CSE 746 - Parallel and High Performance Computing Lecture 4 - Reduction with CUDA

Pawel Pomorski, *HPC Software Analyst*
SHARCNET, University of Waterloo
ppomorsk@sharcnet.ca
http://ppomorsk.sharcnet.ca/

Reductions in CUDA

- Reductions: min/max, average, sum, ...
- Can be a significant bottleneck for the performance, because it breaks pure data parallelism.
- There is no perfect way to do reductions in CUDA. The two commonly used approaches (each with its own set of constraints) are:

  Binary reductions

  Atomic reductions

Binary reductions

- The most universal type of reductions (e.g., the only way to do double precision reductions)
- Even when using single precision (which is faster than double precision), binary summation will be more accurate than atomic summation, because it employs more accurate pairwise summation.
- Usually the more efficient way to do reductions

Binary reductions

- But: typically relies on (very limited) shared memory – placing constraints on how many reductions per kernel one can do
- Relies on thread synchronization, which can only be done within a single block – places constraints on how many threads can participate in a binary reduction (usually 64 ... 256; maximum 1024)
- For a large number of data elements (>1024), this leads to the need to do multi-level (multi-kernel) binary reductions, with storing the intermediate data in device memory; this can reduce the performance
- Can be less efficient for small number of data elements (<64)
- Significantly complicates the code

Atomic reductions

- Very simple and elegant code
  - Almost no change compared to the serial code
  - A single line code: much better for code development and maintenance
  - No need for multiple intermediate kernels (saves on overheads related to multiple kernel launches)
  - Requires no code changes when dealing with any number of data elements – from 2 to millions
- Usually more efficient when the number of data elements is small (<64)

Atomic reductions

- But: atomic operations are serialized, which usually means worse performance
- Only single precision accuracy - can become really bad for summation and averaging when the number of elements is large (many thousands) – because it uses sequential summation.
- All the above means that to find the right way to carry out a reduction in CUDA, with the right balance between code readability, efficiency, and accuracy, one often has to try both binary and atomic ways, and choose the best.

Binary summation with number of elements being a power of 2

```
__shared__ double sum[BLOCK_SIZE];
...
__syncthreads(); // Required if there were prior computations in the kernel
int nTotalThreads = blockDim.x; // Total number of active threads;
// only the first half of the threads will be active.

while(nTotalThreads > 1)
{
  int halfPoint = (nTotalThreads >> 1); // divide by two

  if (threadIdx.x < halfPoint)
  {
    int thread2 = threadIdx.x + halfPoint;
    sum[threadIdx.x] += sum[thread2]; // Pairwise summation
  }
__syncthreads();
nTotalThreads = halfPoint; // Reducing the binary tree size by two
}
```

Binary averaging with number of elements being a power of 2

```
__shared__ double avg[BLOCK_SIZE];
 ...
__syncthreads(); // Required if there were prior computations in the kernel
int nTotalThreads = blockDim.x;

while(nTotalThreads > 1)
{
   int halfPoint = (nTotalThreads >> 1); // divide by two
   if (threadIdx.x < halfPoint)
   {
     int thread2 = threadIdx.x + halfPoint;
     avg[threadIdx.x] += avg[thread2]; //   First sum
     avg[threadIdx.x] /= 2; // and then divide
   }
__syncthreads();
nTotalThreads = halfPoint; // Reducing the binary tree size by two
}
```

Binary min/max with number of elements being a power of 2

```
__shared__ double min[BLOCK_SIZE];
...
__syncthreads(); // Required if there were prior computations in the kernel
int nTotalThreads = blockDim.x;

while(nTotalThreads > 1)
{
  int halfPoint = (nTotalThreads >> 1); // divide by two
  if (threadIdx.x < halfPoint)
  {
    int thread2 = threadIdx.x + halfPoint;
    double temp = min[thread2];
    if (temp < min[threadIdx.x])
       min[threadIdx.x] = temp;
  }
  __syncthreads();
  nTotalThreads = halfPoint; // Reducing the binary tree size by two
}
```

Multiple binary reductions

```
__shared__ double min[BLOCK_SIZE], sum[BLOCK_SIZE];
...
__syncthreads(); // Required if there were prior computations in the kernel
int nTotalThreads = blockDim.x;

while(nTotalThreads > 1)
{
  int halfPoint = (nTotalThreads >> 1); // divide by two

  if (threadIdx.x <  halfPoint)
  {
    int thread2 = threadIdx.x + halfPoint;
    sum[threadIdx.x] += sum[thread2]; // First reduction

    double temp = min[thread2];
    if (temp < min[threadIdx.x])
       min[threadIdx.x] = temp; // Second reduction
  }
__syncthreads();
nTotalThreads = halfPoint; // Reducing the binary tree size by two
}
```

Two-step binary reduction

```
// Host code
#define BSIZE 1024 // Always use a power of two; can be 32...1024
// Total number of elements to process: 1024 < Ntotal < 1024^2

int Nblocks = (Ntotal+BSIZE-1) / BSIZE;

// First step (the results should be stored in global device memory):
x_prereduce <<<Nblocks, BSIZE >>> ();

// Second step (will read the input from global device memory):
x_reduce <<<1, Nblocks >>> ();
```

Binary reduction with an arbitrary number of elements (BLOCK_SIZE)

```
__shared__ double sum[BLOCK_SIZE];
...
__syncthreads(); // Required if there were prior computations in the kernel
int nTotalThreads = blockDim_2; // Total number of threads, rounded up to the next
                                //power of two

while(nTotalThreads > 1)
{
  int halfPoint = (nTotalThreads >> 1); // divide by two

  if (threadIdx.x < halfPoint)
    {
    int thread2 = threadIdx.x + halfPoint;
    if (thread2 < blockDim.x) // Skipping the fictitious threads
                              // blockDim.x ...blockDim_2-1
       sum[threadIdx.x] += sum[thread2]; // Pairwise summation
    }
__syncthreads();
nTotalThreads = halfPoint; // Reducing the binary tree size by two
}
```

Binary reduction with an arbitrary number of elements (BLOCK_SIZE)

- You will have to compute blockDim_2 (blockDim.x rounded up to the next power of two), either on device or on host (and then copy it to device). One could use the following function to compute blockDim_2, valid for 32-bit integers:

```
int NearestPowerOf2 (int n)
{
  if (!n) return n; // (0 == 2^0)

  int x = 1;
  while(x < n)
    {
    x <<= 1;
    }
  return x;
}
```

Atomic reductions

```
// In global device memory:
__device__ float xsum; __device__ int isum, imax;

// In a kernel:
float x;
int i;
__shared__ imin;
...
atomicAdd (&xsum, x);
atomicAdd (&isum, i);
atomicMax (&imax, i);
atomicMin (&imin, i);

//see CUDA specification for full list of atomic operations available
```

Writing your own atomic operations in CUDA

- Possible, but this is a more advanced topic
- Any atomic operation can be implemented based on **atomicCAS()** (Compare and Swap)
- Performance may be very poor
- Here is what a double precision atomicAdd would look like

```
__device__ double atomicAdd(double* address, double val)
{
    unsigned long long int* address_as_ull = (unsigned long long int*)address;
    unsigned long long int old = *address_as_ull, assumed;
    do {
        assumed = old;
        old = atomicCAS(address_as_ull, assumed,
                        __double_as_longlong(val +__longlong_as_double(assumed)));
    } while (assumed != old);
    return __longlong_as_double(old);
}
```

atomicCAS

- reads the 32-bit or 64-bit word **old** located at the address **address** in global or shared memory
- computes (old == compare ? val : old), stores the result back to memory at the same address
- returns **old**

```
int atomicCAS(int* address, int compare, int val);
unsigned int atomicCAS(unsigned int* address,
                       unsigned int compare,
                       unsigned int val);
unsigned long long int atomicCAS(unsigned long long int* address,
                                 unsigned long long int compare,
                                 unsigned long long int val);
```

Exercise - implement reduction via 3 approaches

Can be found in:

/home/ppomorsk/CSE746_lec4/Reduction

Exercise - implement largest prime search

Can be found in:

/home/ppomorsk/CSE746_lec4/Primes