for Bottlenecks

The amount of arithmetic operations and memory transfers tells us what is expected to be a limit for algorithm

- sometimes bottleneck is not clear (overhead instructions, irregular memory access)
- code profiling – suitable to identify issues with instructions throughput or bad memory access pattern, more difficult to identify source of latency problems
- code modifications – more precise, but more difficult and not usable in all cases

Profiling

How close is the code to the hardware limits?

- profiler shows the overall utilization of particular GPU subsystems, such as cache, global memory, FP instructions etc.

Issues identification

- profiler detects some issues, such as shared memory bank conflicts or code divergence

We can inspect a code in details

- time spent on particular instructions/C for CUDA lines of code
- we need to compile the code with flag `-lineinfo`

Code Modifications

Global memory performance

- we comment-out the computation
- but we need to somehow use loaded data (to disallow compiler to exclude loading)
- we can check with profiler that the same amount of data are transfered

Instructions performance

- we comment-out data movement
- but the resulting data is needed to be stored (to disallow compiler to exclude computation)
  - but we do not want to store data...
  - we can move the code storing data into condition which is evaluated as false during computation (but not during compilation)
- be aware of execution overhead in the case of fast kernels

Code Modifications

Be aware of occupancy changes

- code modifications can release some resources
- we can restrict occupancy by allocating some dummy shared memory array

Interpretation of measured times

- original kernel execution time is close to sum of computation and memory kernel time – the latency is an issue
- computation or memory kernel time dominates and is closed to original kernel time – the performance is bounded by computation or memory
- computation and memory kernel times are similar to original kernel time – we need to optimize both

Code Modifications

Approximation of an optimization effect

- when we already know some performance issue
- when we want to know the effect of optimization before we actually implement it
- we can modify the code without preserving original functionality, but preserving amount of work and removing performance issue
  - cannot be done in all cases
  - may show us if we really address the performance issue
  - see matrix transposition example