Level 7 Advanced Computational Physics – Lecture 10

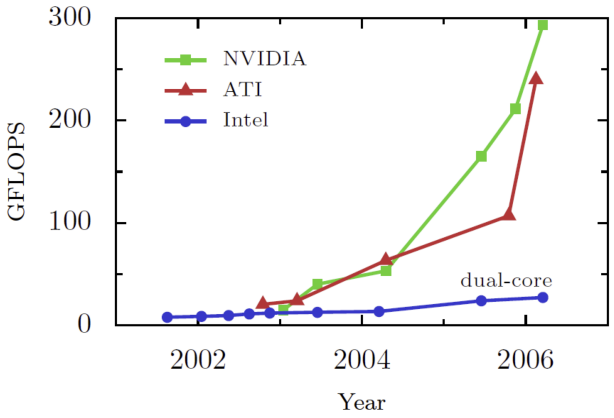
Dr. Simon Hanna

November 9, 2021

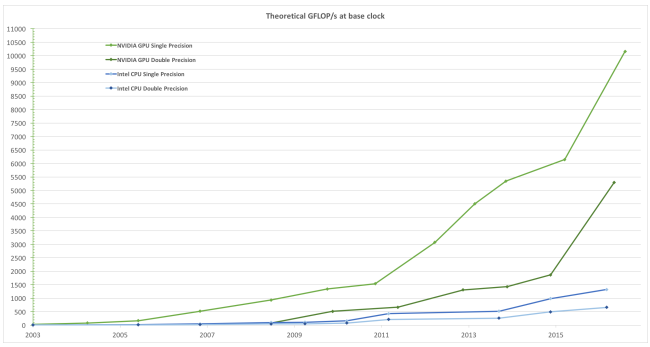
13 Graphics Programming

13.1 Introduction

- In recent years, single-core CPU performance has essentially stagnated and no longer follows Moore's law
- 10 years ago, the solution was multicore and increased parallelism
- Ten years ago, GPU floating point performance easily outstripped CPU[1]



- Graphics processing is inherently parallel
- But GPUs were built to do graphics processing only, with little to no programmability
- In 2006, NVIDIA released "CUDA" (Compute Unified Device Architecture) language to allow non-graphics programs to run on GPUs
- AMD GPUs can use CUDA but generally programmed using Open-CL – a standard that grew out of CUDA.
- Current GPU performance[2]:



- GPUs now widely deployed as accelerators
- Kepler K40 GPUs from NVIDIA have performance of 4Tflops (peak)
  - CM-5, #1 system in 1993 was ~60 Gflops
  - ASCI White (#1 in 2001) was 4.9 Tflops
  - BlueCrystal Phase 3 has ~80 of these
- The GPU is **not** a transparent accelerator – code must be rewritten to target GPUs

13.2 GPU hardware

- Three key ideas:
  1. Use many "slimmed down cores" in parallel
  2. Pack cores full of arithmetic and logic units (ALUs) and make use of:
     (a) Explicit SIMD vector instructions
     (b) Implicit sharing managed by hardware
  3. Avoid latency stalls by interleaving execution of many groups of threads
     - When one group stalls, work on another
     - Rapid thread switching leads to overall performance gains

13.2.1 Practical example: NVIDIA GeForce GTX 580

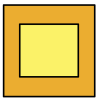
- The basic compute unit is the "Streaming Multiprocessor" (SM) which comprises multiple CUDA

[1] Owens et al., A Survey of General-Purpose Computation on Graphics Hardware

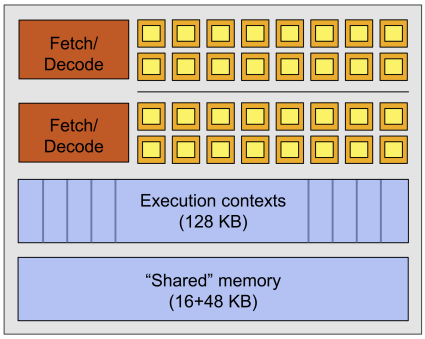
[2] NVIDIA, CUDA C Programming Guide

cores

- A CUDA core is a SIMD functional unit:

 = CUDA core (1 multiply-add operation per clock)

- The SM contains 32 CUDA cores arranged as 2 groups of 16 SIMD functional units

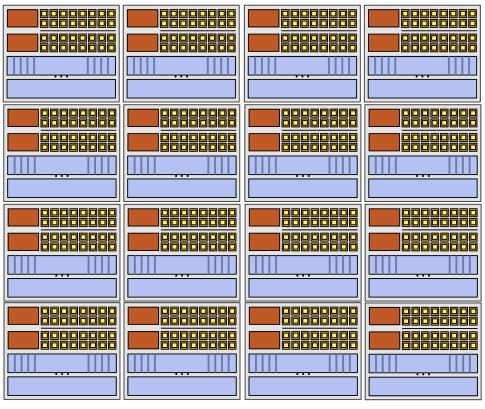


- Groups of 32 CUDA threads (called a warp) share an instruction stream on the SM

- Two warps are selected each clock (decode, fetch, and execute two warps in parallel)

- Up to 48 warps are interleaved, allowing 1536 CUDA threads to be stored simultaneously **on each SM**.

- GTX580 has 16 SMs, so 512 CUDA cores total:



- That means 24,576 CUDA threads can be handled simultaneously.

- More modern GPU cards (e.g. GTX1080Ti, RTX2080) have up to around 4,500 CUDA cores and handle correspondingly more threads (c. 200,000).

13.2.2 Pascal:

- Recent generation of GPU (2016) from NVIDIA known as Pascal P100 (e.g. GeForce 1080)
- Has 56 SM and 3584 SP or 1792 DP CUDA cores
- Achieves 5.3 Tflops DP and 10.6 Tflops SP
- BlueCrystal Phase 4 has 64 of them
- Each has 12GB of on-board memory

13.2.3 More on threads:

- Hierarchical Threading
  - Grid → Thread Blocks → Warps → Threads
  - OpenMP generally uses flat threading model, though different thread levels are possible
- Only threads within same thread block can communicate and synchronize with each other
- Maximum 1024 threads per thread block – depends on GPU generation
- Thread block is divided into mutually exclusive, equally-sized group of threads called *warps*
  - Warp size is hardware-dependent
  - Usually 32 threads
- AMD have a similar language for their GPUs, but a warp is known as a wavefront.

13.3 GPU programming models

13.3.1 Using a GPU:

1. You must retarget code for the GPU
2. The working data must fit in GPU RAM
3. You must copy data to/from GPU RAM
4. You must copy a "kernel" to the GPU
5. Lots of parallelism preferred (throughput, not latency)
6. SIMD-style parallelism best suited
7. High arithmetic intensity (FLOPs/byte) preferred

13.3.2 CUDA:

- "Compute Unified Device Architecture"
- First language to allow general-purpose programming for GPUs
  - preceded by "shader" graphics languages
- Promoted by NVIDIA for their GPUs
- Not supported by other accelerators

13.3.3 OpenCL:

- C99-based dialect for programming heterogenous systems
- Originally based on CUDA
- Supported by more than GPUs (Xeon Phi, FPGAs, CPUs, etc.)
- Source code is portable (somewhat), though performance may not be
- Poorly supported by NVIDIA

13.3.4 "Drop-in" libraries:

- "Drop-in" replacements for popular CPU libraries; examples from NVIDIA:
  - CUBLAS/NVBLAS for BLAS (e.g. ATLAS)
  - CUFFT for FFTW
  - MAGMA for LAPACK and BLAS
- These libraries may still expect you to manage data transfers manually
- Libraries may support multiple accelerators

13.4 A word of caution...

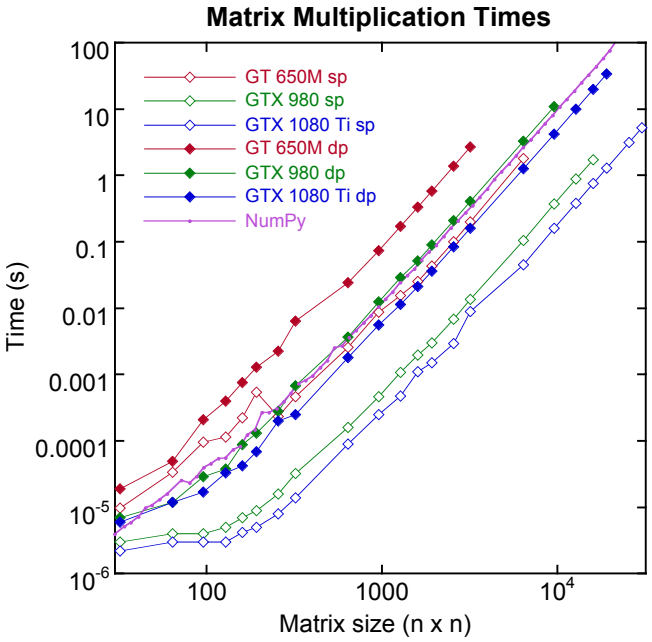
- People sometimes make outlandish claims for the speed-ups that are available by using GPUs – Speed-ups of 100x to 1000x are often quoted.
- The following figures are taken from "The CUDA Handbook" by Nicholas Wilt (2013), based on an n-body gravity simulation:

Implementation	Body-Body interactions per second (billions)	Speed-up relative to basic CPU	Speed-up relative to optimized CPU
Scalar CPU	0.017	1x	
Vectorised CPU	0.307	17.8x	
Vectorised and multithreaded CPU	5.650	332x	1x
Naive GPU	25.0	1471x	4.4x
Optimised GPU	45.2	2659x	8x

- Note: above comparisons between a GK104 GPU (1152 CUDA cores, 823MHz, mid 2012 model) and twin Xeon E2670's with 16 real cores (plus hyperthreading x2, 2.6GHz).
- The speed-up is clearly still worth having but a well-vectorised CPU code can beat a poorly written GPU code.
- In the Gromacs molecular dynamics example given previously, running on a 6th generation 4 core i7, the nVidia GTX980 gpu accelerated the code by 4x in single precision
- GPUs offer a good price / FLOP but remain niche e.g. BlueCrystal P4 has 15000 cores but only 64 GPUs; P3 has 80 GPUs; P5 is likely to have more
- The timings above are all for single precision code. Until the current generation of GPUs (nVidia Pascal and AMD FirePro), only a small fraction of the GPU was capable of double precision calculations (typically $\sim$10% of the cores). On the latest cards, the DP speed is 0.5xSP speed, but **only** on the server-grade cards.
- In fact, sometimes, the algorithm cannot succeed on the GPU e.g. my PiCalc program where the timings are dominated by a single "sqrt()" call, and the method requires double precision variables:

Hardware	PiCalc Time(s)
2012 I7 Scalar	3.9
2012 I7 Vectorised	1.95
2012 I7 Vectorised and multithreaded	0.512
2016 I7 Scalar	3.0
2016 I7 Vectorised	0.73
2016 I7 Vectorised and multithreaded	0.180
2012 nVidia GTX650M	3.32
2015 nVidia GTX980	0.321
2016 nVidia GTX1080	0.18

- A comparison of single and double precision times for matrix multiplication:



13.5 CUDA programming model

13.5.1 Introduction

- CUDA language: C++ dialect
  - Host code (CPU) and GPU code in same file
  - Special language extensions for GPU code
- CUDA Runtime API
  - Manages runtime GPU environment
  - Allocation of memory, data transfers, synchronization with GPU, etc.
  - Usually invoked by host code
- CUDA Device API
  - Lower-level API that CUDA Runtime API is built upon

13.5.2 Simple example, vector addition:

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

- Kernel defined using `__global__`
- `threadIdx` provides unique ID for each thread
- `<<<NumBlocks, ThreadsPerBlock>>>` notation to invoke Kernel
- In practice it is necessary to:
  1. allocate the arrays A, B, C in the cpu memory space;
  2. allocate additional arrays g_A, g_B, g_C in the gpu memory space, using “cuda_Malloc()”;
  3. transfer A, B to g_A, g_B using “cudaMemcpy()”;
  4. pass pointers to g_A, g_B, g_C into the kernel function, so it knows which data to work on;
  5. finally transfer g_C to C using “cudaMemcpy()” again.
- The whole program might appear:

```
#include <cuda.h>
#include <stdio.h>

__global__ void
VecAdd( float *A, float *B, float *C, int numElements){
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < numElements){
        C[i] = A[i] + B[i];
    }
}

int main(void){
    // Vector length to be used, and its size
    int numElements = 50000;
    size_t size = numElements * sizeof(float);

    // Allocate the host arrays
    float *A = (float *)malloc(size);
    float *B = (float *)malloc(size);
    float *C = (float *)malloc(size);

    // Initialize the host arrays
    for (int i = 0; i < numElements; ++i){
        A[i] = rand()/(float)RAND_MAX;
        B[i] = rand()/(float)RAND_MAX;
    }

    // Allocate the device arrays
    float *g_A = NULL, *g_B = NULL, *g_C = NULL;
    cudaMalloc((void **)&g_A, size);
    cudaMalloc((void **)&g_B, size);
    cudaMalloc((void **)&g_C, size);
```

```
    // Copy the host input vectors A and B in host memory
    // to the device input vectors in device memory
    cudaMemcpy(g_A, A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(g_B, B, size, cudaMemcpyHostToDevice);


    // Launch the Vector Add CUDA Kernel
    int threadsPerBlock = 256;
    int blocksPerGrid =
        (numElements + threadsPerBlock - 1) / threadsPerBlock;
    VecAdd<<<blocksPerGrid, threadsPerBlock>>>
        (g_A, g_B, g_C, numElements);

    // Copy the device result vector in device memory
    // to the host result vector in host memory.
    cudaMemcpy(C, g_C, size, cudaMemcpyDeviceToHost);



    // Verify that the result vector is correct
    for (int i = 0; i < numElements; ++i){
        if (fabs(A[i] + B[i] - C[i]) > 1e-5){
            fprintf(stderr, "FAIL at element %d!\n", i);
            exit(EXIT_FAILURE);
        }
    }

    printf("Test PASSED\n");

    // Free device global memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    // Free host memory
    free(h_A);
    free(h_B);
    free(h_C);

    return 0;
}
```

13.5.3 Example, calculation of π:

This is an example of summing a series to evaluate pi. The method is fast, avoiding use of square roots.

```
#include <cuda.h>
#include <stdio.h>
__global__
void pi_kernel(int N, double *pi_value) {
  int tid = blockIdx.x * blockDim.x + threadIdx.x;
  int nthreads = blockDim.x * gridDim.x;
  double pi = 0;
  for(int i = tid; i < N; i+=nthreads) {
    double t = ((i + 0.5) / N);
    pi += 4.0/(1.0+t*t);
  }
  // Store pi value in GPU memory
  pi_value[tid] = pi;
}

int main(void) {
  int NTHREADS = 2048;
  int N = 10485760;
  double *c_pi, *g_pi;
  size_t g_pi_size = NTHREADS * sizeof(double);

  // Allocate memory for CPU
  c_pi = (double *) calloc(NTHREADS, sizeof(double));

  // Allocate memory on GPU
  if(cudaMalloc(&g_pi, g_pi_size) != cudaSuccess) {
    fprintf(stderr, "failed to allocate memory!\n");
    exit(1);
  }

  // Call GPU Kernel
  pi_kernel<<<NTHREADS/256, 256>>>(N, g_pi);

  // Copy back pi values
  if(cudaMemcpy(c_pi, g_pi, g_pi_size,
      cudaMemcpyDeviceToHost) != cudaSuccess) {
    fprintf(stderr,
      "failed to copy data back to CPU!\n");
    exit(1);
  }

  // Reduce pi_value on CPU
  double pi = 0;
  for(int i = 0; i < NTHREADS; i++)
    pi += c_pi[i];
  printf("pi=%16.15f\n", pi/N);
  return 0;
}
```

13.5.4 Python (using pycuda – pyopencl also available):

- Python offers a simplified interface, hiding much of the setup required from the programmer.
- Ultimately the full speed of the GPU is available.
- Need to install development version of CUDA, available from nVidia.
- Also need to install a copy of pycuda in your Python distribution – this is a little tricky (essentially impossible now on a Mac), but some instructions are available here: https://mathema.tician.de/software/pycuda/

```
import pycuda.autoinit
import pycuda.driver as drv
import numpy
from pycuda.compiler import SourceModule

mod = SourceModule("""
__global__ void multiply_them(float *dest,
      float *a, float *b) {
  const int i = threadIdx.x;
  dest[i] = a[i] * b[i];
}
""")
```

```
multiply_them = mod.get_function("multiply_them")

a = numpy.random.randn(400).astype(numpy.float32)
b = numpy.random.randn(400).astype(numpy.float32)

dest = numpy.zeros_like(a)

multiply_them(
        drv.Out(dest), drv.In(a), drv.In(b),
        block=(400,1,1), grid=(1,1))
print (dest-a*b)
```

- Note CUDA kernel in C
- "block=(400,1,1)" selects 400 threads
- "drv.In / Out" handle automatic transfers
- Pi calculation revisited:

```
import pycuda.driver as cuda
import pycuda.autoinit
import numpy as np
from pycuda.compiler import SourceModule

mod = SourceModule("""
   __global__ void pi_kernel(int N, double *pi_value) {
  int tid = blockIdx.x * blockDim.x + threadIdx.x;
  int nthreads = blockDim.x * gridDim.x;
  double pi = 0;
  for(int i = tid; i < N; i+=nthreads) {
    double t = ((i + 0.5) / N);
    pi += 4.0/(1.0+t*t);
  }
  // Store pi value in GPU memory
  pi_value[tid] = pi;
      }
   """,nvcc="/usr/local/cuda/bin/nvcc")

call_pi_kernel = mod.get_function("pi_kernel")

NTHREADS = 2048
N = 10485760

# Allocate memory for CPU
c_pi = np.zeros(NTHREADS)
c_pi = c_pi.astype(np.float64)

# Allocate memory on GPU
g_pi = cuda.mem_alloc(c_pi.nbytes)

# Call GPU Kernel
call_pi_kernel(np.int32(N), g_pi,
    block=(256,1,1), grid=(8,1))

# Copy back pi values
cuda.memcpy_dtoh(c_pi, g_pi)

# Reduce pi_value on CPU
pi = c_pi.sum()
print("pi = %17.15f" % (pi/N))
```

13.5.5 Memory spaces within CUDA

Within the CUDA device there are several distinct types of memory with specific access rules:

Global memory: Accessed via device pointers in CUDA kernels. Accessible from any core.

Constant memory: Read-only memory optimized for broadcast to multiple threads. Should not be changed by individual kernels.

Local memory: This contains variables needed by each kernel, including local variables that cannot be held in registers, parameters, and return addresses for subroutines.

Texture memory: Used by the image output hardware of the gpu, texture memory allows access to a range of useful array-based operations. Can be useful in some image processing applications.

Shared memory: May be used to exchange data between CUDA threads within a block. 10x slower than registers, but 10x faster than global memory.

13.6 OpenCL programming

13.6.1 Introduction

- OpenCL grew out of CUDA and was pushed by vendors, including Apple, as a way to get the most out of the available computing resources. (Apple has now deprecated OpenCL in favour of Metal, which only works on Apple hardware. CUDA is incompatible with Apple OS from 10.14 Mojave onwards).
- OpenCL is meant to be "all things to all processors".
- Basic paradigm similar to CUDA i.e.
  - Kernel runs on GPU cores
  - Host transfers data to/from GPU and handles bookkeeping
- OpenCL framework consists of:

OpenCL platform API: The platform API defines functions used by the host program to discover

OpenCL devices and their capabilities as well as to create the context for the OpenCL application.

OpenCL runtime API: This API manipulates the context to create command-queues and other operations that occur at runtime. For example, the functions to submit commands to the command-queue come from the OpenCL runtime API.

The OpenCL programming language: This is the programming language used to write the code for kernels. It is based on an extended subset of the ISO C99 standard and hence is often referred to as the OpenCL C programming language.

- The main() function either implements or calls functions that perform the following operations:
  - Create an OpenCL context on the first available platform.
  - Create a command-queue on the first available device.
  - Load a kernel file (HelloWorld.cl) and build it into a program object.
  - Create a kernel object for the kernel function hello_kernel().
  - Create memory objects for the arguments to the kernel (result, a, b).
  - Queue the kernel for execution.
  - Read the results of the kernel back into the result buffer.
- Example OpenCL kernel – very similar to a CUDA kernel:

```
__kernel void hello_kernel(
                    __global const float *a,
                    __global const float *b,
                    __global float *result)
{
    int gid = get_global_id(0);
    result[gid] = a[gid] + b[gid];
}
```

13.6.2 PyOpenCL – a friendly approach to OpenCL

- From the same authors as PyCuda, see: https://documen.tician.de/pyopencl/
- Seeks to remove the complexity of OpenCL, so the programmer can focus on important aspects
- Example program:

```
import numpy as np
import pyopencl as cl

a_np = np.random.rand(50000).astype(np.float32)
b_np = np.random.rand(50000).astype(np.float32)

ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)

mf = cl.mem_flags
a_g = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR,
                                        hostbuf=a_np)
b_g = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR,
                                        hostbuf=b_np)

prg = cl.Program(ctx, """
__kernel void sum(
    __global const float *a_g, __global const float *b_g,
                                __global float *res_g)
{
  int gid = get_global_id(0);
  res_g[gid] = a_g[gid] + b_g[gid];
}
""").build()

res_g = cl.Buffer(ctx, mf.WRITE_ONLY, a_np.nbytes)
prg.sum(queue, a_np.shape, None, a_g, b_g, res_g)

res_np = np.empty_like(a_np)
cl.enqueue_copy(queue, res_np, res_g)

# Check on CPU with Numpy:
print(res_np - (a_np + b_np))
print(np.linalg.norm(res_np - (a_np + b_np)))
```