Parallelising Scientific Codes Using Graphical Processing Units (GPUs) in OpenACC

Wadud Miah
Research Computing Group
Software Performance Lifecycle
Scientific Programming

• Early scientific codes were mainly sequential and were executed on single core CPUs;
• CPU performance accelerated from 1 MHz and have peaked at around 3.2 GHz;
• However, scientists want to a) refine and expand computational domains b) explore challenging scientific problems;
• Solution: parallel programming many-core architectures.
CPU Architecture

• L1, L2 and L3 cache and registers;
• Floating point and integer instruction units;
• SIMD (vector) instruction units.
GPU Architecture

• A GPU consists of multiple streaming multi-processors (SM) with local cache;
• Each SM has 32 lightweight cores that operate at around ≈ 1.15 GHz;
• The Nvidia Tesla C2050 has 448 cores (14 SMs).
CUDA - Compute Unified Device Architecture

- Nvidia have released CUDA as an API and an SDK which is an interface for Nvidia GPUs;
- This is a C based language with additional extensions for GPU programming;
- PGI (a compiler company) have developed CUDA for Fortran;
- There is also a simpler directive based programming interface to GPUs called OpenACC.
GPU Architecture (1)

• The GPU architecture consists of a host and device;
• Each have their own memory spaces.
GPU Architecture (2)

• Data has to be transferred from the host to the device via the PCIe 2.0 interconnect;
• The PCIe interconnect has high bandwidth and high latency. Approximately 3 times the latency of RAM;
• Ensure large amounts of data are transferred to the device to hide latency;
• Small amounts of work can be carried out on the CPU → hybrid computing.
Memory Management

- Memory is allocated on the host in the usual manner using `malloc()`;
- Memory is allocated on the GPU device using `cudaMalloc()` and a pointer to memory is passed as an argument - any attempts to de-reference will result in a fault;
- Data is freed on host using `free()` and `cudaFree()` on device.
Data Management

• Data is transferred from host to device and device to host using `cudaMemcpy()` and is asynchronous to hide latency;
• Direction of data movement is also specified using `cudaMemcpyHostToDevice` and `cudaMemcpyDeviceToHost`;
• There is a huge penalty on moving data to and from GPU, so ensure you move enough data to warrant this operation.
GPU Memory Structure

- Global memory - GPU main memory;
- Constant memory - for data that does not change;
- Shared memory - for threads within a block;
- Threads also have access to small set of registers;
- Speed: 1) registers 2) constant 3) shared 4) global.
CUDA Thread Structure (1)

- Threads are structured in grids and blocks;
- Threads in block can a) synchronise b) share data through low latency shared memory;
- Threads from different blocks can exchange data via global memory, but at a higher latency.
CUDA Thread Structure (2)

- Threads can be organised in one, two or three dimensional structures;
- Grids contain a number of blocks of threads.
CUDA Kernel Function

- The function that is to be executed in parallel by the GPU is called the kernel;
- It is defined and executed using CUDA extensions:
  - Executed on device and called by device: 
    ```c
    __device__ float deviceFunc();
    ```
  - Executed on device and called by host: 
    ```c
    __global__ void kernelFunc();
    ```
  - Executed on host and called by host: 
    ```c
    __host__ float hostFunc();
    ```
CUDA Kernel Execution

• Determine dimension of block and how many blocks are required:

```c
dim3 dimBlock(32, 16);
dim3 dimGrid(1, 1);
kernel_func<<<dimGrid, dimBlock>>>(M, N, P, 512);
```

The values in brackets are the arguments to `kernel_func`
CUDA Kernel Definition

• Example kernel function definition:

```c
__global__ void kernel( float *M,
    float *N, float *P, int size ) {

    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int tz = threadIdx.z;
    // execute work on (tx, ty, tz)
}
```
CUDA Index Variables

- Index variables determine which block of data thread is responsible for. Like thread ID in OpenMP and rank in MPI;
- Thread index variables available are:
  - threadIdx.x
  - threadIdx.y
  - threadIdx.z
- Two dimensional block index variables:
  - blockIdx.x
  - blockIdx.y
Block Synchronisation

• Threads in a block can be coordinated using a barrier synchronisation:
  ```c
  __syncthreads()
  ```

• Threads of different blocks cannot be synchronised - only way to achieve this is to wait for kernel call to complete.
Streaming Multiprocessors (SM)

- Threads are executed on streaming multiprocessors with 32 cores on each SM;
- Maximum 1024 threads can be executed on each SM or 8 blocks (with 128 threads).
Thread Scheduling - Warps

• Threads are scheduled to be executed on streaming multiprocessors in batches called warps which typically contains 32 threads;
• For 512 threads which is the maximum for a block, this is divided into 512/32 = 16 warps;
• Large number of warps are required to hide host to GPU latency - as one warp is waiting for data another warp can be executed on a streaming multiprocessor.
CUDA Libraries and Applications

- CUBLAS - Basic Linear Algebra Subprograms;
- CUFFT - Fast Fourier Transform;
- LAPACK, MAGMA - Numerical Linear Algebra;
- Matlab CUDA toolbox;
- NAG numerical routines for GPU;
- PyCUDA - Python interface to CUDA;
- Amber, Gromacs and DL_POLY - Molecular dynamics;
- WRF and MITgcm - Weather and climate forecasting.
Open Acceleration (OpenACC)

- CUDA programming can be difficult, but there is an easier method available;
- This is called Open Acceleration (OpenACC) which is a directive based programming method much like OpenMP;
- Blocks of code are prefixed with special directives for GPU parallelisation;
- The compiler will automatically spawn blocks of threads for execution on the GPU.
Developer Assisted Acceleration

• Developers will have to assist the compiler to accelerate their codes;
• Auto parallelisation just does not work - the days of writing generic codes are history;
• Developer has intimate knowledge of how their codes work which helps the compiler;
• This is the path to code scalability for exascale computing ($10^{18}$ FLOP/s);
• The PGI compiler version 12.3 and above supports OpenACC (Grace has 12.5).
Incremental Code Development

- OpenACC provides an incremental style of parallel software development;
- Test individual modifications to code;
- OpenACC parallelisation is added during the compilation and linking phase;
- PGI flag: \(-\text{acc}\)
- To switch off OpenACC, simply re-compile without the above switch.
Accelerator Region

- Has a single point of entry and a single point of exit;
- Number of blocks and threads fixed during parallel execution;
- No jumps or goto statements are allowed within parallel regions;
- All threads synchronise at the end of the parallel region.
OpenACC Syntax

• OpenACC uses compiler directives to control GPU thread parallelism;
• In Fortran we use a sentinel and directive with optional clauses (case insensitive):
  
  !$ACC directive [clauses]
  C$ACC directive [clauses]
  *$ACC directive [clauses]

• In C/C++ we use pragmas and directive with optional clauses (case sensitive):
  
  #pragma acc directive [clauses]
OpenACC Parallel Construct

• For parallel execution on GPU in Fortran:
  
  ```fortran
  !$ACC PARALLEL [clauses]
  structured block
  !$ACC END PARALLEL
  ```

• In C/C++:

  ```c
  #pragma acc parallel [clauses]
  structured block
  ```

• Blocks of threads are created and one thread from each block begins executing the code in the structured block.
OpenACC Parallel Construct Clauses

• if
• async
• num_gangs
• num_workers
• vector_length
• copy, copyin, copyout, create
• present_or_copy, present_or_copyin, present_or_copyout, present_or_create
• private, first_private
OpenACC Kernels Construct

• For parallel execution on GPU in Fortran:
  !$ACC KERNELS [clauses]
  structured block
  !$ACC END KERNELS

• In C/C++:
  #pragma acc kernels [clauses]
  structured block

• Each loop iteration set will be executed by a distinct kernel;
• Each kernel may have a different number of blocks and threads.
OpenACC Kernels Construct Clauses

- if
- async
- num_gangs
- num_workers
- vector
- copy, copyin, copyout, create
- present
- preset_or_copy, present_or_copyin
- present_or_copyout, present_or_create
OpenACC Data Directive

• The data construct defines variables to be allocated on the device for a parallel region;
• Specifies whether copying should be done and in which direction;
• For Fortran:
  
  !$ACC DATA [clauses]

  structured block

  !$ACC END DATA

• For C/C++:
  
  #pragma acc data [clauses]

  structured block
OpenACC Data Directive Clauses

- if
- copy, copyin, copyout, create
- present
- preset_or_copy, present_or_copyin
- present_or_copyout, present_or_create
OpenACC Declare Directive

• Allows variables to be declared on the device for the duration of the data region;
• Clauses can specify if memory should be moved to and from host;
• In Fortran:
  
  !$ACC DELCARE [clauses]

• In C/C++:
  
  #pragma acc declare [clauses]
OpenACC Declare Directive Clauses

- copy, copyin, copyout, create
- present
- preset_or_copy, present_or_copyin
- present_or_copyout, present_or_create
OpenACC Directive Clauses (1)

• **if** - the compiler generates two versions of parallel region. Will execute accelerated version if condition is true and the host version if false;
• **async** - if present, code will continue to run host code. Can be labelled with an integer and used in conjunction with the `wait` directive;
• **copy** - variables reside on the host that need to be copied to the device and back to the host after calculations;
OpenACC Directive Clauses (2)

- `copyin` - data only needs to be copied from host to device;
- `copyout` - data only needs to be copied from device to host;
- `create` - data will be created only on device and no copying is done;
- `present` - data already resides on device and no copying is done;
- `present_or_create` - allocate variables if not already present on accelerator;
OpenACC Directive Clauses (3)

• `present_or_copy` - data is copied to device if not already present on device. Data is copied back from device to host;

• `present_or_copyin` - data is copied to device if not already present on device. Data is not copied back to host;

• `present_or_copyout` - if data is not present on accelerator, it is allocated. It is then copied back to host.
OpenACC Directive Clauses (4)

- `num_gangs` - number of blocks to use;
- `num_workers` - number of threads in each block;
- `vector_length` - length of vector for SIMD operations for each worker.
OpenACC Loop Construct

• The loop directive parallelises a loop with independent iterations;
• The scheduling is controlled by the type of loop and clauses - the best load balance;
• In Fortran:
  \[
  !$\text{ACC LOOP } [\text{clauses}] \\
  \text{do loop}
  \]
• In C/C++:
  \[
  \#pragma \text{acc loop } [\text{clauses}] \\
  \text{for loop}
  \]
OpenACC Combined Loops

• In Fortran:
  
  !$ACC PARALLEL LOOP [clauses]
  do loop
  !$ACC KERNELS LOOP [clauses]
  do loop

• In C/C++:
  
  #pragma acc parallel loop [clauses]
  for loop
  #pragma acc kernels loop [clauses]
  for loop
OpenACC Loop Construct Clauses (1)

- **collapse** - used to specify how many loops are controlled by the loop construct;
- **gang** - how many blocks are created to execute the loop by the parallel loop construct;
- **workers** - how many threads (in a block) should be created to execute loop iterations;
- **seq** - loop iterations are to be executed in sequential;
- **vector** - number of iterations that should be executed in vector mode;
OpenACC Loop Construct Clauses (2)

- `independent` - loop iterations are data independent;
- `private` - listed variables are private to each thread.
OpenACC Update Directive

- Used within a data region to update variables either on the host or device;
- In Fortran:
  
  !$ACC UPDATE [clauses]

- In C/C++:
  
  #pragma acc update [clauses]
OpenACC Update Directive Clauses

- host(list) - copy variables list from device to host;
- device(list) - copy variables list from host to device;
- if - execute if condition is true;
- async(label) - asynchronous update.
OpenACC Wait Directive

• The wait directive causes the program to wait until an asynchronous activity has completed;

• In Fortran:
  
  !$ACC WAIT [label]

• In C/C++:

  #pragma acc wait [label]
OpenACC Cache Directive

- Specifies that array elements should be fetched into the highest level of cache;
- Will increase performance, particularly where arrays are referenced in loops;
- In Fortran:
  \[ !$\text{ACC } \text{CACHE}(\text{list}) \]
- In C/C++:
  \[ \#\text{pragma acc cache(list)} \]
OpenACC Runtime Functions (1)

- `acc_get_num_devices(type)` - returns the number of devices of `type`, usually `acc_device_nvidia`;
- `acc_set_device_type(type)` - sets the device to use to `type`, usually `acc_device_nvidia`;
- `acc_get_device_type` - returns the device type that will be used;
- `acc_set_device_num(num, type)` - sets the device number to use. `num = 1, 2, ..., n;`
OpenACC Runtime Functions (2)

- **acc_get_device_num** - returns the device number that will be used;
- **acc_async_test(label)** - tests whether asynchronous activity `label` has completed;
- **acc_async_test_all** - tests whether all asynchronous activities have completed;
- **acc_async_wait(label)** - waits until asynchronous activity `label` has completed;
- **acc_async_wait_all** - waits until all asynchronous activities have completed;
OpenACC Runtime Functions (3)

- `acc_init(type)` - initialise device of type;  
- `acc_shutdown(type)` - shutdown device of type.
OpenACC Header Includes

• For Fortran include the file:
  include "openacc_lib.h"
or use pre-compiled header (or module):
  USE openacc

• For C/C++ include the file:
  #include <openacc.h>

• No need to specify the \texttt{-I} or the \texttt{-L} flag during compilation and linking;
• Just add the \texttt{-acc} flag.
OpenACC Environment Variables

• **ACC_DEVICE_TYPE** controls which device type to use during runtime:
  
  ```bash
  export ACC_DEVICE_TYPE=NVIDIA
  ```

• **ACC_DEVICE_NUM** controls which device number to use during runtime:
  
  ```bash
  export ACCDEVICE_NUM=1
  ```

• **PGI_ACC_TIME** displays run time statistics such as compute and data transfer time (in $\mu s$):
  
  ```bash
  export PGI_ACC_TIME=1
  ```

  *unset PGI_ACC_TIME* to disable.
Current PGI Limitations

• The PGI compiler version 12.5 currently does not support reduction and other operations;
• These features will be available in release 12.6;
• Feedback given by scientists to PGI is very helpful in developing a more mature product, so you can get involved;
• Status of future releases can be found at http://www.pgroup.com/resources/accel.htm#accrm
• PGI contact is at trs@pgroup.com
Nvidia Tesla C2050

• 515 GFLOP/s double precision or 1000 GFLOP/s single precision;
• 3 GB of ECC GDDR5 memory to ensure data integrity;
• 448 Tesla cores operating at 1.15 GHz (14 SMs);
• 144 GB/s of memory bandwidth;
• Grace has one node with two C2050 GPU cards;
• Use the following switches in your job script:
  
  #BSUB -R "rusage[gpushared=1]"
  #BSUB -q gpu
Conclusion

• CUDA provides an interface to GPU programming, but can be complicated to use;
• OpenACC provides a much simpler directive based interface to GPU programming;
• Although OpenACC is still in development, it is a viable solution to GPU programming;
• More functionality is being added to the PGI compiler and the standard is evolving;
• OpenACC is the future for accelerator based programming.
Practical Exercises

• Please use the PGI compiler:
  module load pgi/12.5

• To compile an OpenACC C program:
  pgcc -acc -Mpreprocess program.c \ 
  -o program

• To compile a Fortran program:
  pgf90 -acc -Mpreprocess program.f90 \ 
  -o program
Reference

1. *Programming Massively Parallel Processors*, D. Kirk and W. Hwu
2. *CUDA by Example*, J. Sanders and E. Kandrot
4. http://grace-head00.uea.ac.uk/grace-docs/OpenACC.1.0_0.pdf