CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

CUDA: 2D Kernels, MPI and Concurrency

Ben Cumming, CSCS
July 24, 2016

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Going 2D and 3D

Launch Configuration

- so far we have used one-dimensional launch configurations
  - threads in blocks indexed using `threadIdx.x`
  - blocks in a grid indexed using `blockIdx.x`
- many kernels map naturally onto 2D and 3D indexing
  - e.g. matrix-matrix operations
  - e.g. stencils

Full Launch Configuration

kernel launch dimensions can be specified with `dim3` structs

```
kernel<<<dim3 gridDim, dim3 blockDim>>>(...);
```

- `dim3.x`, `dim3.y` and `dim3.z` specify the launch dimensions
- can be constructed with 1, 2 or 3 dimensions
- unspecified `dim3` dimensions are set to 1

launch configuration examples

```
// 1D: 128x1x1 for 128 threads
dim3 a(128);
// 2D: 16x8x1  for 128 threads
dim3 b(16, 8);
// 3D: 16x8x4  for 512 threads
dim3 c(16, 8, 4);
```

The `threadIdx`, `blockDim`, `blockIdx` and `gridDim` can be treated like 3D vectors via the `.x`, `.y` and `.z` members.

matrix addition example

```
__global__
void MatAdd(float *A, float *B, float *C, int n) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if(i<n && j<n) {
        auto pos = i + j*n;
        C[pos] = A[pos] + B[pos];
    }
}
int main() {
    // ...
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(n / threadsPerBlock.x, n / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    // ...
}
```

Exercise: Launch Configurations

- 2D stencil in `diffusion/diffusion2d.cu`
  - a plotting script is provided for visualizing the results
  - use a small domain for visualization
- **extra**: can you improve performance with shared memory?

running the code

```
module load python/2.7.6
cd cuda/practicals/diffusion

make
srun diffusion2d.cuda 8 1000000
python plotting.py
```

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

MPI with GPUs

MPI with data in device memory

Our GPU-accelerated applications use GPUs to parallelize on-node computation

- and MPI for communication between nodes

It is likely that communication between MPI ranks will involve information on the GPU

1. allocate buffers in host memory
2. manually copy from device→host memory
3. perform MPI communication with host buffers
4. copy received data from host→device memory

This approach can be very fast:

- have a CPU thread dedicated to asynchronous host↔device and MPI communication

GPU-Aware MPI

GPU-aware MPI implementations can automatically handle MPI transactions with pointers to GPU memory

- MVAPICH 2.0
- OpenMPI since version 1.7.0
- Cray MPI

How it works

- each pointer passed to MPI is checked to see if it is in host or device memory if not set, MPI assumes that all pointers are to host memory, and your application will probably crash with segmentation faults
- small messages between GPUs (up to ≈8 k) are copied directly with **RDMA**
- larger messages are **pipelined** via host memory

How to use G2G communication

- set the environment variable

```
export MPICH_RDMA_ENABLED_CUDA=1
```

  - if not set, MPI assumes that all pointers are to host memory, and your application will probably crash with segmentation faults
- experiment with the environment variable

```
MPICH_G2G_PIPELINE
```

  - sets the maximum number of 512 kB message chunks that can be in flight (default 16)

MPI with G2G example

```
MPI_Request srequest, rrequest;
auto send_data = malloc_device<double>(100);
auto recv_data = malloc_device<double>(100);

// call MPI with GPU pointers
MPI_Irecv(recv_data, 100, MPI_DOUBLE, source, tag, MPI_COMM_WORLD,
     &rrequest);
MPI_Isend(send_data, 100, MPI_DOUBLE, target, tag, MPI_COMM_WORLD,
     &srequest);
```

Capabilities and Limitations

- support for most MPI API calls (point-to-point, collectives, etc)
- robust support for common MPI API calls
  - i.e. point-to-point operations
- no support for user-defined MPI data types

Exercise: MPI with G2G

- 2D stencil with MPI in `diffusion/diffusion2d_mpi.cu`
- copy the kernel and kernel lanuch from previous example
  1. implement MPI communication with host buffering
  2. implement MPI communication with G2G
  3. can you observe any performance differences between the two?
  4. why are we restricted to just 1 MPI rank per node?

running the MPI example

```
cd cuda/practicals/diffusion
make
srun -n2 -N2 diffusion2d_mpi.cuda 8
module load python/2.7.6
python plotting.py
# once it gets the correct results:
sbatch job.batch
```

Exercises: 2D Diffusion with MPI Results

Time for 1000 time steps $128 \times 16{,}382$

nodes	G2G off	G2G on
1	0.479	0.479
2	0.277	0.274
4	0.183	0.180
8	0.152	0.151
16	0.167	0.117

Concurrency

Concurrency

Concurrency is the ability to perform multiple CUDA operations simultaneously

- CUDA kernels
- copying from host to device
- copying from device to host
- operations on the host CPU

Concurrency enables

- both CPU and GPU can work at the same time
- multiple tasks can be run on GPU simultaneously
- overlapping of communication and computation

Host code

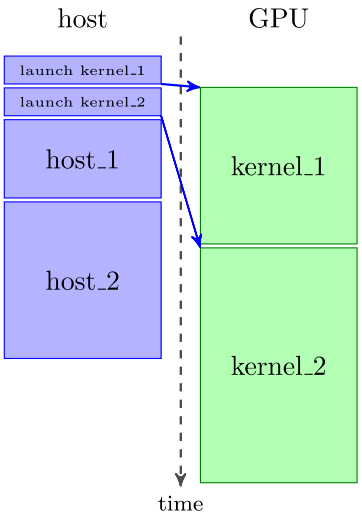
```
kernel_1<<<...>>>(...);
kernel_2<<<...>>>(...);
host_1(...);
host_2(...);
```

The host:

- launches the two CUDA kernels
- then executes host calls sequentially

The GPU:

- executes asynchronously to host
- executes kernels sequentially



The CUDA language and runtime libraries provide mechanisms for coordinating asynchronous GPU execution

- **CUDA streams** can concurrently run independent kernels and memory transfers
- **CUDA events** can be used to synchronize streams and query the status of kernels and transfers

Streams

A CUDA stream is a sequence of operations that execute in **issue order** on the GPU

- CUDA operations are kernels and copies between host and device memory spaces

Streams and concurrency

- operations in different streams **may** run concurrently
  - there have to be sufficient resources on the GPU (registers, shared memory, SMXs, etc)
- operations in the same stream **are** executed sequentially
- if no stream is specified, all kernels are launched in the default stream

Managing streams

A stream is represented using a `cudaStream_t` type

- `cudaStreamCreate(cudaStream_t* s)` and `cudaStreamDestroy(cudaStream_t s)` can be used to create and free CUDA streams respectively
- To launch a kernel on a stream specify the stream id as a fourth parameter to the launch syntax

```
kernel<<<grid_dim, block_dim, shared_size, stream>>>(...)
```

- the default CUDA stream is the `NULL` stream, or stream 0 (`cudaStream_t` is an integer)

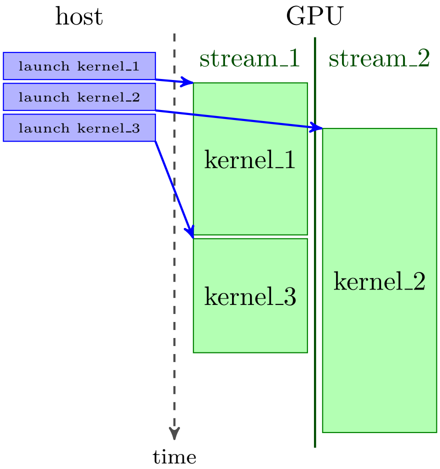
Basic cuda stream usage

```
// create stream
cudaStream_t stream;
cudaStreamCreate(&stream);
// launch kernel in stream
my_kernel<<<grid_dim, block_dim, shared_size, stream>>>(..)
// release stream when finished
cudaStreamDestroy(stream);
```

Host code

```
kernel_1<<<,,,stream_1>>>();
kernel_2<<<,,,stream_2>>>();
kernel_3<<<,,,stream_1>>>();
```

- `kernel_1` and `kernel_3` are serialized in `stream_1`
- `kernel_2` can run asynchronously in `stream_2`
- note that `kernel_2` will only run concurrently if there are sufficient resources available on the GPU, i.e. if `kernel_1` is not using all of the SMXs.



Asynchronous copy

```
cudaMemcpyAsync(*dst, *src, size, kind, cudaStream_t stream = 0);
```

- takes an additional parameter stream, which is 0 by default
- returns immediately after initiating copy
  - host can do work while copy is performed
  - only if **pinned memory** is used
- copies in the same direction (i.e. H2D or D2H) are serialized
  - copies in opposite directions are concurrent if in different streams

What is pinned memory?

Pinned memory (or page-locked) memory will not be paged out to disk when memory runs low

- the GPU can safely remotely read/write the memory directly without host involvement
- only use for transfers, because it easy to run out of memory

Managing pinned memory

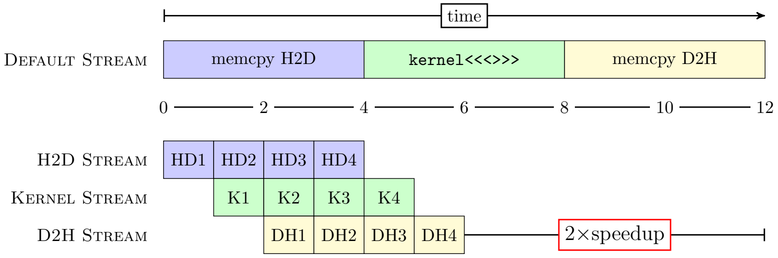
`cudaMallocHost(**ptr, size);` and `cudaFreeHost(*ptr);`

- allocate and free pinned memory (`size` is in bytes).

Asynchronous copy example: streaming workloads

Computations that can be performed independently, e.g. our `axpy` example:

- data in host memory has to be copied to the device, and the result copied back after the kernel is computed.
- we can overlap the copies with the kernel calls by breaking the data into chunks.



CUDA events

To implement the streaming workload we have to coordinate operations on the GPU. CUDA events can be used for this purpose.

- synchronize tasks in different streams, e.g.:
  - don't start kernel in kernel stream until data copy stream has finished.
  - wait until required data has finished copy from host before launching kernel
- query status of concurrent tasks
  - has kernel finished/started yet?
  - how long did a kernel take to compute?

Managing events

`cudaEventCreate(cudaEvent_t*);` and `cudaEventDestroy(cudaEvent_t);`

- create and free `cudaEvent_t`

`cudaEventRecord(cudaEvent_t, cudaStream_t);`

- enqueue an event in a stream

`cudaEventSynchronize(cudaEvent_t);`

- make host execution wait for event to occur.

`cudaEventQuery(cudaEvent_t)`

- test if the work before an event in a queue has been completed

`cudaEventElapsedTime(float*, cudaEvent_t, cudaEvent_t);`

- get time between two events

Using events to time kernel execution

```
cudaEvent_t start, end;
cudaStream_t stream;
float time_taken;

// initialize the events and streams
cudaEventCreate(&start);
cudaEventCreate(&end);
cudaStreamCreate(&stream);

cudaEventRecord(start, stream); // enqueue start in stream
my_kernel<<<grid_dim, block_dim, 0, stream>>>();
cudaEventRecord(end, stream);   // enqueue end in stream
cudaEventSynchronize(end);      // wait for end to be reached
cudaEventElapsedTime(&time_taken, start, end);

std::cout << "kernel took " << 1000*time_taken << " s\n";

// free resources for events and streams
cudaEventDestroy(start);
cudaEventDestroy(end);
cudaStreamDestroy(stream);
```

Copy→kernel synchronization

```
cudaEvent_t event;
cudaStream_t kernel_stream, h2d_stream;
size_t size = 100*sizeof(double);
double *dptr, *hptr;

// initialize
cudaEventCreate(&event);
cudaStreamCreate(&kernel_stream);
cudaStreamCreate(&h2d_stream);

cudaMalloc(&dptr, size);
cudaMallocHost(&hptr, size); // use pinned memory!

// start asynchronous copy in h2d_stream
cudaMemcpyAsync(dptr, hptr, size,
                cudaMemcpyHostToDevice, h2d_stream);
// enqueue event in stream
cudaEventRecord(event, h2d_stream);
// make kernel_stream wait for copy to finish
cudaStreamWaitEvent(kernel_stream, event, 0);
// enqueue my_kernel to start when event has finished
my_kernel<<<grid_dim, block_dim, 0, kernel_stream>>>();

// free resources for events and streams
cudaEventDestroy(event);
cudaStreamDestroy(h2d_stream);
cudaStreamDestroy(kernel_stream);
cudaFree(dptr);
cudaFreeHost(hptr);
```

Exercises

1. Open `util.h` in `cuda/practicals/async` and understand `copy_to_{host/device}_async()` and `malloc_pinned_host()`
2. Open `CudaEvent.h` and `CudaStream.h`
   - what is the purpose of these classes?
   - what does `CudaStream::enqueue_event()` do?
3. Open `memcopy1.cu` and run
   - what does the benchmark test?
   - what is the effect of turning on `USE_PINNED`?
     Hint: try small and large values for `n` (8, 16, 20, 24)
4. Inspect `memcopy2.cu` and run
   - what effect does changing the number of chunks have?
5. Inspect `memcopy3.cu` and run
   - how does it differ from `memcopy2.cu`?
   - what effect does changing the number of chunks have?

Using events to time kernel execution : **with helpers**

```
CudaStream stream(true);

auto start = stream.enqueue_event();
my_kernel<<<grid_dim, block_dim, 0, stream.stream()>>>();
auto end = stream.enqueue_event();
end.wait();
auto time_taken = end.time_since(start);

std::cout << "kernel took " << 1000*time_taken << " s\n";
```

Copy→kernel synchronization : **with helpers**

```
CudaStream kernel_stream(true), h2d_stream(true);
auto size = 100;
auto dptr = device_malloc<double>(size);
auto hptr = pinned_malloc<double>(size);

copy_to_device_async<double>(hptr,dptr,size,h2d_stream.stream());
auto event = h2d_stream.enqueue_event();
kernel_stream.wait_on_event(event);
my_kernel<<<grid_dim, block_dim, 0, kernel_stream.stream()>>>();

cudaFree(dptr);
cudaFreeHost(hptr);
```

Profiling CUDA applications

To analyze concurrent applications we need tools that can visually represent application flow.
The CUDA toolkit provides the tools **nvprof** and **nvvp** for profiling our GPU applications

- there are visual tools for Windows and Eclipse too
- they work for OpenACC applications too

nvprof

nvprof is a command line tool

- can be used to generate text reports
- `nvprof --help` for a full list of options
- `nvprof app.exe` will perform basic profling of application and print text summary
- `nvprof -o profile.out app.exe` will save profile information to file `profile.out` for visualization with nvvp

Demonstration

Use nvprof on the memcopy test codes

nvvp

nvvp is a graphical tool for visualizing CUDA applications

- can also be used to perform interactive profiling and guided analysis
- this is not so easy on Cray systems
- we can also use the output from nvprof
  - use `nvprof -o profile.out ... ./app.out` to generate detailed analysis
  - this can take a long time, because each kernel has to be replayed multiple times to collect all of the information required for the report.

Demonstration

Use nvvp on the output of nvprof for the memcopy examples

Some rough guidelines for concurrency

Ideally for most workloads you don't want to rely on streams to fill the GPU with work

- a sign that the working set per GPU is not large enough
- full concurrency is difficult in practice
  - a low-level optimization strategy for the last few %
- this isn't a hard and fast rule

Streams come into their own for overlapping communication and computation

- possible to transfer data in both directions concurrently with kernel execution

Final Thoughts

Is this so hard?

Each topic so far has not been too complicated:

- moving memory between host and device
- writing kernels
- parallel kernel launching
- ... ok, concurrency is hard (but concurrency is hard on the CPU too)

However, there is a lot that you have to keep in mind:

- where is my data?
- loop free thinking: individual work items to be performed in parallel
- synchronization between threads, host and device?

Combined this makes GPU programming tough to start with.

- OpenMP and OpenACC attempt to make things "easy", but they still have shortcomings

What next?

You have seen some features of CUDA

- not an exhaustive overview!
- there are some "essential" features that weren't covered

And more importantly, I hope that you have started to think about parallel work items

- a different perspective than thinking in loops
- an important skill for more than just GPU programming