Lecture 6: odds and ends

Prof. Mike Giles

mike.giles@maths.ox.ac.uk

Oxford University Mathematical Institute

Oxford e-Research Centre

Overview

- synchronicity
- multiple streams and devices
- multiple GPUs
- other odds and ends

Warnings

- I haven't tried most of what I will describe
- some of these things have changed from one version of CUDA to the next – everything here is for the latest version
- overall, keep things simple unless it's really needed for performance
- if it is, proceed with extreme caution, do practical 11, and check out the examples in the SDK

Synchronicity

A computer system has lots of components:

- CPU(s)
- GPU(s)
- memory controllers
- network cards

Many of these can be doing different things at the same time – usually for different processes, but sometimes for the same process

Synchronicity

The von Neumann model of a computer program is synchronous with each computational step taking place one after another

- this is an idealisation – almost never true in practice
- compiler frequently generates code with overlapped instructions (pipelined CPUs) and does other optimisations which re-arrange execution order and avoid redundant computations
- however, it is usually true that as a programmer you can think of it as a synchronous execution when working out whether it gives the correct results
- when things become asynchronous, the programmer has to think very carefully about what is happening and in what order

Synchronicity

With GPUs we have to think even more carefully:

- host code executes on the CPU(s);
  kernel code executes on the GPU(s)
- . . . but when do the different bits take place?
- . . . can we get better performance by being clever?
- . . . might we get the wrong results?

Key thing is to try to get a clear idea of what is going on
– then you can work out the consequences

GPU code

- for each warp, code execution is effectively synchronous
- different warps execute in an arbitrary overlapped fashion – use `__syncthreads()` if necessary to ensure correct behaviour
- different thread blocks execute in an arbitrary overlapped fashion

All of this has been described over the past 3 days – nothing new here.

The focus of these new slides is on host code and the implications for CPU and GPU execution

Host code

Simple/default behaviour:

- 1 CPU
- 1 GPU
- 1 thread on CPU (i.e. scalar code)
- 1 default “stream” on GPU

Host code

- most CUDA calls are synchronous / blocking:
- example: `cudaMemcpy`
  - host call starts the copying and waits until it has finished before the next instruction in the host code
  - why? – ensures correct execution if subsequent host code reads from, or writes to, the data being copied

Host code

- CUDA kernel launch is asynchronous / non-blocking
  - host call starts the kernel execution, but doesn't wait for it to finish before going on to next instruction
- similar for `cudaMemcpyAsync`
  - starts the copy but doesn't wait for completion
  - has to be done through a "stream" with page-locked memory (also known as pinned memory) – see documentation
- in both cases, host eventually waits when at a `cudaDeviceSynchronize()` call
- benefit? – in general, doesn't affect correct execution, and might improve performance by overlapping CPU and GPU execution

Host code

What could go wrong?

- kernel timing – need to make sure it's finished
- could be a problem if the host uses data which is read/written directly by kernel, or transferred by `cudaMemcpyAsync`
- `cudaDeviceSynchronize()` can be used to ensure correctness (similar to `__syncthreads()` for kernel code)

Multiple Streams

Quoting from section 3.2.5.5 in the CUDA Programming Guide:

> Applications manage concurrency through streams.
>
> A stream is a sequence of commands (possibly issued by different host threads) that execute in order.
>
> Different streams, on the other hand, may execute their commands out of order with respect to one another or concurrently.

Multiple Streams

Optional stream argument for

- kernel launch
- `cudaMemcpyAsync`

with streams creating using `cudaStreamCreate`

Within each stream, CUDA operations are carried out in order (i.e. FIFO – first in, first out); one finishes before the next starts

Key to getting better performance is using multiple streams to overlap things

Page-locked memory

Section 3.2.4:

- host memory is usually paged, so run-time system keeps track of where each page is located
- for higher performance, can fix some pages, but means less memory available for everything else
- CUDA uses this for better host <–> GPU bandwidth, and also to hold “device” arrays in host memory
- can provide up to 100% improvement in bandwidth
- also, it is required for `cudaMemcpyAsync`
- allocated using `cudaHostAlloc`, or registered by `cudaHostRegister`

Default stream

The way the default stream behaves in relation to others depends on a compiler flag:

- no flag, or `--default-stream legacy`

  old (bad) behaviour in which a `cudaMemcpy` or kernel launch on the default stream blocks/synchronizes with other streams

- `--default-stream per-thread`

  new (good) behaviour in which the default stream doesn't affect the others

- note: flag label is a bit odd – it has other effects too

Practical 11

```
cudaStream_t streams[8];
float *data[8];

for (int i = 0; i < 8; i++) {
  cudaStreamCreate(&streams[i]);
  cudaMalloc(&data[i], N * sizeof(float));

  // launch one worker kernel per stream
  kernel<<<1, 64, 0, streams[i]>>>(data[i], N);

  // do a Memcpy and launch a dummy kernel on default stream
  cudaMemcpy(d_data,h_data,sizeof(float),
             cudaMemcpyHostToDevice);
  kernel<<<1, 1>>>(d_data, 0);
}
cudaDeviceSynchronize();
```

Default stream

The second (main?) effect of the flag comes when using multiple threads (e.g. OpenMP or POSIX multithreading)

In this case the effect of the flag is to create separate independent (i.e. non-interfering) default streams for each thread

Using multiple default streams, one per thread, is a good alternative to using multiple “proper” streams

Practical 11

```
omp_set_num_threads(8);
float *data[8];

for (int i = 0; i < 8; i++)
  cudaMalloc(&data[i], N * sizeof(float));

#pragma omp parallel for
for (int i = 0; i < 8; i++) {
  printf(" thread ID = %d \n",omp_get_thread_num());

  // launch one worker kernel per thread
  kernel<<<1, 64>>>(data[i], N);
}

cudaDeviceSynchronize();
```

Stream commands

Each stream executes a sequence of kernels, but sometimes you also need to do something on the host.

There are at least two ways of coordinating this:

- use a separate thread for each stream
  - it can wait for the completion of all pending tasks, then do what's needed on the host
- use just one thread for everything
  - for each stream, add a callback function to be executed (by a new thread) when the pending tasks are completed
  - it can do what's needed on the host, and then launch new kernels (with a possible new callback) if wanted

Stream commands

- `cudaStreamCreate()`
  creates a stream and returns an opaque “handle”
- `cudaStreamSynchronize()`
  waits until all preceding commands have completed
- `cudaStreamQuery()`
  checks whether all preceding commands have completed
- `cudaStreamAddCallback()`
  adds a callback function to be executed on the host once all preceding commands have completed

Stream events

Useful for synchronisation and timing between streams:

- `cudaEventCreate(event)`
  creates an “event”
- `cudaEventRecord(event,stream)`
  puts an event into a stream (by default, stream 0)
- `cudaEventSynchronize(event)`
  CPU waits until event occurs
- `cudaStreamWaitEvent(stream,event)`
  stream waits until event occurs
- `cudaEventQuery(event)`
  check whether event has occured
- `cudaEventElapsedTime(time,event1,event2)`

Multiple devices

What happens if there are multiple GPUs?

CUDA devices within the system are numbered, not always in order of decreasing performance

- by default a CUDA application uses the lowest number device which is “visible” and available
- visibility controlled by environment variable `CUDA_VISIBLE_DEVICES`
- current device can be set by using `cudaSetDevice`
- `cudaGetDeviceProperties` does what it says
- each stream is associated with a particular device – current device for a kernel launch or a memory copy
- see `simpleMultiGPU` example in SDK
- see section 3.2.6 for more information

Multiple devices

If a user is running on multiple GPUs, data can go directly between GPUs (peer – peer) – doesn't have to go via CPU

- very important when using new direct NVlink interconnect – much faster than PCIe
- `cudaMemcpy` can do direct copy from one GPU's memory to another
- a kernel on one GPU can also read directly from an array in another GPU's memory, or write to it
- this even includes the ability to do atomic operations with remote GPU memory
- for more information see Section 4.11, "Peer Device Memory Access" in CUDA Runtime API documentation: `https://docs.nvidia.com/cuda/cuda-runtime-api/`

Multi-GPU computing

Single workstation / server:

- a big enclosure for good cooling
- up to 4 high-end cards in 16x PCIe v3 slots – up to 12GB/s interconnect
- 2 high-end CPUs
- 1.5kW power consumption – not one for the office

NVIDIA DGX-1 Deep Learning server

- 8 NVIDIA GV100 GPUs, each with 32GB HBM2
- 2 $\times$ 20-core Intel Xeons (E5-2698 v4 2.2 GHz)
- 512 GB DDR4 memory, 8TB SSD
- 150GB/s NVlink interconnect between the GPUs

Multi-GPU computing

A bigger configuration:

- NVIDIA DGX-2 Deep Learning server
  - 16 NVIDIA GV100 GPUs, each with 32GB HBM2
  - 2 $\times$ 24-core Intel Xeons (Platinum 8168)
  - 1.5 TB DDR4 memory, 32TB SSD
  - NVSwitch interconnect between the GPUs
- a distributed-memory cluster / supercomputer with multiple nodes, each with
  - 2-4 GPUs
  - 100 Gb/s Infiniband
- PCIe v3 bandwidth of 12 GB/s similar to Infiniband bandwidth

Multi-GPU computing

The biggest GPU systems in Top500 list (June 2018):

- Summit (Oak Ridge National Lab, USA)
  - 122 petaflop (#1), 9MW
  - IBM Power 9 CPUs, NVIDIA Volta GV100 GPUs
- Sierra (Lawrence Livermore National Lab, USA)
  - 76 petaflop (#3)
  - IBM Power 9 CPUs, NVIDIA Volta GV100 GPUs
- ABCI (AIST, Japan)
  - 19 petaflop (#5), 2MW
  - Intel Xeon CPUs, NVIDIA Volta V100 GPUs
- Piz Daint (CSCS Switzerland)
  - 20 petaflop (#6), 2MW
  - Cray XC50 with NVIDIA P100 GPUs

Multi-GPU computing

How does one use such machines?

Depends on hardware choice:

- for single machines, use shared-memory multithreaded host application
- for clusters / supercomputers, use distributed-memory MPI message-passing

MPI approach

In the MPI approach:

- one GPU per MPI process (nice and simple)
- distributed-memory message passing between MPI processes (tedious but not difficult)
- scales well to very large applications
- main difficulty is that the user has to partition their problem (break it up into separate large pieces for each process) and then explicitly manage the communication

- note: should investigate GPU Direct for maximum performance in message passing

Multi-user support

What if different processes try to use the same device?

Depends on system compute mode setting (section 3.4):

- in “default” mode, each process uses the fastest device
  - good when one very fast card, and one very slow
  - not good when you have 2 identical fast GPUs
- in “exclusive” mode, each process is assigned to first unused device; it’s an error if none are available
- `cudaGetDeviceProperties` reports mode setting
- mode can be changed by sys-admin using `nvidia-smi` command line utility

Odds and ends

Appendix B.21: loop unrolling

If you have a loop:

```
for (int k=0; k<4; k++) a[i] += b[i];
```

then `nvcc` will automatically unroll this to give

```
a[0] += b[0];
a[1] += b[1];
a[2] += b[2];
a[3] += b[3];
```

to avoid cost of incrementing and looping.

The pragma

```
#pragma unroll 5
```

will also force unrolling for loops without explicit limits

Odds and ends

Appendix B.2.5: `__restrict__` keyword

```
void foo(const float* __restrict__ a,
         const float* __restrict__ b,
               float* __restrict__ c) {
 c[0] = a[0] * b[0];
 c[1] = a[0] * b[0];
 c[2] = a[0] * b[0] * a[1];
 c[3] = a[0] * a[1];
 c[4] = a[0] * b[0];
 c[5] = b[0];
 ...
}
```

The qualifier asserts that there is no overlap between `a,b,c` , so the compiler can perform more optimisations

Odds and ends

Appendix E.3.3.3: `volatile` keyword

Tells the compiler the variable may change at any time, so not to re-use a value which may have been loaded earlier and apparently not changed since.

This can sometimes be important when using shared memory

Odds and ends

Compiling:

- `Makefile` for first few practicals uses `nvcc` to compile both the host and the device code
  - internally it uses `gcc` for the host code, at least by default
  - device code compiler based on open source LLVM compiler
- sometimes, prefer to use other compilers (e.g. `icc`, `mpicc`) for main code that doesn't have any CUDA calls
- this is fine provided you use `-fPIC` flag for position-independent-code (don't know what this means but it ensures interoperability)
- can also produce libraries for use in the standard way

Odds and ends

Prac 6 Makefile:

```
INC   := -I$(CUDA_HOME)/include -I.
LIB   := -L$(CUDA_HOME)/lib64 -lcudart
FLAGS := --ptxas-options=-v --use_fast_math

main.o: main.cpp
	g++ -c -fPIC -o main.o main.cpp

prac6.o: prac6.cu
	nvcc prac6.cu -c -o prac6.o $(INC) $(FLAGS)

prac6: main.o prac6.o
	g++ -fPIC -o prac6 main.o prac6.o $(LIB)
```

Odds and ends

Prac 6 Makefile to create a library:

```
INC    := -I$(CUDA)/include -I.
LIB    := -L$(CUDA)/lib64 -lcudart
FLAGS := --ptxas-options=-v --use_fast_math

main.o: main.cpp
     g++ -c -fPIC -o main.o main.cpp

prac6.a: prac6.cu
     nvcc prac6.cu -lib -o prac6.a $(INC) $(FLAGS)

prac6a: main.o prac6.a
     g++ -fPIC -o prac6a main.o prac6.a $(LIB)
```

Odds and ends

Other compiler options:

- `-arch=sm_35`
  specifies GPU architecture
- `-maxrregcount=n`
  asks compiler to generate code using at most `n` registers; compiler may ignore this if it's not possible, but it may also increase use up to this limit

  This is much less important now since threads can have up to 255 registers

Odds and ends

Launch bounds (B.20):

- `-maxrregcount` modifies default for all kernels
- each kernel can be individually controlled by specifying launch bounds heuristics

```
__global__ void
__launch_bounds__(maxThreadsPerBlock,
                  minBlocksPerMultiprocessor)
MyKernel(...)
```

Conclusions

This lecture has discussed a number of more advanced topics

As a beginner, you can ignore almost all of them

As you get more experienced, you will probably want to start using some of them to get the very best performance