CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Writing GPU Kernels

Sebastian Keller, Javier Otero, Prashanth Kanduri
and Ben Cumming, CSCS
July 19, 2021

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Going Parallel : Kernels and Threads

Threads and Kernels

- **Threads** are streams of execution, run simultaneously on GPU.
- A **kernel** is the function run by each thread.
- CUDA provides language support for:
  - writing kernels;
  - launching many threads to execute a kernel in parallel.
- CUDA hides the low-level details of launching threads.

Threads and Kernels

- **Threads** are streams of execution, run simultaneously on GPU.
- A **kernel** is the function run by each thread.
- CUDA provides language support for:
  - writing kernels;
  - launching many threads to execute a kernel in parallel.
- CUDA hides the low-level details of launching threads.

The process for developing CUDA kernels

1. Formulate algorithm in terms of parallel work items.
2. Write a kernel implementing a work item on one thread.
3. Launch the kernel with the required number of threads.

Scaled Vector Addition (axpy)

We have used CUBLAS to perform scaled vector addition:

$$\mathbf{y} = \mathbf{y} + \alpha\mathbf{x}$$

- $\mathbf{x}$ and $\mathbf{y}$ are vectors of length n; $x, y \in \mathbb{R}^n$
- α is scalar. $\alpha \in \mathbb{R}$

Applying `axpy` requires n operations:

$$y_i \leftarrow y_i + a * x_i, \quad i = 0, 1, \ldots, n-1$$

which can be performed **independently** and **in any order**.

Scaled Vector Addition (axpy)

We have used CUBLAS to perform scaled vector addition:

$$\mathbf{y} = \mathbf{y} + \alpha\mathbf{x}$$

- $\mathbf{x}$ and $\mathbf{y}$ are vectors of length n; $\quad x, y \in \mathbb{R}^n$
- α is scalar. $\quad \alpha \in \mathbb{R}$

Applying `axpy` requires n operations:

$$y_i \leftarrow y_i + a * x_i, \quad i = 0, 1, \ldots, n-1$$

which can be performed **independently** and **in any order**.

`axpy` implemented on CPU with a loop

```
void axpy(double* y, const double* x, double a, int n) {
    for(int i=0; i<n; ++i)
        y[i] = y[i] + a*x[i];
}
```

Kernels

A **kernel** defines the work item for a single thread

- The work is performed by many threads executing the same kernel **simultaneously**.
- Conceptually corresponds to the inner part of a loop for BLAS1 operations like `axpy`.

Kernels

A **kernel** defines the work item for a single thread

- The work is performed by many threads executing the same kernel **simultaneously**.
- Conceptually corresponds to the inner part of a loop for BLAS1 operations like `axpy`.

host : add two vectors

```
void add_cpu(int* a, int* b, int n){
  for(auto i=0; i<n; ++i)
    a[i] = a[i] + b[i];
}
```

CUDA : add two vectors

```
__global__
void add_gpu(int* a, int* b, int n){
  auto i = threadIdx.x;
  a[i] = a[i] + b[i];
}
```

- `__global__` keyword indicates a kernel
- `threadIdx` used to find unique id of each thread

Launching a kernel

- Host code launches a kernel on the GPU **asynchronously**.
- CUDA provides the "triple chevron" `<<<_,_>>>` syntax for launching a kernel.

host : add two vectors

```
auto n = 1024;
auto a = host_malloc<int>(n);
auto b = host_malloc<int>(n);
add_cpu(a, b, n);
```

CUDA : add two vectors

```
auto n = 1024;
auto a = device_malloc<int>(n);
auto b = device_malloc<int>(n);
add_gpu<<<1,n>>>(a, b, n);
```

- `add_gpu<<<1, num_threads>>>(args... )` launches the kernel `add_gpu` with `num_threads` parallel threads.

Exercise: My First Kernel

Open `axpy/axpy.cu`

1. Write a kernel that implements **axpy** for `double`
   - `axpy_kernel(double* y, double* x, double a, int n)`
   - **extra**: can you write a C++ templated version for any type?
2. launch the kernel (look for `TODO`)
3. Compile the test and run
   - it will pass with no errors on success
   - first try with small vectors of size 8
   - try increasing launch size... what happens?
4. **extra**: can you extend the kernel to work for larger arrays?

Scaling Up : Thread Blocks

In the `axpy` exercises we were limited to 1024 threads for a kernel launch

- but we need to scale beyond 1024 threads for the **massive parallelism** we were promised!

Thread blocks and grids

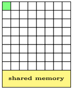
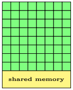
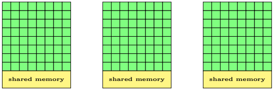
kernels are executed in groups of threads called **thread blocks**

- the launch configuration `axpy<<<grid_dim, block_dim>>>(...)`
  - launch a **grid** of `grid_dim` **blocks**
  - each **block** has `block_dim` **threads**
  - for a total of `grid_dim` × `block_dim` threads
- previously we launched just one thread block `axpy<<<1, n>>>(...)`

Why the additional complexity?

Coordination between threads doesn't scale:

- Threads in a block can synchronize and share resources
- This does not scale past a certain number of cores/threads
- EACH P100 GPU **streaming multiprocessor** (SMX) has 64 CUDA cores, and can run 2048 threads
- Threads in a block run on the same SMX, with shared resources and thread cooperation
- Work is broken into blocks, which are distributed over the 56 SMXs on the GPU.

concept	hardware	
thread		■ each thread executed on one core
block		■ block executed on 1 SMX ■ multiple blocks per SMX if sufficient resources ■ threads in a block share SMX resources
grid		■ kernel is executed in grid of blocks ■ blocks distributed over SMXs ■ multiple kernels can run at same time

Calculating thread indexes

A kernel has to calculate the index of its work item

- In `axpy` we used `threadIdx.x` for the index.
- With multiple blocks, we need more information, which is available in the following **magic variables**:

`gridDim`	: total number of blocks in the grid
`blockDim`	: number of threads in a thread block
`blockIdx`	: index of block `[0, gridDim-1]`
`threadIdx`	: index of thread in thread block `[0, blockDim-1]`

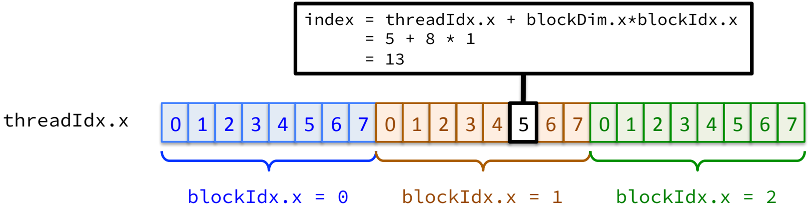
Calculating thread indexes

Consider accessing an array of length 24 with 8 threads per block. The **dimensions** of the kernel launch are:

- `blockDim.x == 8` (8 threads/block)
- `gridDim.x == 3` (3 blocks)

We calculate the index for our thread using the formula

```
auto index = threadIdx.x + blockIdx.x*blockDim.x
```



Calculating grid dimensions

The number of thread blocks and the number of threads per block are parameters for the kernel launch:

```
kernel<<<blocks, threads_per_block>>>(...)
```

Remember to guard against overflow when the number of work items is not divisible by the thread block size

vector addition with multiple blocks

```
__global__
void add_gpu(int* a, int* b, int n){
  auto i = threadIdx.x + blockIdx.x*blockDim.x;
  if(i<n) { // guard against access off end of arrays
    a[i] += b[i];
  }
}

// in main()
auto block_size = 512;
auto num_blocks = (n + (block_size-1)) / block_size;
add_gpu<<<num_blocks, block_size>>>(a, b, n);
```

Calculating grid dimensions

We have to take care when calculating the number of blocks in the grid, i.e. `blocks`:

```
kernel<<<blocks, threads_per_block>>>(...)
```

Most likely, the number of work items `n` is not a multiple of `threads_per_block`

- some threads in the last thread block will be idle.

Calculating grid dimensions

```
// in main()
auto block_size = 512;
auto num_blocks = (n + block_size-1) / block_size;
add_gpu<<<num_blocks, block_size>>>(a, b, n);
```

How many threads per block?

The number of threads per block has an impact on performance

- The optimal number depends on resources required by the kernel (registers, shared memory, computational intensity, etc).

The short answer is 64 or 128 on P100.

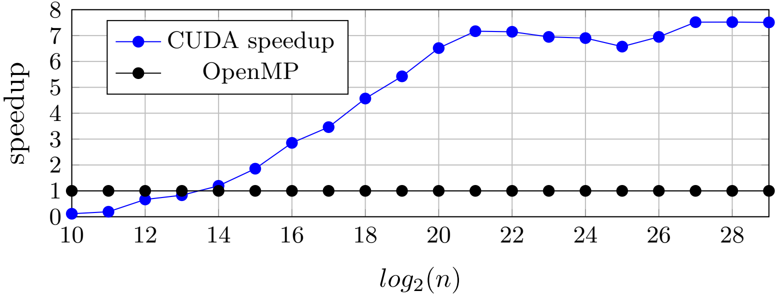
- For the main kernels in your application, perform experiments to find the ideal block size.

Exercise: Blocks

Open `axpy/axpy.cu` from the last exercise

1. Extend the **axpy** kernel for arbitrarily large input arrays (any `n`)
2. Update the call site to calculate the grid configuration
3. Compile the test and run
   - it will pass with no errors on success
4. Experiment with varying the size of the arrays (scaling)
   - start small and increase
5. finish the `newton.cu` example
   - how do the h2d, d2h and kernel timings compare?
6. **extra**: Compare scaling with the `axpy_omp` benchmark
7. **extra**: Experiment with varying the block size

Exercise: Results



The GPU is a throughput device:

- CUDA breaks even for $n \geq 2^{14} \approx 16,000$
- requires $2^{21} \approx 2,000,000$ to gain "full" $7\times$ speedup

You have to provide enough parallelism to exploit many cores