Introduction to GPUs

Mark Greenstreet and Ian M. Mitchell

CpSc 418 – February 14, 2018

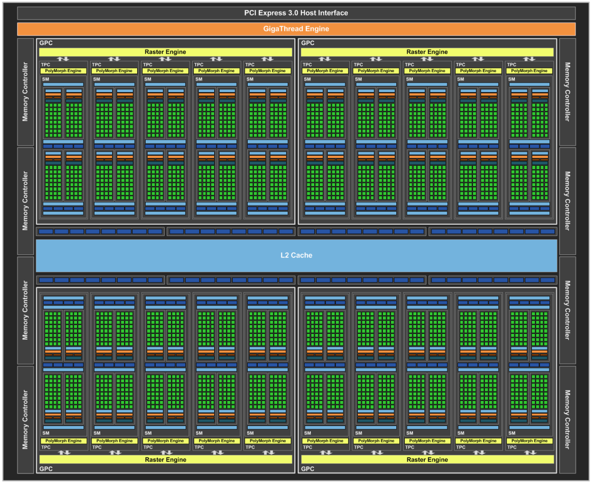
A Brief History of GPUs

- Early 1980's: bit-blit hardware for simple 2D graphics.
  - Draw lines, simple curves, rectangles, triangles, and text.
- 1989 brought the SGI Geometry Engine: basic hardware rendering through matrix-vector products.
  - Coordinate transformations: rotation, translation, scaling, perspective.
- Moore's Law growth led to more functions on GPUs
  - Shading, texture mapping, physical simulation, . . .
  - Rather than buidling dedicated hardware for each operation, GPUs became progammable

GPUs exploit data parallelism

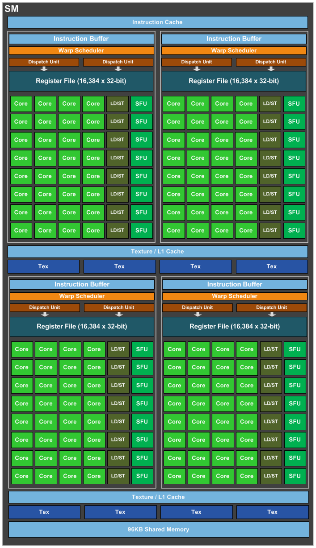
- Perform the same operations on each vertex, triangle, etc.
- The vertices, and triangles can be handled largely independently.
- In the early 2000s some intrepid researchers with experience in graphics and scientific computing realize they could use GPUs as powerful scientific computers.
- The same features make GPUs ideal for many machine learning problems.
- Big picture:
  - This is why we teach concdpts
  - Graphics, scientific computing, and machine learning all use a lot of linear algebra.
  - A solid math/science/CS foundation lets you draw the connections.
  - You also need “build stuff” skilz – the first people to use GPUs for non-graphics applications weren’t just solid at graphics and math, they are also amazing programmers.

High-End GPU Architecture



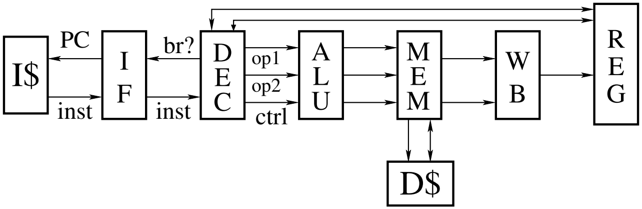
- The nVidia GTX1080 GPU chip
- SM = “Streaming Multiprocessor”
- From:

A SM



- Each SM has 128 SPs ("Streaming Processors")
- The GTX 1080 has a total of 2560 SPs
  - An SP is what nVidia calls a CUDA core.
  - With all these cores, we can see why CUDA program need so many threads.

Typical CPU Core

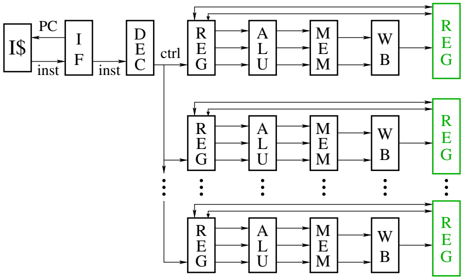


A RISC Pipeline

- Instruction fetch, decode and other control takes
  - most of the transistors and
  - much more power

than actually performing ALU and other operations!

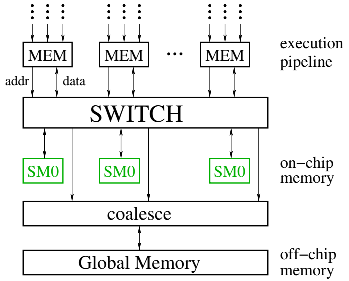
Typical GPU Core



A SIMD Pipeline

- Multiple execution pipelines execute the same instructions.
- Each pipeline has its own registers and operates on separate data values.
- Commonly, pipelines access **adjacent** memory locations.
- Great for operating on matrices, vectors, and other arrays.

What about Memory?



Memory Architecture

- On-chip “shared memory” switched between cores.
- Off-chip references are “coalesced”: the hardware detects reads from or writes to consecutive locations and combines them into larger, block transfers.

More about GPU Cores

- Execution pipeline can be very deep: 20–30 stages.
  - Many operations are floating point and take multiple cycles.
  - A floating point unit that is deeply pipelined is easier to design, can provide higher throughput, and use less power than a lower latency design.
- No bypasses
  - Instructions block until instructions that they depend on have completed execution.
  - GPUs rely on extensive multi-threading to get performance.
- Branches use **predicated execution**:
  - Execute the then-branch code, disabling the "else-branch" threads.
  - Execute the else-branch code, disabling the "then-branch" threads.
  - The order of the two branches is unspecified.
- All of these choices optimize the hardware for graphics applications.
- To get good performance, the programmer needs to understand how the GPU hardware executes programs.

Heterogenous Computing: Execution Model

A CUDA program consists of three kinds of functions:

- Host functions.
  - Called from code running on the host, but not the GPU.
  - Run on the host CPU.
  - In CUDA C, these look like normal functions.
- Device functions.
  - Called from code running on the GPU, but not the host.
  - Run on the GPU.
  - In CUDA C, these are declared with a `__device__` qualifier.
- Global functions ("kernels").
  - Called by code running on the host CPU ("launching the kernel").
  - Run on the GPU.
  - In CUDA C, these are declared with a `__global__` qualifier.

Example: `saxpy`

- Very common vector-vector operation.
- Name comes from the Basic Linear Algebra Subroutines (BLAS):

  `saxpy` = "`s`ingle precision (scalar) `a` times `x` `p`lus `y`".
- GPU version: requires both host and device code.

saxpy: host code (part 1 of 5)

```
int main(int argc, char **argv) {
  uint n = atoi(argv[1]);
  float *x, *y, *yy;
  float *dev_x, *dev_y;
  int size = n*sizeof(float);
  x = (float *)malloc(size);
  y = (float *)malloc(size);
  yy = (float *)malloc(size);
  for(int i = 0; i < n; i++) {
    x[i] = i;
    y[i] = i*i;
  }
  ...
}
```

- Declare variables for the arrays on the host and device.
- Allocate and initialize values in the host array.
  - We chose $x_i = i$ and $y_i = i^2$, but any value would do.

saxpy: host code (part 2 of 5)

```
int main(void) {
  ...
  cudaMalloc((void**)(&dev_x), size);
  cudaMalloc((void**)(&dev_y), size);
  cudaMemcpy(dev_x, x, size, cudaMemcpyHostToDevice);
  cudaMemcpy(dev_y, y, size, cudaMemcpyHostToDevice);
  ...
}
```

- Allocate arrays on the device.
- Copy data from host to device.

saxpy: host code (part 3 of 5)

```
int main(void) {
  ...
  float a = 3.0;
  saxpy<<<ceil(n/256.0),256>>>(n, a, dev_x, dev_y);
  cudaMemcpy(yy, dev_y, size, cudaMemcpyDeviceToHost);
  ...
}
```

- Invoke the code on the GPU ("launching the kernel"):
  - Create $\lceil/256\rceil$ blocks of threads.
  - Each block consists of 256 threads.
  - Each thread executes function `saxpy(...)`.
  - The pointers to the arrays (in device memory) and the values of `n` and `a` are passed to the threads.
- Copy the result back to the host.

saxpy: host code (part 4 of 5)

```
  ...
  for(int i = 0; i < n; i++) { // check the result
    if(yy[i] != a*x[i] + y[i]) {
      fprintf(stderr, "ERROR: i=%d, a[i]=%f, b[i]=%f, c[i]=%f\n",
                i, a[i], b[i], c[i]);
      exit(-1);
    }
  }
  printf("The results match!\n");
  ...
}
```

- Check the results by comparing against a serial implementation.

saxpy: host code (part 5 of 5)

```
int main(void) {
  ...
  free(x);
  free(y);
  free(yy);
  cudaFree(dev_x);
  cudaFree(dev_y);
  exit(0);
}
```

- Clean up and quit.

saxpy: device code

```
__global__void saxpy(uint n, float a, float *x, float *y) {
  uint i = blockIdx.x*blockDim.x + threadIdx.x; // nvcc built-ins
  if(i < n)
    y[i] = a*x[i] + y[i];
}
```

- We use built-in `blockIdex.x` and `threadIdx.x` indexes to distinguish between different threads.
  - Each thread has `x`, `y` and `z` versions of these indices, but we use only one dimension `x` for this example.
- We create one thread per vector element.
  - Exploits GPU hardware support for multithreading.
  - Keep in mind that there are a large but not infinite number of threads available.

`saxpy: remarks`

`saxpy<<<ceil(n/256.0),256>>>(n, a, dev_x, dev_y);`

- Kernel launch has the form

  ```
  saxpy<<<BlockDim,ThreadDim>>>(n, a, dev_x, dev_y);
  ```

- Each block executes on a single SM
  - We need at least as many blocks as we have SMs if we are going to fully utilize the GPU.
- Each SM dispatches instructions to 32 SPs at a time.
  - A group of 32 threads that execute together are called a "warp"
  - If the number of threads per warp is a multipe of 32, we get more efficient execution.
  - More importantly, accesses to the shared-memory (on-chip, per SM), and global memory (off-chip) depends on warp-level optimizations to get good performance.
- The pipelines are deep
  - The SM needs to have lots of warps to interleave in execution to achieve good performances.
- Conclusion: CUDA programs need thousands of threads.
  - And you need to understand the block/warp/thread organization to get good performance.