GPU Programming

CCIT CITI

Ashwin Srinath

Clemson University

Frequently Asked Question

How much speedup can I get from using GPUs?

A: 0.001-100x

GPUs: what are they?

- **Computing devices**: traditionally good for graphics calculations. Over the last decade or so have been used for general-purpose computing with much success
- **Massively parallel processors**: 1000s of small, "weak" cores as opposed to handful of fast, powerful cores (CPU)
- **Accelerators:**, not computers: memory space different from CPU. Applications typically primarily run on the CPU, but offload compute intensive parts to the GPU
- **Application areas**: AI/Machine Learning, Numerical Simulations (CFD, molecular dynamics, weather sciences, etc.,), Imaging and Computer Vision, Bioinformatics, Data Science, etc.,

Using GPUs

Three ways to accelerate applications using GPUs

- **1. GPU Programming**
  - Programming models: CUDA, OpenCL - we'll focus on CUDA
  - **Programming languages**: C, C++, Fortran - we'll use C today
  - Other languages like Python, MATLAB and R can interface with CUDA - we'll see an example of this with Python
  - Most performance and flexibility, but requires the most effort

Using GPUs

Three ways to accelerate applications using GPUs

- **2. CUDA-enabled libraries**
  - "Drop-in" GPU acceleration requiring small amount of code changes
  - Libraries available for Machine Learning, Linear Algebra, Parallel Algorithms, Signal Processing, etc.,
  - Less effort than programming "from scratch", library functions are generally well-tested and performant

Using GPUs

Three ways to accelerate applications using GPUs

- **3. Compiler directives**
  - Least effort, requires minimal changes to code
  - Compiler handles details of parallelism management and data movement to/from GPU
  - Uncertain performance

GPU Hardware

- NVIDIA offers special GPUs targeting HPC/scientific workloads (**Tesla**)
  - more resources on the GPU are dedicated to floating-point operations
- Example spec: **NVIDIA Tesla P100 GPU**:
  - Number of CUDA cores: $\sim$3500
  - On-chip memory: 16 GB
  - Double-precision performance: 4.7 Teraflops
  - Single-precision performance: 9.3 Teraflops
  - Memory bandwidth: 732 GB/s

Key to good performance on GPUs

- Expose as much parallelism as possible
- Avoid copies from host to device and vice-versa
- Leverage the GPU memory hierarchy
- Leverage GPU libraries

GPUs on the Palmetto Cluster

```
$ qsub -I -l select=1:ncpus=2:mem=20gb:ngpus=1,\
walltime=10:00:00 -q R2387430
```

Keep in mind the following:

- The `ngpus=` option must be specified
- Ask for *at least 2 cores*. For single core jobs, the rest of the PBS resource limits specification is ignored.
- The special queue `R2387430` is valid only for today. After today, you will not need to specify this option.
- You can also specify `gpu_model=k20|k40|p100` as part of your resource limits specification. But the workshop queue has only `k40` GPUs.

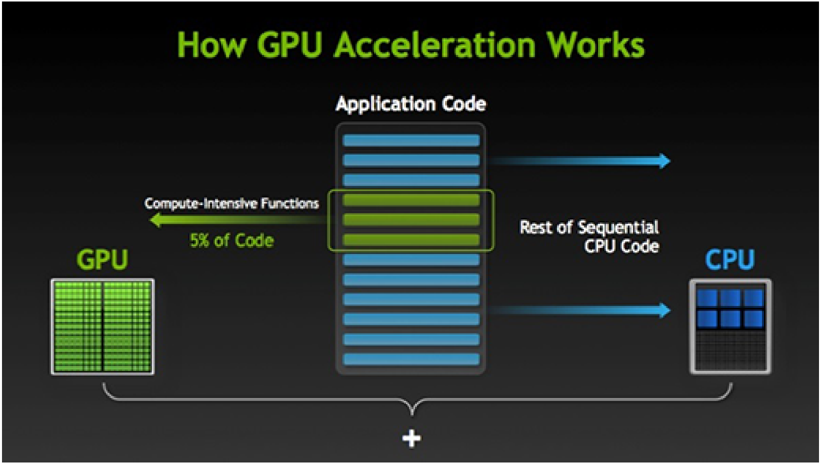

Figure 1: How GPU acceleration works

CUDA Hello World

- CUDA C programs look like “normal” C programs, but include calls to special functions that execute on the GPU, called **kernels**
- Kernels are executed in parallel by several GPU threads. Kernels are defined using the `__global__` specifier as shown below:

```
__global__ void helloKernel() {
    printf("Hello from the GPU!\n");
}
```

- Calls to a CUDA kernel should specify the number of threads that will execute the kernel:

```
helloKernel <<<1, 64>>> (); // Execute on 64 GPU threads
```

Memory management in CUDA

GPU (*device*) and CPU (*host*) have different memory spaces and are allocated and managed differently. CUDA provides `cudaMalloc` and `cuFree` for this:

```
int *a
int *d_a

/* allocating memory on host: */
a = (int *)malloc( size );

/* allocating memory on device: */
cudaMalloc( (void **) &d_a, size );

/* freeing memory on host: */
free(a);

/* freeing memory on device: */
cudFree(d_a);
```

Memory management in CUDA

Data needs to be explicitly copied between CPU and GPU:

```
/* copy data from host to device: */
cudaMemcpy( d_a, a, size, cudaMemcpyHostToDevice );

/* do the work on the device */

/* copy data back from device to host: */
cudaMemcpy( a, d_a, size, cudaMemcpyDeviceToHost );
```

Summing vectors: CPU v/s GPU

CPU: Loop from 1 to N:

```
void sum ( *a, *b, *c, N )
{
  for ( int i=0; i<N; i++ )
  {
    c[i] = a[i] + b[i];
  }
}
```

GPU: No loop, launched with N threads:

```
__global__ void sumKernel( *d_a, *d_b, *d_c )
{
  int i = threadIdx.x;
  c[i] = a[i] + b[i];
}
```

```
sumKernel <<<1, N>>> (d_a, d_b, d_c);
```

Heat conduction with a point source

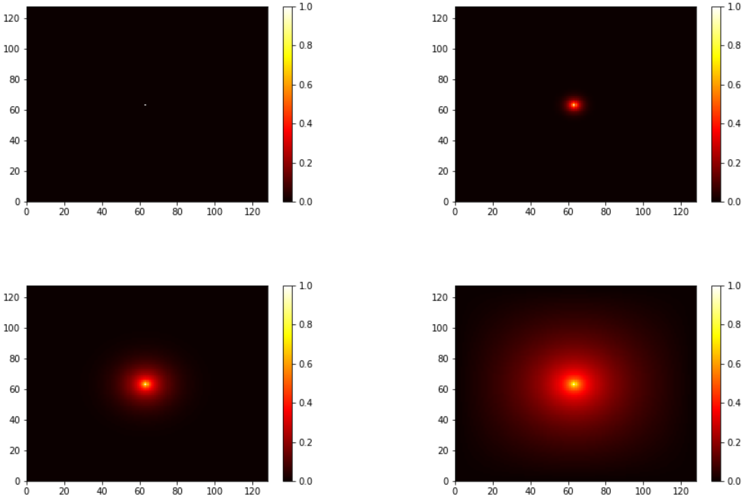

Figure 2: Temperature distribution at 0, 5, 50, and 500 steps

Heat conduction with a point source: algorithm

```
void heat_conduction_step(double *T1, double *T2) {
    for (int i = 1; i < N - 1; i++) {
        for (int j = 1; j < N - 1; j++) {
            T1[i*N + j] = (
                    T2[(i-1)*N + j] +
                    T2[(i+1)*N + j] +
                    T2[i*N + (j-1)] +
                    T2[i*N + (j+1)]) / 4.0;
        }
    }
    if (i == Ny / 2 && j == Nx / 2) {
        T1[i, j] = 1.0;
}

temp = T1; T1 = T2; T2 = temp;
```

Matrix multiplication

Matrix multiplication