Alan Gray, James Perry

EPCC

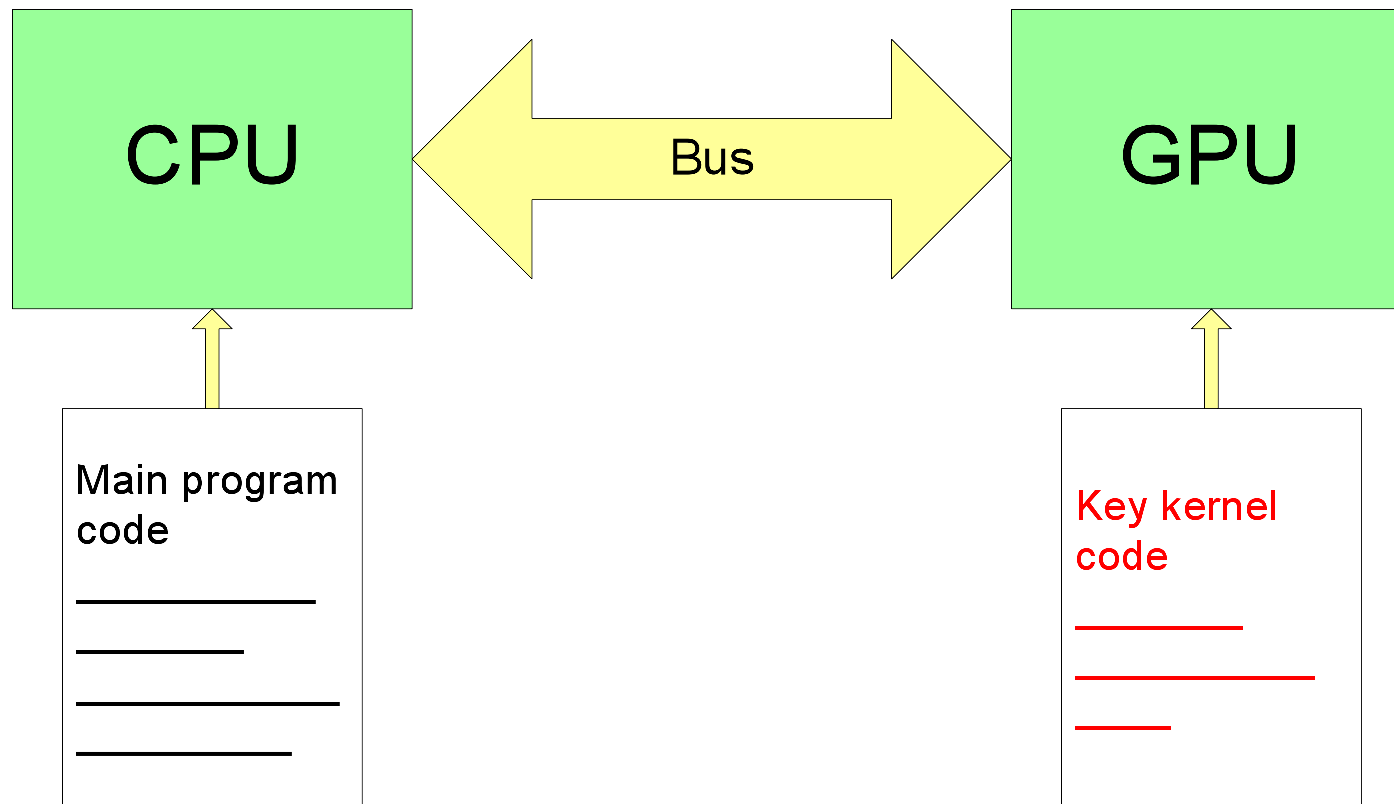
The University of Edinburgh

Contents

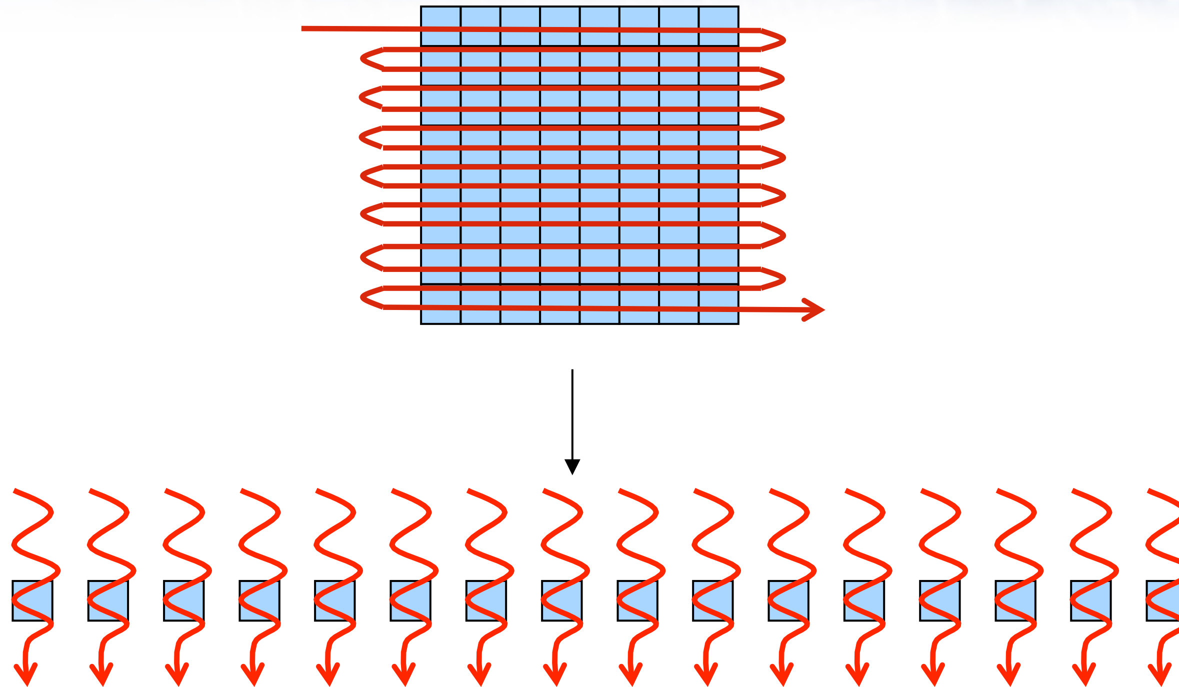
- NVIDIA CUDA C
  - Proprietary interface to NVIDIA architecture
- CUDA Fortran
  - Provided by PGI
- OpenCL
  - Cross platform API

NVIDIA CUDA

- CUDA allows NVIDIA GPUs to be programmed in C and C++
  - defines language extensions for defining *kernels*
  - kernels execute in multiple threads concurrently on the GPU
  - provides API functions for e.g. device memory management
- PGI provide a commercial Fortran version of CUDA

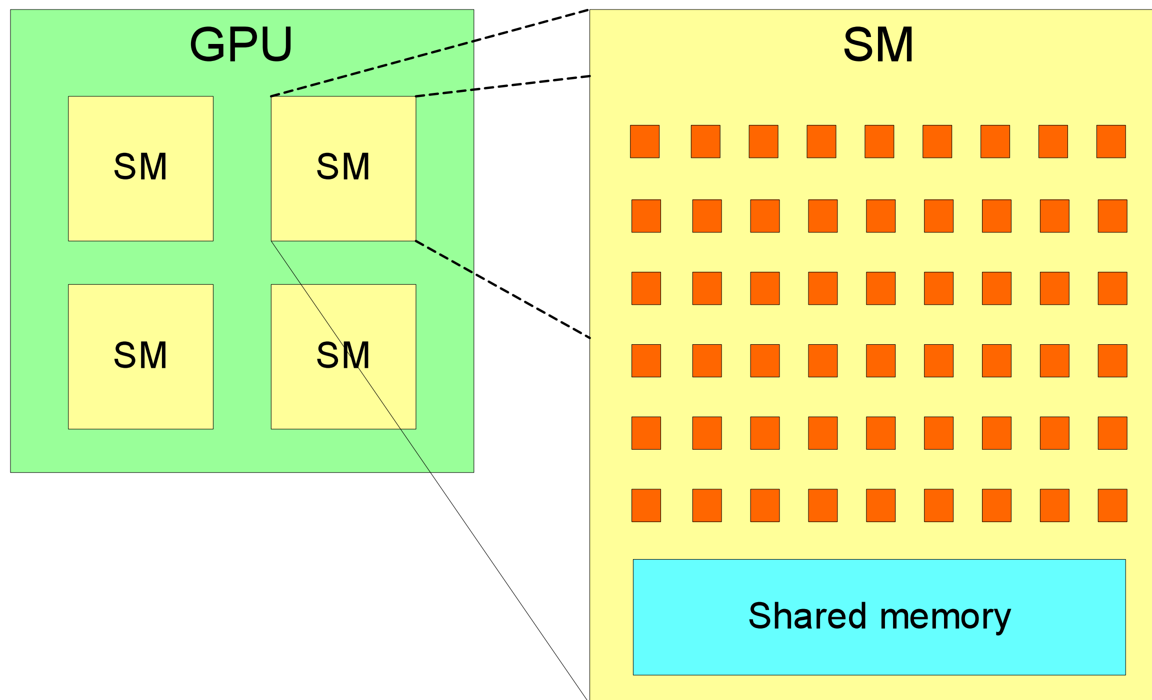

CPU
Bus
GPU
Main program code
Key kernel code

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 Stream Multiprocessors SMs
    - each with multiple cores



- 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.

- Example: vector addition. Consider the following loop:

```
for (i=0;i<N;i++){
  c[i] = a[i] + b[i];
}
```

- This is parallel in nature: we can assign each iteration to a separate CUDA thread.

CUDA C Example

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

CUDA C Example

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

```
int blocksPerGrid=1; //use only one block
int threadsPerBlock=N; //use N threads in the block

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

CUDA C Example

- The previous example only uses 1 block, i.e. only 1 SM on the GPU. In practice, we need to use multiple blocks, e.g.:

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


...
int blocksPerGrid=N/256; //assuming 256 divides N exactly
int threadsPerBlock=256;

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

- `blockIdx.x`: Unique to every block in the grid
- `blockDim.x`: Number of threads per block
- We have chosen to use 256 threads per block, which is typically a good number (see practical).

CUDA C Example

- You can think of this as restructuring the original loop

```
for (i=0;i<N;i++){
  c[i] = a[i] + b[i];
}
```

as

```
for (i0=0;i0<(N/256);i0++){
  for (i1=0;i1<256;i1++){
    i = i0*256 + i1;
    c[i] = a[i] + b[i];
  }
}
```

and parallelising inner loop over threads in a block, outer loop over blocks.

2D Example

- The previous examples were one dimensional.
- Each thread block can be 1D, 2D or 3D to best fit the algorithm, e.g. for matrix addition:

```
__global__ void matrixAdd(float a[N][N], float b[N][N], float c[N][N])
{
  int i = threadIdx.x;
  int j = threadIdx.y;

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

int main()
{
 dim3 blocksPerGrid(1); /* 1 block per grid (1D) */
 dim3 threadsPerBlock(N, N); /* NxN threads per block (2D) */
 matrixAdd<<<blocksPerGrid, threadsPerBlock>>>(a, b, c);
}
```

- **`dim3`** is a CUDA type, containing 3 integers (x,y and z components)

Multiple Block 2D Example

- Grid can also be be 1D, 2D or 3D

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

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

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

Memory Management - allocation

- The GPU has a separate memory space from the host CPU
- We cannot simply pass normal C pointers to CUDA threads
- 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

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

CUDA FORTRAN

- Allows Fortran codes to take advantage of GPU acceleration
- Available in the commercial PGI Fortran compiler from the Portland Group
- Very similar to CUDA C
  - define kernels (subroutines) to be executed on the device with `global` attribute
  - invoke them using `<<< >>>` syntax
  - same API functions available (`cudaMalloc`, `cudaFree`, `cudaMemcpy`, etc.)
  - Some features are more intuitive than CUDA C, e.g. can use Fortran array syntax to copy to/from GPU instead of API functions

CUDA FORTRAN Example

```
! Kernel declaration
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

! Kernel invocation
blocksPerGrid = dim3(N/256, 1, 1)
threadsPerBlock = dim3(256, 1, 1)
call vectorAdd <<<blocksPerGrid, threadsPerBlock>>> (a, b, c)
```

Multi-block 2D Example

```
! 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) = a(i) + b(i)
end subroutine

! Kernel invocation
blocksPerGrid = dim3(N/16, N/16, 1)
threadsPerBlock = dim3(16, 16, 1)
call matrixAdd <<<blocksPerGrid, threadsPerBlock>>> (a, b, c)
```

CUDA FORTRAN – Data management

- Can use allocate() and deallocate() as for host memory

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

- Can copy data between host and device using assignment

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

though this is a blocking operation. Can also use

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

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

Example OpenCL kernel

```
kernel void vectoradd(global const float *a, global
                      const float *b, global float *c)
{
    int i = get_global_id(0);
    c[i] = a[i] + b[i];
}
```

- `kernel` and `global` are OpenCL extensions to C
- `get_global_id` is OpenCL function
  - Performs similar role to CUDA's `blockIdx/threadIdx`
- Invoking kernels is a much more involved process than in CUDA

Summary

- CUDA allows NVIDIA GPUs to be programmed using C/C++
  - defines language extensions and APIs to enable this
- PGI provide commercial Fortran version of CUDA
- 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