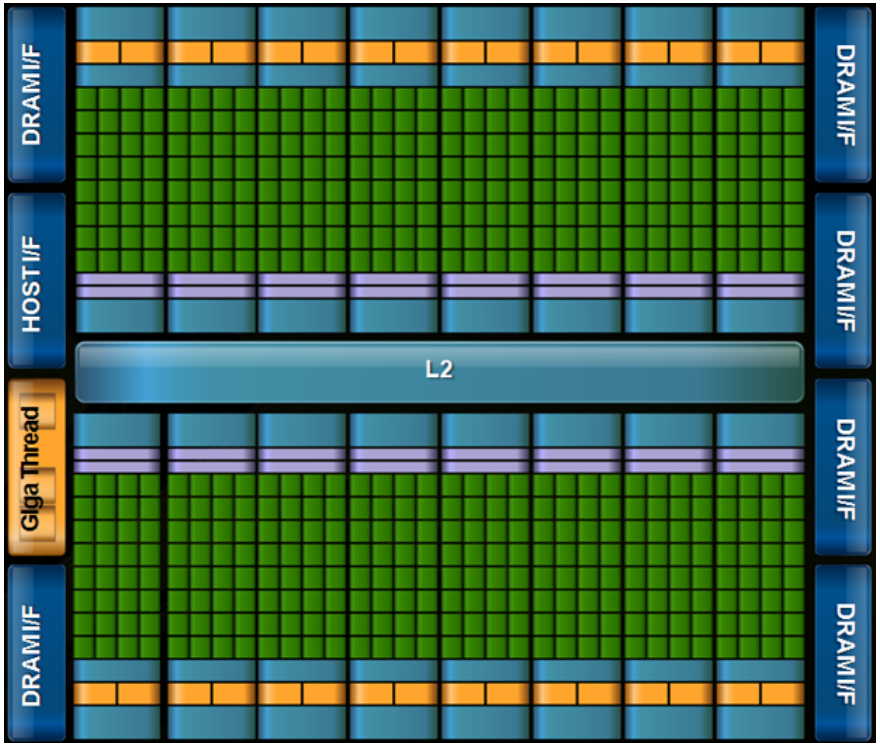
Plan for the day

- **Basic use of CUDA:**
  - **GPU memory**
  - **Writing and launching device-side functions**
  - **Transformation vs reduction**
- **Thrust library**
- **Gaussian fit**

GPU structure

- A 'Fermi' GPU has 16 'Streaming Multiprocessors', SMs.
- Each SM has 32 cores and 64KB shared memory (which doubles as L1 cache); total 512 cores.
- The Fermi also has an L2 cache, shared between all the SMs.
- There are several 'DRAM' (dynamic random access memory) banks.
- Each core in an SM is, at a given time, performing the same step on different data.

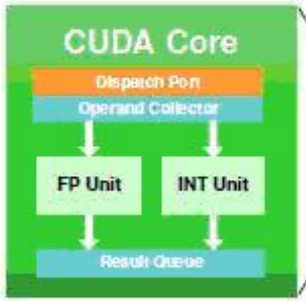


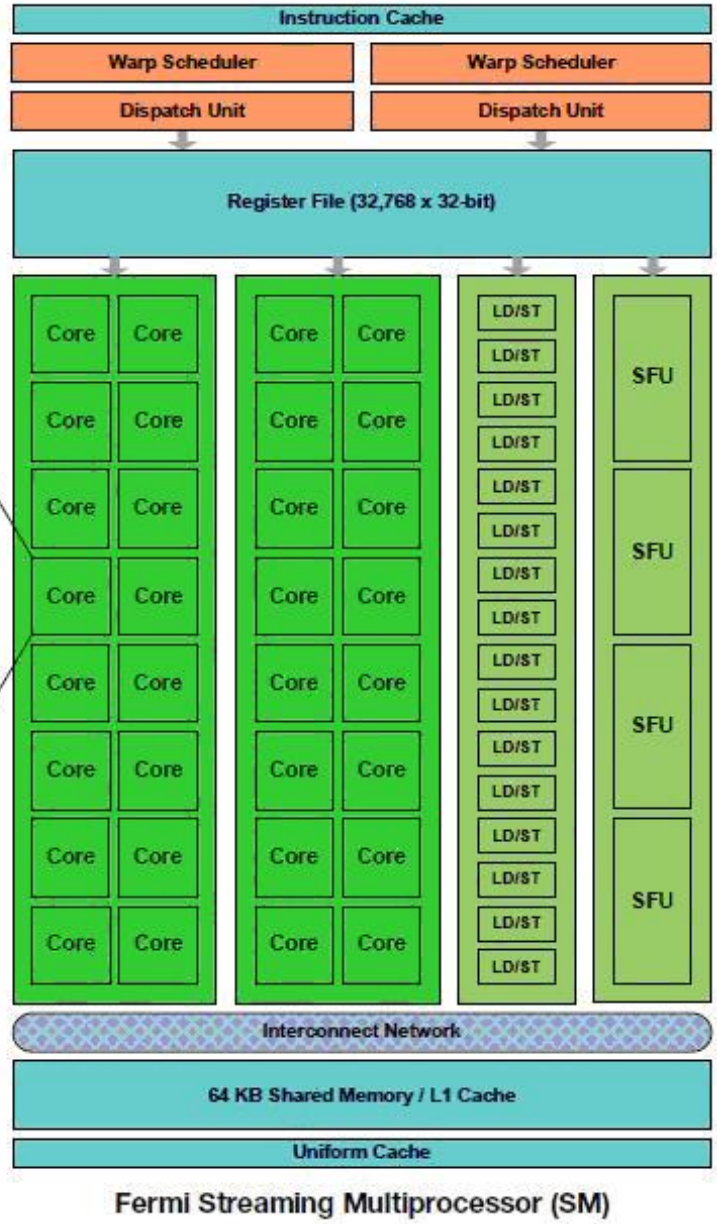
Source: nVidia

- Notice that GPU memory is a different address space from host memory; the pointer `0xBAADF00D` points to different bits of metal according to whether you're on GPU or CPU.

SM structure

- Two groups of 16 cores, all doing the same thing at each instant.
- Warp scheduler: If a group of threads ('warp') is waiting for something, eg loading something from memory, set them aside for a few cycles and execute some steps of a different group.
- Shared memory: Fast compared to main memory. Efficient 'broadcast' lookup for cases where 16 cores want the same information.





Fermi Streaming Multiprocessor (SM)

Source: nVidia

About CUDA

- **CUDA is, in no particular order:**
  - **An extension to C or C++, allowing you to insert extra keywords in your programs so that the `nvcc` compiler will make code that runs on a GPU.**
  - **A function library.**
  - **A runtime, allowing your programs to communicate with the GPU driver.**
- **Some of these overlap. For our purposes, the important thing is to write C or C++ code that executes partly on the GPU.**

Making loops into kernels

- **Suppose you want to add 1 to each of 100 data items:**

```
for (int i = 0; i < 100; ++i) {
  host_arrayOfData[i]++;
}
```

This code snippet has data parallelism: Each iteration of the loop is executed independently of the others. No calculation depends on any other calculation. In jargon, there are no loop-carried dependencies.

- **There is no need for element 84 to wait for the processing of element 83 to finish; but the CPU does it this way because it can *only do one thing at a time! How many processors do you think I have?!***
- **Ok! Ok! No need to bite our heads off!**
- **What if we had more than one processor, though? Then we might do this:**

```
// Kernel function
__global__ void device_addOneToEach (int* startOfArray) {
  startOfArray[threadIdx.x]++;
}

// Kernel launch
device_addOneToEach<<<1, 100>>>(device_arrayOfData);
```

This is referred to as a loop-kernel transformation.

Some details

- The `__global__` keyword tells the compiler that this is a device-side (GPU) function which can be called from host-side (CPU) code. We can also mark functions as `__device__`, in which case they cannot be launched as kernels but can be *called from* kernels. There is also a `__host__` keyword, which informs the compiler that this is a plain-old-vanilla C function to be run on the CPU. We can combine these like so: `__host__ __device__`, which tells the compiler to make two copies of the function: One to be called from host code, one from device code.

- The construction `someFunction<<<N, M>>>(args)` tells the compiler to create N blocks each of M threads. Each thread does one run-through of `someFunction` with argument `args`. Hence, since I had 100 data items, I wanted 100 threads; I could also have asked for 2 blocks each of 50 threads. Note that, though the code has integers, the runtime actually expects `dim3` arguments; that is, N and M are both three-dimensional. Giving only a single integer invokes the `dim3` constructor with defaults, so the other two dimensions are set to 1.

- Each block's threads are conceptually (not necessarily physically on the metal) run in parallel. If there are enough resources, two or more blocks may run simultaneously. A collection of blocks is referred to as a grid. Grid-internal scheduling decisions are made by the CUDA runtime.

- **Between the thread and the block we find the warp, a collection of threads that is executed physically, not just conceptually, in parallel. At a given clock cycle, every thread in a warp is at the same location in the code; this is not true of blocks.**
- **The variable `threadIdx.x` is a CUDA built-in; the compiler knows where to find it, although it is never declared in user code. It keeps track of which thread the current code is in. Note the `.x`; there are also `threadIdx.{y,z}`, allowing us to conceptually organise the threads in three dimensions.**
- **A similar variable `blockIdx.{x,y,z}` tracks the block; we can thus organise our threads in a three-dimensional matrix where each element is itself a three-dimensional matrix.**
- **Some limitations: Each block can have at most 1024 threads (possibly more on future cards); additionally the three dimensions of a block cannot exceed (1024, 1024, 64) (for a 650M card). We can extract these and other quantities for specific cards by calling `cudaGetDeviceProperties`:**

```
cudaDeviceProp devProp;
cudaGetDeviceProperties(&devProp, 0);
std::cout << "Maximum threads per block: "
          << devProp.maxThreadsPerBlock
          << std::endl;
```

Where does that pointer come from?

- **Each thread is given a `startOfArray` pointer, and calculated an offset from that to get the data it should look at. This pointer contains a memory location in device memory, which was called `device_arrayOfData` in the kernel invocation.**

```
__global__ void device_addOneToEach (int* startOfArray) {
  startOfArray[threadIdx.x]++;
}
```

- **At some point we need to allocate and fill this memory, or the kernel will be working with random garbage. Let's start with the CPU:**

```
// Static allocation
int host_arrayOfData_static[100];
// Dynamic using new (C++ only)
int* host_arrayOfData_new = new int[100];
// Dynamic using malloc
int* host_arrayOfData = (int*) malloc(100*sizeof(int));
```

- **On the GPU we have only the C options:**

```
// Static
__device__ int device_arrayOfData[100];
// Dynamic using cudaMalloc
int* device_arrayOfData_dynamic;
cudaMalloc((void**) &device_arrayOfData_dynamic, 100*sizeof(int));
```

Notice that `device_arrayOfData` is a pointer to device-side memory. A pointer is just a number, so the host-side code can pass it around, manipulate it, and change its value; but if you try to *look up* what is at that memory location you will get unpredictable results. So in host code you can do this:

```
// Ok, passes a device location around without doing
// anything with the memory.
device_doSomethingWithThis<<<1, 100>>>(device_arrayOfData);
```

but not these:

```
// Bad: Tries to look up a device location in host memory.
// Who knows what's there?
int currData = device_arrayOfData[3];
// Dreadful: Tries to write to a device location, but
// will in fact write to host memory. Hope we didn't
// overwrite anything important!
device_arrayOfData[3] = 5;
```

- **Now, suppose we have read in the data from a file, and filled up the host array; we need to copy the information to the device array:**

```
cudaMemcpy(device_arrayOfData, host_arrayOfData,
           100*sizeof(int), cudaMemcpyHostToDevice);
```

- **Note that exact documentation on all these functions is only a Google away, and includes several "see also" variants. Suggestion: Look at the documentation for each function mentioned here, and become at least cursorily familiar with the variants.**

Example application: Dot product

- **Now that we know the basics, let's try a simple example.**

```
__global__ void dev_vectorMult (double* vec1, double* vec2) {
  vec1[threadIdx.x] *= vec2[threadIdx.x];
}

int main (int argc, char** argv) {
  // Create and fill CPU-side vectors
  int vectorSize = atoi(argv[1]);
  double* host_vector1 = new double[vectorSize];
  double* host_vector2 = new double[vectorSize];

  for (int i = 0; i < vectorSize; ++i) {
    host_vector1[i] = rand() % 10;
    host_vector2[i] = rand() % 10;
  }

  // CPU-side multiplication
  double dotProduct = 0;
  for (int i = 0; i < vectorSize; ++i) {
    dotProduct += host_vector1[i] * host_vector2[i];
  }

  // Create and fill GPU-side vectors
  double* dev_vector1;
  double* dev_vector2;
```

```
  int sizeInBytes = vectorSize*sizeof(double);
  cudaMalloc((void**) &dev_vector1, sizeInBytes);
  cudaMemcpy(dev_vector1, host_vector1, sizeInBytes, cudaMemcpyHostToDevice);
  cudaMalloc((void**) &dev_vector2, sizeInBytes);
  cudaMemcpy(dev_vector2, host_vector2, sizeInBytes, cudaMemcpyHostToDevice);

  // Transform step: Multiply elements
  dotProduct = 0;
  dev_vectorMult<<<1, vectorSize>>>(dev_vector1, dev_vector2);

  // Reduce step: Add results
  cudaMemcpy(host_vector1, dev_vector1, sizeInBytes, cudaMemcpyDeviceToHost);
  for (int i = 0; i < vectorSize; ++i) dotProduct += host_vector1[i];

  return 0;
}
```

Transformation and reduction

- The dot-product example shows a common conceptual division: We do a transformation to each element that produces some number, in this case the product of the two vector elements; followed by a reduction, in this case summing all the products.
- This division is not so clear in the CPU version, where they are interleaved: Every multiplication is followed by an addition, if the code is written straightforwardly. We can write it to demonstrate the two-part structure, instead:

```
for (int i = 0; i < vectorSize; ++i) host_vector1[i] *= host_vector2[i];
double dotProduct = 0;
for (int i = 0; i < vectorSize; ++i) dotProduct += host_vector1[i];
```

  but this is not what you would intuitively think of first.
- We see that only the multiplications are properly data parallel - clearly, the sum of the first 83 elements and the 84th element does depend on what the 83rd element is. Consequently, only the transformation was done on the device; then we moved the resulting transformed vector back to the host before doing the reduction.
- This is not very elegant. (Although elegance isn't everything.) Maybe dot products aren't quite suitable for a loop-kernel transform?
- On the other hand, perhaps the reduction can be done on the device. Consider:

```
__global__ void device_reduce_vector (double* vector, double* result) {
  int currentArraySize = blockDim.x;

  while (currentArraySize > 1) {
    int secondHalfBegin = (1 + currentArraySize) / 2;
    if (threadIdx.x + secondHalfBegin < currentArraySize) {
      vector[threadIdx.x] += vector[secondHalfBegin + threadIdx.x];
    }
    __syncthreads();
    currentArraySize = secondHalfBegin;
  }

  if (0 == threadIdx.x) (*result) = vector[0];
}
```

- **This is $\mathcal{O}(\log_2(N))$ instead of $\mathcal{O}(N)$.**
- **Note that `blockDim.x` is another CUDA built-in; it stores the width of the current block. For our case it is 100.**
- **`__syncthreads()` is a thread block: No thread will go past the point where it is called until every thread *in the block* has reached it.**

Memory types

- **CUDA has several kinds of memory:**
  - **Global: Vanilla memory store for general use.**
  - **Constant: Special memory to which threads are not allowed to write. This allows some optimisation in lookups.**
  - **Shared: Small, fast cache local to each block.**
  - **Texture: Special memory optimised for locality in one or two dimensions; ie if you look up location (i, j), it is fast to look up $(i \pm 1, j \pm 1)$.**
  - **Pinned: Host-side memory accessed directly from the GPU.**
- **Constant memory must be declared statically:**

```
__constant__ double dataArray[100];
```

and written to using special `cudaMemcpyToSymbol` method:

```
double host_dataArray[100];
readDataFromFile(host_dataArray);
cudaMemcpyToSymbol(dataArray, host_dataArray, 100*sizeof(double));
```

- **Shared memory is usually declared statically:**

```
__global__ void someCalculation (int* args) {
  __shared__ double threadCache[1024];
  threadCache[threadIdx.x] = someCalculation(args);
  __syncthreads();
  // All elements of threadCache have been initialised and
  // can be accessed by all threads in this block; lookup is
  // fast.
}
```

Jargon summary

- **Host and device: The CPU and GPU, respectively.**
- **Thread: The code to be executed on a single subprocessor.**
- **Warp: A group of threads *physically* executed in parallel.**
- **Block: A *conceptually* parallel group of threads. Not necessarily executed physically simultaneously.**
- **Grid: A set of blocks; a 'job'.**
- **Loop-kernel transformation: The conceptual recoding of a for loop as a set of threads - each thread corresponding to one iteration of the loop.**
- **Data parallelism: The state in which what is done to data element X_i is independent of what happens to X_j, for all i, j.**
- **Kernel function: Code to be executed on the device, in parallel. Sometimes called 'curnels', for "CUDA kernel".**
- **Kernel launch: Invoking a curnel, with the `<<<N, M>>>` syntax specifying the thread and block dimensions.**
- **Transformation: An operation which is done on every element, such as multiplying a vector by a scalar.**
- **Reduction: An operation that reduces the number of elements, such as taking a sum, or finding the maximum.**