COMP90025 Parallel and Multicore Computing

GPUs / CUDA

Lachlan Andrew

School of Computing and Information Systems
The University of Melbourne

2019 Semester II

GPU – A short background

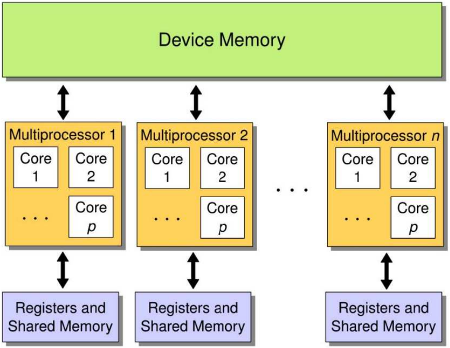
What is a GPU?

A Graphics processing unit (GPU) is a specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer intended for output to a display device.

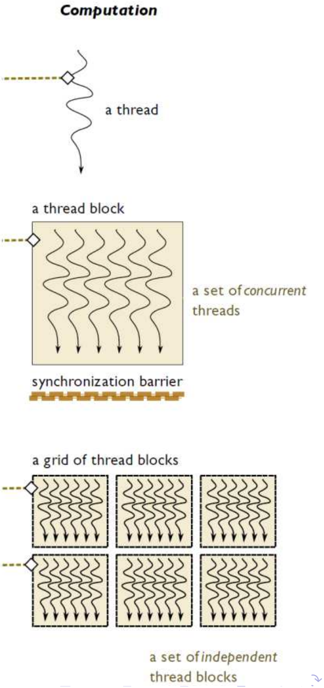
- GPUs can be foun in various computing devices:
  - Embedded systems, mobile phones, PCs, workstations, game consoles, and even supercomputers
  - A CPU can be present on a video card, or it can be embedded on the modeherboard or the CPU die

GPU components

- Hardware:
  - **SP**: streaming processor (AMD), aka CUDA core (Nvidia)
  - **SM**: streaming multiprocessor
- Software:
  - **Thread**: the smallest sequence of programmed instructions
  - **Block**: a bunch of threads that execute in a single SM and communicate through shared memory, aka *warp* (Nvidia)
  - **Grid**: a bunch of blocks that execute a kernel function



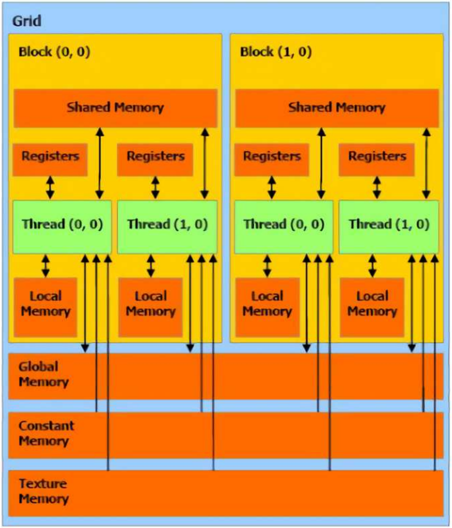
Execution hierarchy



- Thread: smallest execution entity
  - Each thread has its own ID
  - Thousands of threads executing the same program logic
- Threads are grouped into blocks
  - Threads in a block can synchronize execution
- Blocks are grouped in a grid
  - Blocks are independent (must be able to be executed in any order)

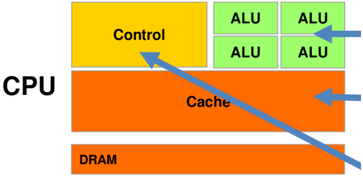
Memory hierarchy

- There are three types of memory
  - Global memory: 0.5 – 24 GB, with most now having ∼4 GB
  - Shared memory: ∼48 kB using hardware L1 cache
  - Registers and local memory: word width 32 bit
- Latency of memory access
  - Global memory: ∼300 ns
  - Shared memory: 5 ns
  - Registers and local memory: Fastest “memory”, about 10× faster than shared memory
- Purposes
  - Global memory: I/O for grid
  - Shared memory: thread collaboration within a block
  - Registers: store stack vars



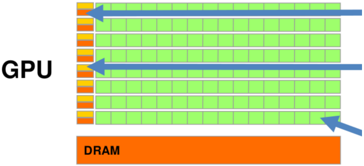
GPU vs CPU

CPU: Latency-oriented Design



- Powerful ALU: Reduced operation latency
- Large caches: Reduce memory latency
- Sophisticated control
  - branch prediction
  - data forwarding

GPU: Throughput-oriented design



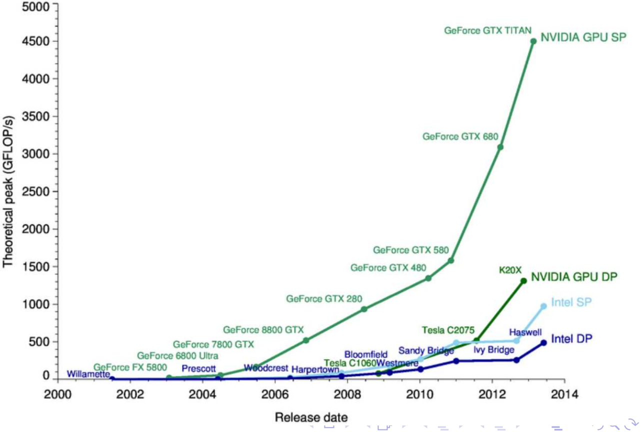
- Small caches boost memory throughput
- Simple control
- Energy efficient ALUs
  - Many
  - Long latency, but heavily pipelined
- Require massive number of threads to tolerate latencies

GPUs in parallel computing

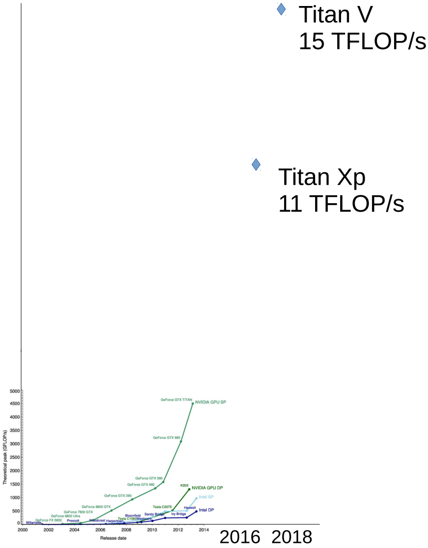
Advantages of using GPUs for parallel computation

The highly parallel structure maes them more efficient than general-purpose CPUs for algorithms where the processing of large blocks of data is done in parallel.

- Massively parallel
- Highly scalable
- Rapidly advancing



GPUs in parallel computing



GPUs in parallel computing

- GPU acceleration can yield impressive speed-up for some algorithms
- The speed-up ratio depends on the non-parallel part: **Amdahl's Law**
- **Significance**: an accelerated program on a GPU can be as fast as its serial part.
- (Remember Gustavson's Law: If the serial part doesn't grow as the problem size grows, then it becomes insignificant.)

Figure

CUDA – A short background

What is CUDA?

CUA is a parallel computing platform and application programming interface (API) model created by Nvidia. It allows software developers and softwre engineers to use a CUDA-enabled graphics processing unit (GPU) for general purpose processing.

- CUDA stands for **C**ompute **U**nified **D**evice **A**rchitecture
- That means it reveals a little bit about the GPU's architecture to allow parallel processing, but stays abstract enough that code is portable across Nvidia devices.
- CUDA is a compiler and toolkit for programming Nvidia GPUs
- The CUDA API extends C/C++, adds directives to translate them into instructions than run on the host CPU or GPU when needed
- CUDA allows for easy multi-threading – parallel executing on all streaming processors on the GPU

Two sides of CUDA

Advantages

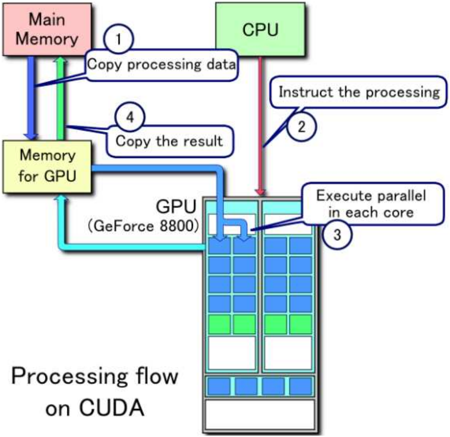
- Abstraction from the hardware
  - Programmers don't see every little aspect of the machine
  - Gives flexibility to the vendor to update hardware but keep legacy code forward compatible
- Automatic thread management
  - Multi-threading: hides latency and helps maximizing the GPU utilization
  - Transparent for the programmer
  - Limited synchronization between threads is provided
  - Avoid dead-lock (no message passing)

Disadvantages

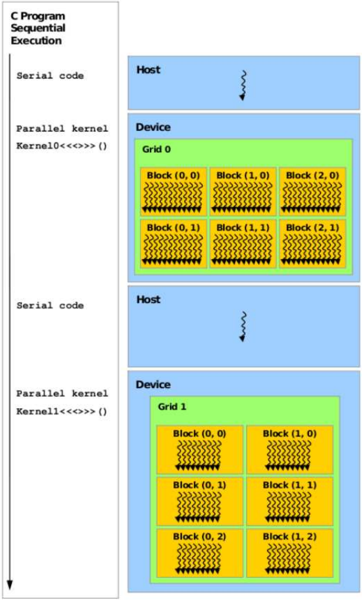
- Vendor-lock to Nvidia (Alternative architecture-independent schemes – e.g., OpenCL – require more programming effort)
- Still difficult to program; impossible to accelerate chaotic code flow
- Hard to debug (not even printf in early versions)

Example of CUDA processing flow

1. Copy data from main memory to GPU memory
2. CPU initiates the GPU compute kernel
3. GPU's CUDA cores execute the kernel in parallel
4. Copy the resulting data from GPU memory to main memory



Example of program execution flow



Serial code executes on the host while parallel code executes on the device.

1. Host executes serial code
2. Device executes parallel kernel 0
3. Host executes serial code
4. Device executes parallel kernel 1

Programming in CUDA

Basic C extensions

- Function modifiers: programmer can define where a function should run
  - __host__ : to be called and executed by the host CPU
  - __device__ : to run on the GPU, and the function can only be called by code running on the GPU
  - __global__ : to run on the GPU but called from the host. This is the access point to start multi-threaded code on the GPU
- Variable modifiers
  - _device_ : the variable resides in the GPU's global memory and is defined while the code runs
  - _shared_ : variable in shared memory, with the same lifespan as the block.
- _syncthreads(): sync of threads within a block

writing a __global__ function

- All calls to a global function must specify how many threaded copies to launch and in what configuration
- CUDA syntax: $<<< \cdots >>>$
  - Inside the $<<<>>>$, we need at least two arguments (can be more to overwrite default values)
  - Call example: *my_func* $<<<$*bg, tb*$>>>$(*arg1*,*arg2*)
  - *bg* specifies the dimensions of the block grid
  - *tb* specifies the dimensions of each thread block
  - *bg* and *tb* are both of type *dim3* (a new data type defined by CUDA: three unsigned ints where any unspecified component defaults to 1)
  - *dim3* has strick-like access: members are x, y and z
  - 1-D syntax allowed: *myfunc*$<<<$*5, 6*$>>>$() makes 5 blocks in a linear array, with 6 threads each, and runs *myfunc* on them all.

Allowing the CUDA kernel to get data

- Allocate CPU memory for n integers, e.g., malloc (...)
- Allocate GPU memory for n integers, e.g., cudaMalloc(...)
- Copy the CPU memory to GPU memory for n integers, e.g., cudaMemcpy(..., cudaMemcpyHostToDevice)
- Copy the GPU memory to CPU once computation is done, e.g., cudaMemcpy(..., cudaMemcpyDeviceToHost)
- Free the GPU and CPU memory, e.g., cudaFree(...)

Example: Vector adder

Simple example: add two arrays

```
// Device code
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}
```

Example: Vector adder

1. Memory allocation
2. Memory copy: Host → GPU
3. Kernel call
4. Memory copy: GPU → Host
5. Free GPU memory

```
// Host code
int main ()
{
    // Allocate vectors in device memory
    size_t size = N * sizeof(float);
    float* d_A;
    cudaMalloc((void**)&d_A, size);
    float* d_B;
    cudaMalloc((void**)&d_B, size);
    float* d_C;
    cudaMalloc((void**)&d_C, size);

    // Copy vectors from host memory to device memory
    // h_A and h_B are input vectors stored in host memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Invoke kernel
    int threadsPerBlock = 256;
    int threadsPerGrid =
            (N + threadsPerBlock - 1) / threadsPerBlock;
    VecAdd<<<threadsPerGrid, threadsPerBlock>>>(d_A, d_B, d_C);

    // Copy result from device memory to host memory
    // h_C contains the resut in host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
}
```

Example: Block Cipher

```
__global__ void shift_cypher (
    unsigned int * input_array, unsigned int * output_array,
    unsigned int shift_amount, unsigned int alphabet_max,
    unsigned int array_length)
{
    unsigned int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int shifted = input_array [tid] + shift_amount ;
    if (shifted > alphabet_max)
        shifted = shifted % (alphabet_max + 1);
    output_array [tid] = shifted ;
}
```

Example: Block Cipher

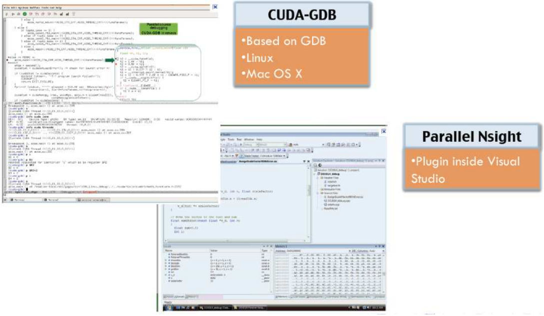
```
# include < stdio .h >
int main () {
    unsigned int num_bytes = sizeof (int) * (1 << 22);
    unsigned int * input_array = 0;
    unsigned int * output_array = 0;
    ...
    cudaMalloc ((void**)&input_array, num_bytes);
    cudaMalloc ((void**)&output_array, num_bytes);
    cudaMemcpy (input_array, host_input_array, num_bytes, cudaMemcpyHostToDevice);
    dim3 dimGrid (ceil (array_length)/ block_size);
    dim3 dimBlock (block_size);
    // gpu will compute the kernel and transfer the results
    // out of the gpu to host .
    shift_cypher<<<dimGrid, dimBlock>>>(input_array ,
output_array, shift_amount, alphabet_max,
array_length);
    cudaMemcpy (host_output_array, output_array, num_bytes,
    cudaMemcpyDeviceToHost);
    ...
    // free the memory
    cudaFree (input_array);
    cudaFree (output_array);
}
```

Compiling CUDA program

- CUDA code must be compiled using *nvcc*
- *nvcc* generates both instructions for CPU and GPU (PTX instruction set), as well as instructions to send data back and forwards between them

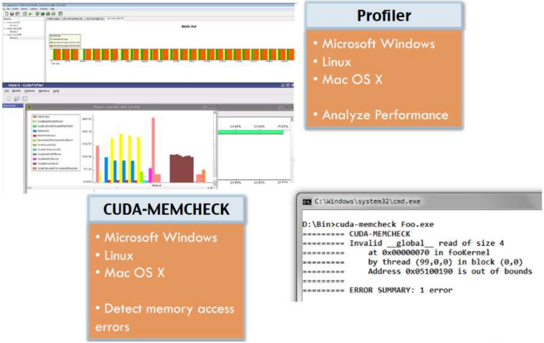
Debugging a CUDA program

- Various tools are available in the market for simultaneous CPU and GPU debugging
  - Set breakpoints and conditional breakpoints
  - Dump stack frames for thousands of CUDA threads
  - Inspect memory, registers, local/shared/global variables
- Runtime error detection
  - Supports multipel GPUs, multiple contexts, multiple kernels



Profiling a CUDA program

- The *Visual Profiler* is a graphical profiling tool that displays a timeline of your application's CPU and GPU activity
- *Profiler* also includes an automated analysis engine to identify optimization opportunities
- The *nvprof* profiling tool enables you to collect and view profiling data from the command line



An architecture-independent alternative — OpenCL

- CUDA and Intel's MIC interface were written by companies who want to encourage people to use their software
  - Ease of use is vital
  - Exploiting this specific architecture to the fullest is desired
  - Portability of code to other vendors is actually a disadvantage for the framework designers
- OpenCL is a framework for writing portable many-core code
  - Code should run on GPUs, Xeon Phi, CPU, DSPs, FPGAs
  - For maximum portabiity, kernel is compiled at runtime
    - Can part-compile to intermediate representation SPIR-V at compile time

A “Hello world” program in openCL

```
#define CL_USE_DEPRECATED_OPENCL_2_0_APIS

#include<CL/cl.hpp>
#include<iostream>
#include <fstream>

int main()
{
    std::vector<cl::Platform> platforms;
    cl::Platform::get(&platforms);

    auto platform = platforms.front();
    std::vector<cl::Device> devices;
    platform.getDevices(CL_DEVICE_TYPE_CPU, &devices);

    auto device = devices.front();
```

A "Hello world" program in openCL

```
std::ifstream helloWorldFile("hello.cl");
std::string src(std::istreambuf_iterator<char>(helloWorldFile),
                (std::istreambuf_iterator<char>()));

cl::Program::Sources sources(1, std::make_pair(src.c_str(),
                             src.length() + 1));

cl::Context context(device);
cl::Program program(context, sources);

auto err = program.build("-cl-std=CL1.2");

char buf[16];
cl::Buffer memBuf(context,
                  CL_MEM_WRITE_ONLY | CL_MEM_HOST_READ_ONLY,
  sizeof(buf));
cl::Kernel kernel(program, "HelloWorld", &err);
kernel.setArg(0, memBuf);
```

A "Hello world" program in openCL

```
    cl::CommandQueue queue(context, device);
    queue.enqueueTask(kernel);
    queue.enqueueReadBuffer(memBuf, GL_TRUE, 0, sizeof(buf), buf);

    std::cout << buf;
}
```

OpenCL "Hello world" program

```
__kernel void HelloWorld(__global char* data)
{
    data[0] = 'H';
    data[1] = 'E';
    data[2] = 'L';
    data[3] = 'L';
    data[4] = 'O';
    data[5] = ' ';
    data[6] = 'W';
    data[7] = 'O';
    data[8] = 'R';
    data[9] = 'L';
    data[10] = 'D';
    data[11] = '!';
    data[12] = '\n';
}
```

Reference

http://lorenabarba.com/gpuatbu/Program_files/
Cruz_gpuComputing09.pdf
https://www.slideshare.net/piyushmittalin/a-beginners-guide-t
https://www.slideshare.net/RaymondTay1/introduction-to-cudas
https://www.khronos.org/opencl