GPU Programming

Alan Gray

EPCC

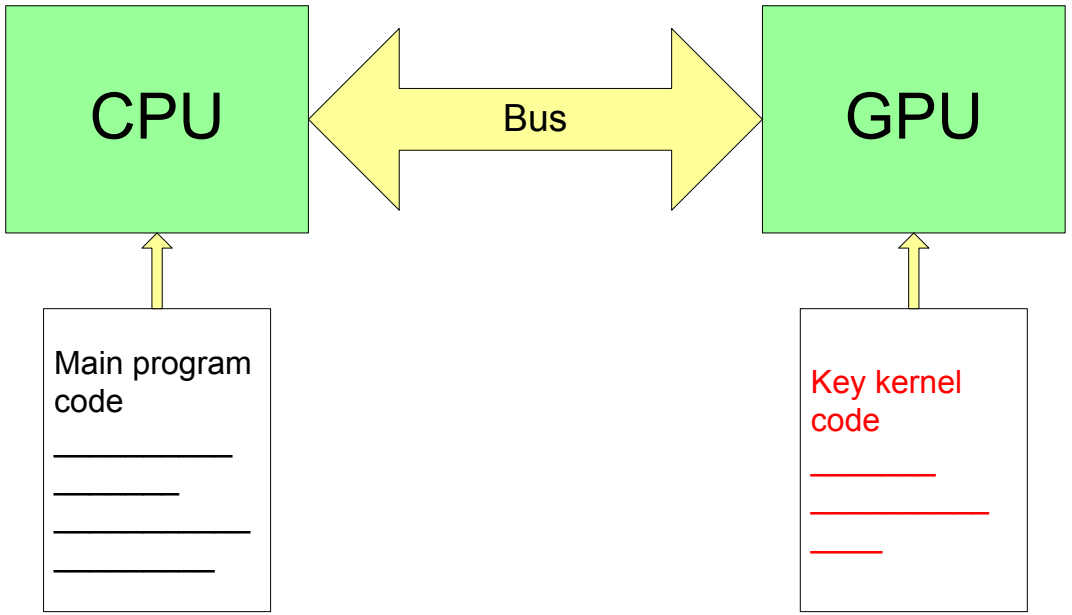
The University of Edinburgh

Overview

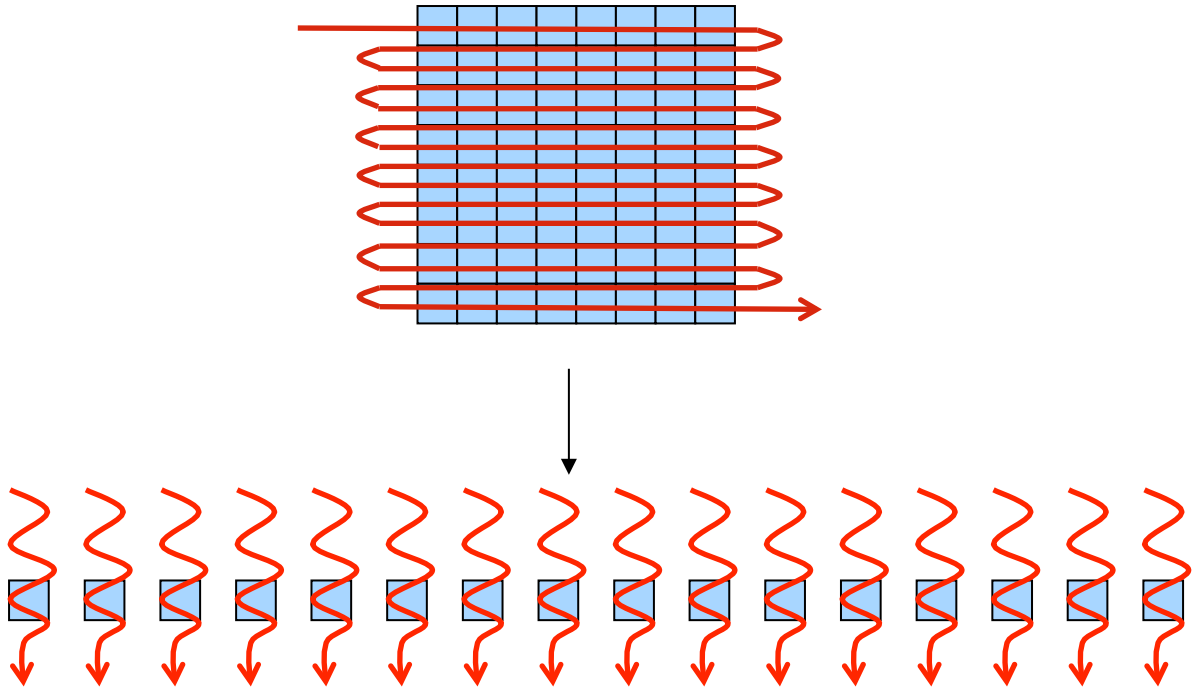
- Motivation and need for CUDA
- Introduction to CUDA
  - CUDA kernels, decompositions
  - CUDA memory management
  - C and Fortran
- OpenCL

NVIDIA CUDA

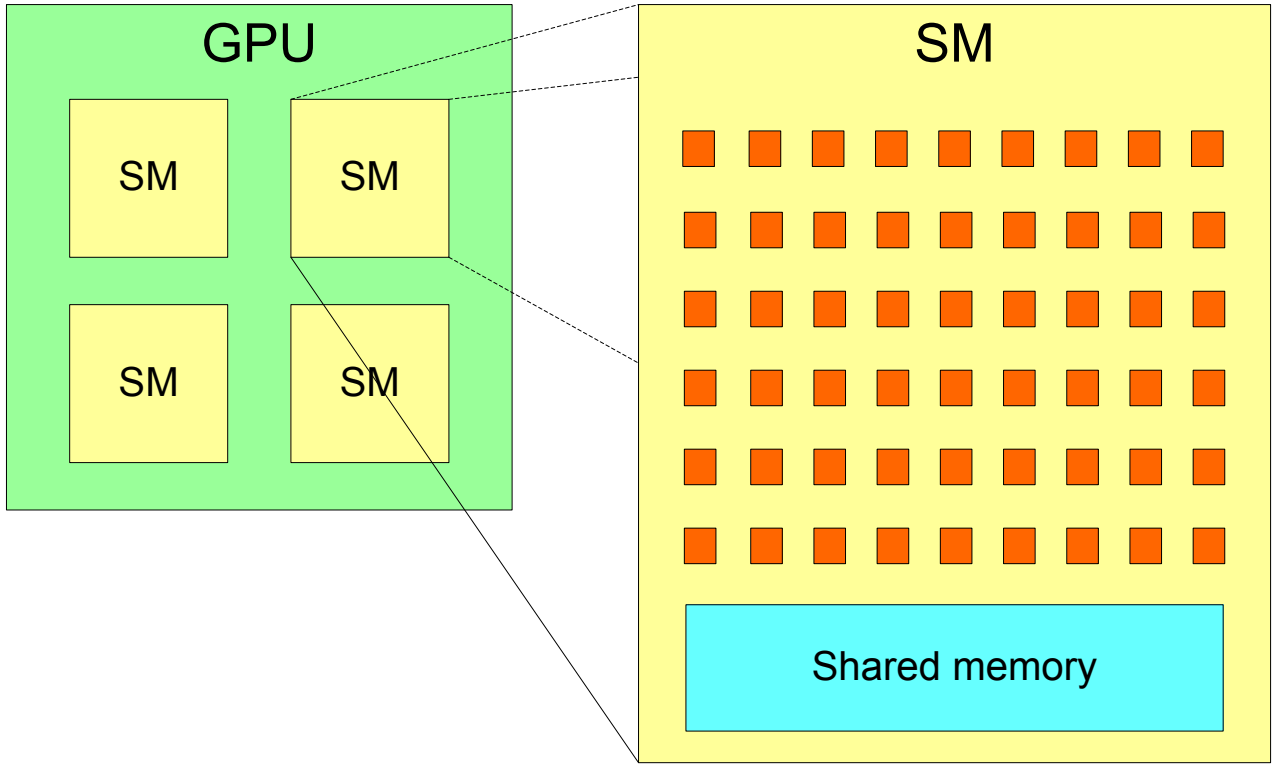
- Traditional languages alone are not sufficient for programming GPUs
- CUDA allows NVIDIA GPUs to be programmed in C/C++ or Fortran
  - defines language extensions for defining *kernels*
  - kernels execute in multiple threads concurrently on the GPU
  - provides API functions for e.g. device memory management


CPU
Bus
GPU
Main program code
Key kernel code

GPGPU: Stream Computing



- Data set decomposed into a *stream* of elements
- A single computational function (*kernel*) operates on each element
  - "thread" defined as execution of kernel on one data element
- Multiple cores can process multiple elements in parallel
  - i.e. many threads running in parallel
- Suitable for data-parallel problems



- NVIDIA GPUs have a 2-level hierarchy:
  - Multiple *Streaming Multiprocessors (*SMs), each with multiple *cores*
- The number of SMs, and cores per SM, varies across generations

- In CUDA, this is abstracted as *Grid* of *Thread Blocks*
  - The multiple **blocks** in a grid map onto the multiple **SMs**
    - Each block in a grid contains multiple **threads**, mapping onto the **cores** in an SM
- We don't need to know the exact details of the hardware (number of SMs, cores per SM).
  - Instead, *oversubscribe*, and system will perform scheduling automatically
    - Use more blocks than SMs, and more threads than cores
  - Same code will be portable and efficient across different GPU versions.

CUDA dim3 type

- CUDA introduces a new `dim3` type
  - Simply contains a collection of 3 integers, corresponding to each of X,Y and Z directions.

C:

```
dim3 my_xyz_values(xvalue,yvalue,zvalue);
```

Fortran:

```
type(dim3) :: my_xyz_values
my_xyz_values = dim3(xvalue,yvalue,zvalue)
```

- X component can be accessed as follows:

C: `my_xyz_values.x`

Fortran: `my_xyz_values%x`

And similar for Y and Z

- E.g. for

```
my_xyz_values = dim3(6,4,12)
```

then `my_xyz_values%z` has value 12


CUDA
HOTEL
CAFÉ CRU
TERRACE

Analogy

- You check in to the hotel, as do your classmates
  - Rooms allocated in order
- Receptionist realises hotel is less than half full
  - Decides you should all move from your room number *i* to room number *2i*
  - so that no-one has a neighbour to disturb them

- Serial Solution:
  - Receptionist works out each new number in turn

- Parallel Solution:

"*Everybody: check your room number. Multiply it by 2, and move to that room.*"

- Serial solution:

```
for (i=0;i<N;i++){
  result[i] = 2*i;
}
```

- We can parallelise by assigning each iteration to a separate CUDA thread.

CUDA C Example

```
__global__ void myKernel(int *result)
{
  int i = threadIdx.x;
  result[i] = 2*i;
}
```

- Replace loop with function
- Add **__global__** specifier
  - To specify this function is to form a GPU kernel
- Use internal CUDA variables to specify array indices
  - **threadidx.x** is an internal variable unique to each thread in a block.
    - X component of dim3 type. Since our problem is 1D, we are not using the Y or Z components (more later)

CUDA C Example

- And launch this kernel by calling the function
  - *on multiple CUDA threads using <<<…>>> syntax*

```
dim3 blocksPerGrid(1,1,1); //use only one block
dim3 threadsPerBlock(N,1,1); //use N threads in the block

myKernel<<<blocksPerGrid, threadsPerBlock>>>(result);
```

CUDA FORTRAN Equivalent

Kernel:

```
attributes(global) subroutine myKernel(result)
    integer, dimension(*) :: result
    integer :: i
    i = threadidx%x
    result(i) = 2*i
end subroutine
```

Launched as follows:

```
blocksPerGrid = dim3(1, 1, 1)
threadsPerBlock = dim3(N, 1, 1)
call myKernel <<<blocksPerGrid, threadsPerBlock>>> (result)
```

CUDA C Example

- The previous example only uses 1 block, i.e. only 1 SM on the GPU, so performance will be very poor. In practice, we need to use multiple blocks to utilise all SMs, e.g.:

```
__global__ void myKernel(int *result)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  result[i] = 2*i;
}


...
dim3 blocksPerGrid(N/256,1,1); //assuming 256 divides N exactly
dim3 threadsPerBlock(256,1,1);

myKernel<<<blocksPerGrid, threadsPerBlock>>>(result);
...
```

FORTRAN

```
attributes(global) subroutine myKernel(result)
    integer, dimension(*) :: result
    integer :: i
    i = (blockidx%x-1)*blockdim%x + threadidx%x
    result(i) = 2*i
end subroutine
...
blocksPerGrid = dim3(N/256, 1, 1) !assuming 256 divides N exactly
threadsPerBlock = dim3(256, 1, 1)
call myKernel <<<blocksPerGrid, threadsPerBlock>>> (result)
...
```

- We have chosen to use 256 threads per block, which is typically a good number (see practical).

CUDA C Example

- More realistic 1D example: vector addition

```
__global__ void vectorAdd(float *a, float *b, float *c)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  c[i] = a[i] + b[i];
}


...
dim3 blocksPerGrid(N/256,1,1); //assuming 256 divides N exactly
dim3 threadsPerBlock(256,1,1);

vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(a, b, c);
...
```

CUDA FORTRAN Equivalent

```
attributes(global) subroutine vectorAdd(a, b, c)
    real, dimension(*) :: a, b, c
    integer :: i
    i = (blockidx%x-1)*blockdim%x + threadidx%x
    c(i) = a(i) + b(i)
end subroutine
...
blocksPerGrid = dim3(N/256, 1, 1)
threadsPerBlock = dim3(256, 1, 1)
call vectorAdd <<<blocksPerGrid, threadsPerBlock>>> (a, b, c)
...
```

CUDA C Internal Variables

For a 1D decomposition (e.g. the previous examples)

- `blockDim.x`: Number of threads per block
  - Takes value 256 in previous example
- `threadIdx.x`: unique to each thread in a block
  - Ranges from 0 to 255 in previous example
- `blockIdx.x`: Unique to every block in the grid
  - Ranges from 0 to (N/256 - 1) in previous example

CUDA Fortran Internal Variables

For a 1D decomposition (e.g. the previous example)

- `blockDim%x`: Number of threads per block
  - Takes value 256 in previous example
- `threadIdx%x`: unique to each thread in a block
  - Ranges from 1 to 256 in previous example
- `blockIdx%x`: Unique to every block in the grid
  - Ranges from 1 to (N/256) in previous example

2D Example

- 2D or 3D CUDA decompositions also possible, e.g. for matrix addition (2D):

```
__global__ void matrixAdd(float a[N][N], float b[N][N], float c[N][N])
{
  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];
}

int main()
{
  dim3 blocksPerGrid(N/16,N/16,1); // (N/16)x(N/16) blocks/grid (2D)
  dim3 threadsPerBlock(16,16,1); // 16x16=256 threads/block (2D)
  matrixAdd<<<blocksPerGrid, threadsPerBlock>>>(a, b, c);
}
```

CUDA Fortran Equivalent

```
! Kernel declaration
attributes(global) subroutine matrixAdd(N, a, b, c)
    integer, value :: N
    real, dimension(N,N) :: a, b, c
    integer :: i, j
    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)
end subroutine

! Kernel invocation
blocksPerGrid = dim3(N/16, N/16, 1) ! (N/16)x(N/16) blocks/grid (2D)
threadsPerBlock = dim3(16, 16, 1) ! 16x16=256 threads/block (2D)
call matrixAdd <<<blocksPerGrid, threadsPerBlock>>> (N, a, b, c)
```

Memory Management - allocation

- The GPU has a separate memory space from the host CPU
- Data accessed in kernels must be on GPU memory
- Need to manage GPU memory and copy data to and from it explicitly
- cudaMalloc is used to allocate GPU memory
- cudaFree releases it again

```
float *a;
cudaMalloc(&a, N*sizeof(float));
…
cudaFree(a);
```

Memory Management - cudaMemcpy

- Once we've allocated GPU memory, we need to be able to copy data to and from it
- cudaMemcpy does this:

```
cudaMemcpy(array_device, array_host, N*sizeof(float),
      cudaMemcpyHostToDevice);
cudaMemcpy(array_host, array_device, N*sizeof(float),
                  cudaMemcpyDeviceToHost);
```

- The first argument always corresponds to the *destination* of the transfer.
- Transfers between host and device memory are relatively slow and can become a bottleneck, so should be minimised when possible

CUDA FORTRAN – Data management

- Data management is more intuitive than CUDA C
  - Because Fortran has array syntax, and also compiler knows if a pointer is meant for CPU or GPU memory
- Can use allocate() and deallocate() as for host memory

```
real, device, allocatable, dimension(:) :: d_a
allocate( d_a(N) )
…
deallocate ( d_a )
```

- Can copy data between host and device using assignment

```
d_a = a(1:N)
```

- Or can instead use CUDA API (similar to C), e.g.

```
istat = cudaMemcpy(d_a, a, N)
```

Synchronisation between host and device

- Kernel calls are *non-blocking.* This means that the host program continues immediately after it calls the kernel
  - Allows overlap of computation on CPU and GPU
- Use `cudaThreadSynchronize()` to wait for kernel to finish

```
vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(a, b, c);
//do work on host (that doesn't depend on c)
cudaThreadSynchronise(); //wait for kernel to finish
```

- Standard `cudaMemcpy` calls are blocking
  - Non-blocking variants exist

Synchronisation between CUDA threads

- Within a kernel, to syncronise between threads in the same block use the `syncthreads()` call
- Therefore, threads in the same block can communicate through memory spaces that they share, e.g. assuming `x` local to each thread and `array` in a shared memory space

```
if (threadIdx.x == 0) array[0]=x;
syncthreads();
if (threadIdx.x == 1) x=array[0];
```

- It is *not possible* to communicate between different blocks in a kernel: must instead exit kernel and start a new one

Unified Memory

- The GPU has a separate memory space from the host CPU
- Recent advances in CUDA and in hardware allow this aspect to be largely hidden from the programmer with automatic data movement.
  - "Unified Memory"
- **HOWEVER** for performance it is often necessary to manually manage these distinct spaces.
  - And this lecture has shown how to do this
- But unified memory can be useful to help get codes running quickly
  - Possibly an incremental stepping stone to manual data management

Unified Memory

- With our previous examples, for each array we maintained both a host and device copy.
  - The device copy was allocated using `cudaMalloc`
  - And we used `cudaMemcpy` to transfer
- With Unified Memory, a single copy can be accessed on either the CPU or GPU if allocated using the `cudaMallocManaged` call (and freed using `cudaFree`), e.g.

```
float *array;
cudaMallocManaged(&array, N*sizeof(float));
// array can now be accessed either on host or device
... setup, launch kernel, process output ...
cudaFree(array);
```

- The data will be automatically transferred to/from the GPU as necessary.

Multi-GPU with MPI

- In this lecture, you have seen how to adapt a C or Fortran code to utilise a GPU using CUDA
- We can combine with MPI, to utilise multiple GPUs (possibly distributed across multiple nodes)
- Simply set the number of MPI tasks equal to the number of nodes
  - And each MPI task controls its own GPU
- MPI communications: can either
  - Explicitly copy from/to GPU with CUDA before/after any MPI communications which access host data
  - Use CUDA-aware MPI (if available) such that MPI directly accesses GPU memory

Compiling CUDA Code

- CUDA C code is compiled using `nvcc`:

```
nvcc -o example example.cu
```

- CUDA Fortran is compiled using PGI compiler
  - either use `.cuf` filename extension for CUDA files
  - and/or pass `-Mcuda` to the compiler command line

```
pgf90 -Mcuda -o example example.cuf
```

OpenCL

- Open Compute Language (OpenCL): “The Open Standard for Heterogeneous Parallel Programming”
  - Open cross-platform framework for programming modern multicore and heterogeneous systems
- Supports wide range of applications and architectures, including GPUs
  - Supported on NVIDIA Tesla + AMD FireStream
- See http://www.khronos.org/opencl/

OpenCL vs CUDA on NVIDIA

- NVIDIA support both CUDA and OpenCL as APIs to the hardware.
  - But put much more effort into CUDA
  - CUDA more mature, well documented and performs better
- OpenCL and C for CUDA conceptually very similar
  - Very similar abstractions, basic functionality etc
  - Different names e.g. “Thread” CUDA -> “Work Item” (OpenCL)
  - Porting between the two should in principle be straightforward
- OpenCL is a lower level API than C for CUDA
  - More work for programmer
- OpenCL obviously portable to other systems
  - But in reality work will still need to be done for efficiency on different architecture
- OpenCL may well catch up with CUDA given time

Summary

- Traditional languages alone are not sufficient for programming GPUs
- CUDA allows NVIDIA GPUs to be programmed using C/C++ or Fortran
  - defines language extensions and APIs to enable this
- We introduced the key CUDA concepts and gave examples
- OpenCL provides another means of programming GPUs in C
  - conceptually similar to CUDA, but less mature and lower-level
  - supports other hardware as well as NVIDIA GPUs