GPU Performance Optimisation

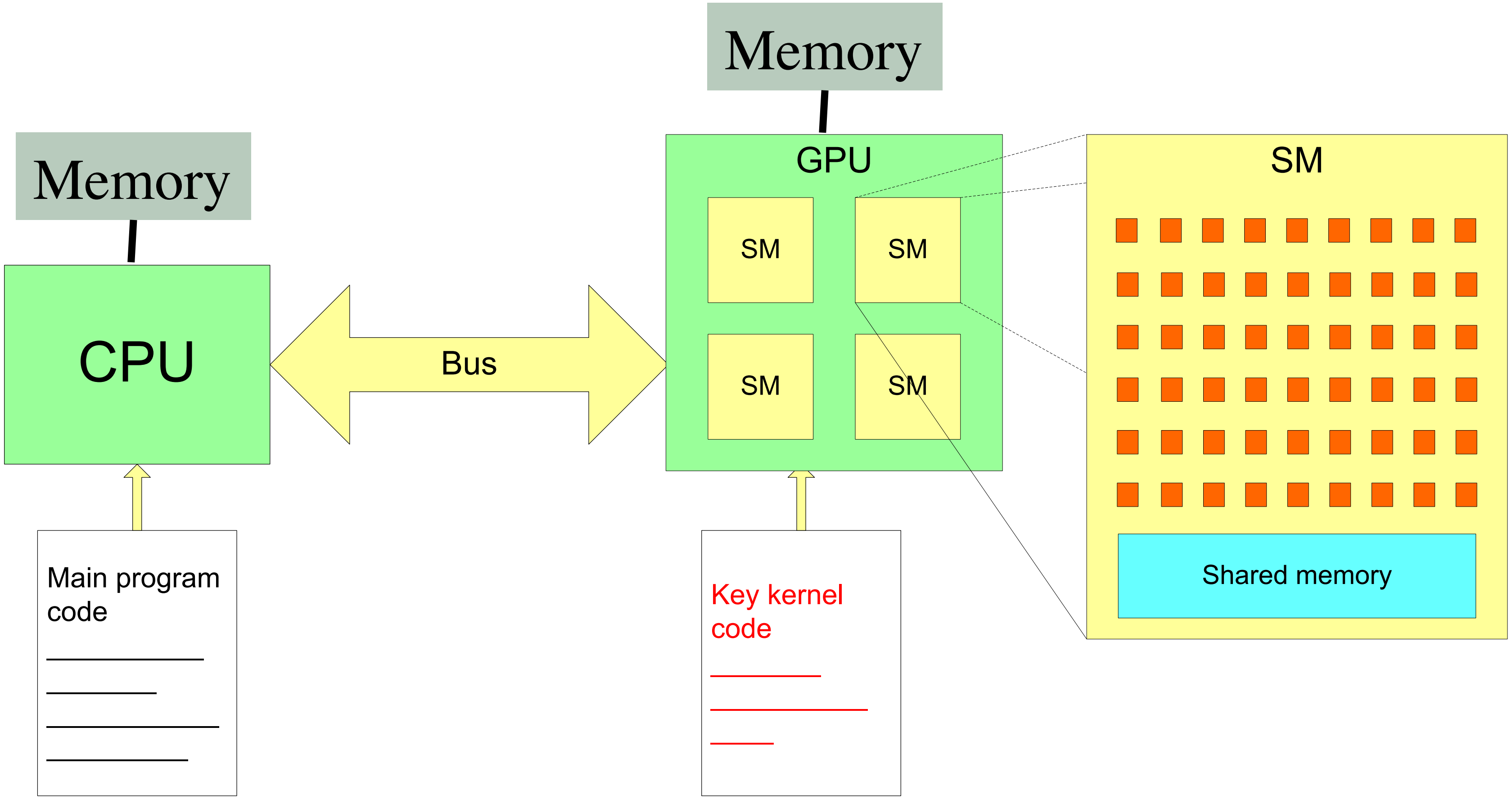
Alan Gray

EPCC

The University of Edinburgh

Hardware

NVIDIA accelerated system:



GPU performance inhibitors

- Copying data to/from device
- Device under-utilisation/ GPU memory latency
- GPU memory bandwidth
- Code branching

This lecture will address each of these

- And advise how to maximise performance
- Concentrating on NVIDIA, but many concepts will be transferable to e.g. AMD

Host – Device Data Copy

- CPU (host) and GPU (device) have separate memories.
- All data read/written on the device must be copied to/from the device (over PCIe bus).
  - This very expensive
- Must try to minimise copies
  - Keep data resident on device
    - May involve porting more routines to device, even if they are not computationally expensive
  - Might be quicker to calculate something from scratch on device instead of copying from host

Data copy optimisation example

```
Loop over timesteps
   inexpensive_routine_on_host(data_on_host)
   copy data from host to device
   expensive_routine_on_device(data_on_device)
   copy data from device to host
End loop over timesteps
```

- Port inexpensive routine to device and move data copies outside of loop

```
copy data from host to device
Loop over timesteps
   inexpensive_routine_on_device(data_on_device)
   expensive_routine_on_device(data_on_device)
End loop over timesteps
copy data from device to host
```

Exposing parallelism

- GPU performance relies on parallel use of many threads
  - Degree of parallelism much higher than a CPU
- Effort must be made to expose as much parallelism as possible within application
  - May involve rewriting/refactoring
- If significant sections of code remain serial, effectiveness of GPU acceleration will be limited (Amdahl’s law)

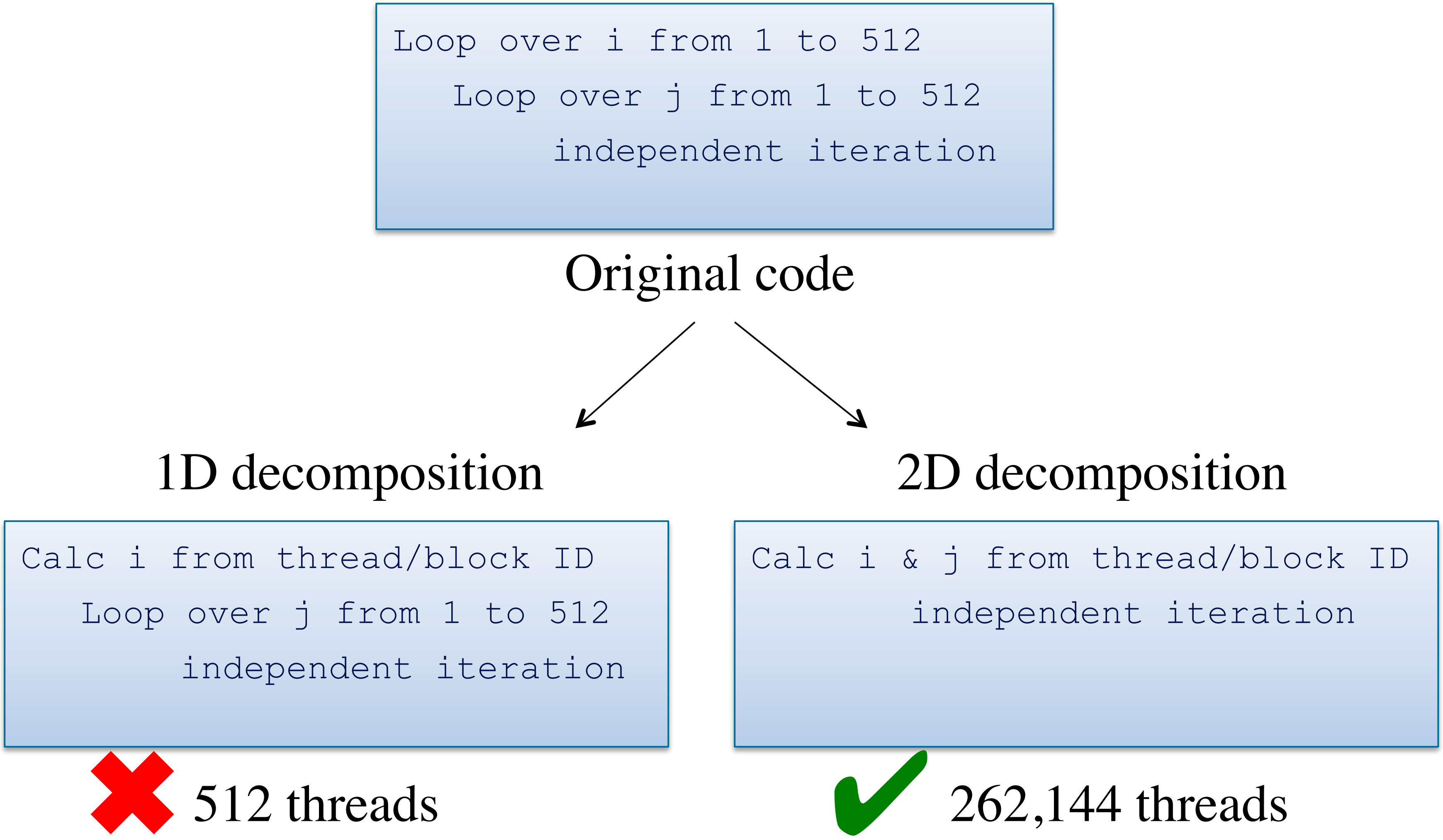
Occupancy and Memory Latency hiding

- Programmer decomposes loops in code to threads
  - Obviously, there must be at least as many total threads as cores, otherwise cores will be left idle.
- For best performance, actually want

  #threads >> #cores

- Accesses to GPU memory have several hundred cycles latency
  - When a thread stalls waiting for data, if another thread can switch in this latency can be hidden.
- NVIDIA GPUs have very fast thread switching, and support many concurrent threads

Exposing parallelism example



```
Loop over i from 1 to 512
   Loop over j from 1 to 512
        independent iteration
```

Original code

1D decomposition

```
Calc i from thread/block ID
   Loop over j from 1 to 512
        independent iteration
```

✗ 512 threads

2D decomposition

```
Calc i & j from thread/block ID
        independent iteration
```

✓ 262,144 threads

Memory coalescing

- GPUs have high *peak* memory bandwidth
- Maximum memory bandwidth is only achieved when data is accessed for multiple threads in a single transaction: *memory coalescing*
- To achieve this, ensure that **consecutive threads access consecutive memory locations**
- Otherwise, memory accesses are serialised, significantly degrading performance
  - Adapting code to allow coalescing can dramatically improve performance

Memory coalescing example

- *consecutive threads* are those with consecutive `threadIdx.x` or `threadidx%x` values
- Do consecutive threads access consecutive memory locations?

C:

```
index = blockIdx.x*blockDim.x + threadIdx.x;
output[index] = 2*input[index];
```

F:

```
index = (blockidx%x-1)*blockdim%x + threadidx%x
result(index) = 2*input(index)
```

✓ Coalesced. Consecutive `threadIdx` values correspond to consecutive `index` values

Memory coalescing examples

- Do consecutive threads read consecutive memory locations?
- In C, outermost index runs fastest: `j` here

```
i = blockIdx.x*blockDim.x + threadIdx.x;
for (j=0; j<N; j++)
    output[i][j]=2*input[i][j];
```

✗ Not Coalesced. Consecutive `threadIdx.x` corresponds to consecutive `i` values

```
j = blockIdx.x*blockDim.x + threadIdx.x;
for (i=0; i<N; i++)
    output[i][j]=2*input[i][j];
```

✓ Coalesced. Consecutive `threadIdx.x` corresponds to consecutive `j` values

Memory coalescing examples

- Do consecutive threads read consecutive memory locations?
- In Fortran, innermost index runs fastest: `i` here

```
j = (blockIdx%x-1)*blockDim%x + threadIdx%x

do i=1, 256
  output(i,j) = 2*input(i,j)
end do
```

✖ Not Coalesced. Consecutive `threadIdx%x` corresponds to consecutive `j` values

```
i = (blockIdx%x-1)*blockDim%x + threadIdx%x

do j=1, 256
  output(i,j) = 2*input(i,j)
end do
```

✔ Coalesced. Consecutive `threadIdx%x` corresponds to consecutive `i` values

Memory coalescing examples

- What about when using 2D or 3D CUDA decompositions?
  - Same procedure. X component of `threadIdx` is always that which increments with consecutive threads
  - E.g., for matrix addition, coalescing achieved as follows:

C:

```
 int j = blockIdx.x * blockDim.x + threadIdx.x;
 int i = blockIdx.y * blockDim.y + threadIdx.y;

 c[i][j] = a[i][j] + b[i][j];
```

F:

```
i = (blockidx%x-1)*blockdim%x + threadidx%x
j = (blockidx%y-1)*blockdim%y + threadidx%y

c(i,j) = a(i,j) + b(i,j)
```

Code Branching

- On NVIDIA GPUs, there are less instruction scheduling units than cores
- Threads are scheduled in groups of 32, called a *warp*
- Threads within a warp must execute the same instruction in lock-step (on different data elements)
- The CUDA programming allows branching, but this results in all cores following all branches
  - With only the required results saved
  - This is obviously suboptimal
- Must avoid intra-warp branching wherever possible (especially in key computational sections)

Branching example

- E.g you want to split your threads into 2 groups:

```
i = blockIdx.x*blockDim.x + threadIdx.x;
if (i%2 == 0)
   …
else
   …
```

✗ Threads within warp diverge

```
i = blockIdx.x*blockDim.x + threadIdx.x;
if ((i/32)%2 == 0)
   …
else
   …
```

✓ Threads within warp follow same path

CUDA Profiling

- Simply set COMPUTE_PROFILE environment variable to 1
- Log file, e.g. cuda_profile_0.log created at runtime: timing information for kernels and data transfer

```
# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 Tesla M1060
# CUDA_CONTEXT 1
# TIMESTAMPFACTOR fffff6e2e9ee8858
method,gputime,cputime,occupancy
method=[ memcpyHtoD ] gputime=[ 37.952 ] cputime=[ 86.000 ]
method=[ memcpyHtoD ] gputime=[ 37.376 ] cputime=[ 71.000 ]
method=[ memcpyHtoD ] gputime=[ 37.184 ] cputime=[ 57.000 ]
method=[ _Z23inverseEdgeDetect1D_colPfS_S_ ] gputime=[ 253.536 ] cputime=[ 13.00
0 ] occupancy=[ 0.250 ]
...
```

- Possible to output more metrics (cache misses etc)
  - See doc/Compute_Profiler.txt file in main CUDA installation

Conclusions

- GPU architecture offers higher Floating Point and memory bandwidth performance over leading CPUs
- There are a number of factors which can inhibit application performance on the GPU.
  - And a number of steps which can be taken to circumvent these inhibitors
    - Some of these may require significant development/tuning for real applications
- It is important to have a good understanding of the application, architecture and programming model.