CUDA Programming

NVIDIA GPU architectures

- **Kepler**
  - **SMX**: 3X performance/watt compared to SM (1 petaflop of computing in 10 server racks)
  - **dynamic parallelism**:
  - **hyper-Q**:
- PCIe 3.0: 8GT/s (gigatransfers per second[1])

Kernels

- A **kernel** is a C function, when called, which is executed N times in parallel by N different CUDA **threads**. A kernel is defined using the specifier `__global__`.
- The programmer organizes threads into a hierarchy of **grids** of **thread blocks**.
- A **thread block** is a set of concurrent threads that can cooperate among themselves through barrier synchronization and shared access to a memory space private to the block.
- A **grid** is a set of *thread blocks* that may be executed *independently* and thus may execute in parallel.
- **execution configuration**: each kernel call can be called with *caller-specific* configuration which specifies how threads will be deployed in CUDA processors. A execution configuration has the form

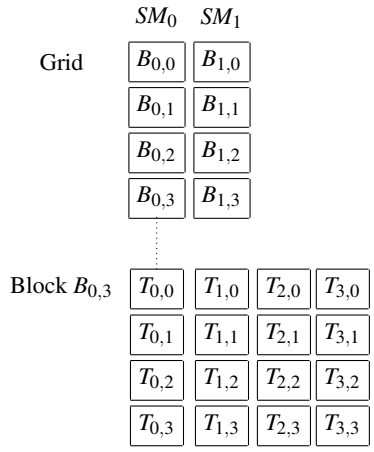
  $$\texttt{<<<}\ G,B,M,S\ \texttt{>>>},$$

  where
  - G (type `dim3`) specifies the dimension and size of **grid** s.t. $G_x * G_y * G_z$ equals the number of **blocks** being launched (for devices of compute capability 1.x, $G.z$ must be 1). G can be an integer if there are only one dimension in a grid.
  - B (type `dim3`) specifies the dimension and size of each **block** s.t. $B_x * B_y * B_z$ equals the number of **threads** per block. B can be an integer if there are only one dimension in a block.
  - optional M (type `size_t`), specifies the number of *bytes* in shared memory that is *dynamically allocated per block* for this call in addition to the statically allocated memory. Default value is 0 and M must not exceed the amount of shared memory available.
  - optional S (type `cudaStream_t`) specifies the associated stream. Default value is 0.

[1] The **data transfer rate** is defined as **channel width (bits/transfer)** × transfers/second = bits/second.

Thread hierarchy

- `threadIdx`: 3-component vector variable which specifies the unique location of a thread inside a **thread block**. This variable is *a per-thread variable* and each thread will have a different value which is unique w.r.t. a block.
- A **thread ID**, a number unique to a thread within a block, can be computed from the value of `threadIdx`.
  - In 2-dimensional block of size (B_x, B_y), the thread id is $x + yD_x$.
  - In 3-dimensional block of size (B_x, B_y, B_z), the thread id is $x + yD_x + zD_xD_y$.
- **example**: There are two SMs and we can schedule 4 blocks to each SM. Within a thread block, we can have 16 threads.



- **kernel invocation example**:

```
dim3 threadsPerBlock(4, 4);
dim3 blocksInGrid(N/threadsPerBlock.x,
                  N/threadsPerBlock.y);
add<<<blocksPerGrid, threadsPerBlock>>>(...);
```

Kernel grid execution

- When a CUDA program on the host CPU invokes a kernel grid, the **CWD (compute work distribution) unit** enumerates the blocks of the grid and begins distributing them to SMs with available execution capacity.
- The threads of a *thread block* execute concurrently on one SM.
- As thread blocks terminate, the CWD unit launches new blocks on the vacated SMs.
- **Tesla architecture example**:
  - One SM creates, manages, executes up to 768 concurrent threads in hardware with zero scheduling overhead.
  - It can execute up to 8 CUDA thread blocks concurrently.
  - **single-instruction, multiple-thread (SIMT)**

	Tesla	Fermi		Kepler	
GPU		GF100	GF104	GK104	GK110
capability		2.0	2.1	3.0	3.5
threads/warp		32	32	32	32
max warps/SM(X)		48	48	64	64
max threads/SM(X)		1536	1563	2048	2048
max thread blocks/SM(X)		8	8	16	16
32-bit registers/SM(Xx)		32768	32768	65536	65536
max registers/thread		63	63	63	255
max registers/thread block		1024	1024	1024	1024
SM(X)					
memory/board				8 GB	TBA
memory bandwidth				320G Bps	TBA
dynamic parallelism				YES	YES
Hyper-Q				YES	YES

Figure 1: Comparison of NVIDIA GPU architectures

  - One thread is mapped to one SP scalar core.
  - Each scalar thread executes independently with its own instruction address and register state.
  - **SM SIMT** unit creates, manages, schedules, and executes threads in groups of 32 parallel threads called **warps**.
  - Each SM manages a pool of 24 warps of 32 threads, a total of 768 threads.
  - Every **instruction issue time**, the SIMT unit selects a warp that is ready to execute and issues the next instruction to the active threads of the warp.
  - A warp executes one common instruction at a time, so full efficiency is realized when all 32 threads of a warp agrees on their execution path.

Memory hierarchy

- A thread has a **per-thread local memory**.
- A thread block has a **per-block shared memory**, which is shared by all threads in the thread block.
- A grid (or multiple grids) has a **global memory**, which is shared by all threads with grids.
- In CUDA programming model, there are two memory spaces: **host memory** and **device memory**. The device memory is further classified into **global memory**, **constant memory**, and **texture memory**, which are visible to kernels through calls to CUDA runtime.

CUDA programming interface

- The **CUDA driver API** is the lowest-level API which is used by **CUDA runtime API**. Users can also use this API for fine-control over the devices. It exposes low-level concepts such as **CUDA contexts** (similar to processes in the host) and **CUDA modules** (similar to DLLs).

NVCC compilation

- The `nvcc` compilation flow is as follows.
  1. Separate the device code from host code.
  2. Compile the device code into assembly form (**PTX code**) and/or binary form (**cubin** object).
  3. Instrument the host code so that execution configuration (`<<<...>>>`) is replaced with CUDA runtime function calls to load and launch each compiled kernel from the PTX code and/or cubin object.
- Instrumented host code can be either compiled by `nvcc` or manually by a user-defined tool. Or, users can directly control the device code using CUDA driver API to load and execute PTX code or cubin object.
- **JIT compilation**: PTX code loaded by an application at runtime is compiled into binary code by the device driver on the fly. This JIT compilation increases load time, but allows applications to benefit from latest compiler improvements.
- Device driver maintains the **compute cache** which is used to cache a copy of binary code generated from PTX code through JIT compilation. JIT compilation can be controlled by following environment variables
  - `CUDA_CACHE_DISABLE`: disable cache
  - `CUDA_CACHE_MAXSIZE`: specifies the cache in bytes (default is 32MB and maximum is 4GB)
  - `CUDA_CACHE_PATH`: the folder where the cache files are stored (default is `~/.nv/ComputeCache` on Linux)
  - `CUDA_FORCE_PTX_JIT`: when 1, forces the device driver to ignore any binary code embedded in an application but to JIT compile embedded PTX code instead

PTX compatibility

- Some PTX instructions are only supported on devices of higher **compute capability**. For example, *atomic instructions on global memory* are supported on devices of compute capability of 1.1 and above.
- The `-arch` compiler option specifies the compute capability that is assumed when compiling C to PTX code. For example, the code that contains double-precision arithmetic must be compiled with "`-arch=sm_13`" option.

Application compatibility

- Which PTX and binary code gets embedded in a CUDA C application is controlled by the `-arch`/`-code` options or the `-gencode` option.
- For example, the compiler option

  ```
  -gencode arch=compute_10,code=sm_10
  ```

  embeds binary code compatible with compute capability 1.0.

CUDA C runtime

- The runtime is implemented in the `cudaart` dynamic library which is typically included in the application installation package. All its entry points are prefixed with `cuda`.
- The CUDA runtime is **initialized** when the first runtime function call happens. During initialization, the runtime creates a **CUDA context** for each device, which is called the **primary context** for the device shared among all the host threads of the application.

Device memory

- Device memory can be allocated either as **linear memory** or as **CUDA arrays**.
- Linear memory is typically allocated and freed using `cudaMalloc()` and `cudaFree()`, respectively. The data transfer between host and device memory is done using `cudaMemcpy()`.
  - Linear memory can also be allocated through `cudaMallocPitch()` and `cudaMalloc3D()`, for allocations of 2D or 3D arrays as it makes sure that the allocation is appropriately **padded** to meet the alignment requirements. Also, `cudaMemcpy2D()` and `cudaMemcpy3D()` are used for data transfer.
- As for the linear memory, Devices of compute capability 1.x has 32-bit address space (4GB). Devices of compute capability have 40-bit address space (1TB).

Shared memory

- A shared memory is allocated using the `__shared__` qualifier.

Constant memory

Texture memory

Overview

- suitable for **data-parallel computation**: the same program executed on many data elements in parallel (with **high arithmetic intensity** – the ratio of arithmetic operations to memory operations)
- NVIDIA device consists of multiple **streaming multiprocessors (SM)**
- **SM** consists of 8 **scalar thread processors (SP)**

NVIDIA GeForce 9500 GT

- CUDA cores: 32 (4 SMs $\times$ 8 cores/SM)
- memory: 1GB (128-bit interface)
- bus: PCI express x16 gen1 (max width: x16; max speed: 2.5 GT/s)
- graphics clock: 550MHz
- memory clock: 400MHz
- processor clock: 1375MHz
- cuda capability: 1.1

Development environment

- CUDA-enabled GPU
- NVIDIA device driver: allows communication with GPU
- CUDA development toolkit
- standard C compiler
- **flow**: CUDA program $\rightarrow$ standard C program with CUDA toolkit funcalls $\rightarrow$ assembler

SIMT thread execution

- **warp** (**"thread-vector"**): a set of parallel threads that execute a single instruction
  - groups of 32 threads formed into **warps**
  - these threads executes the same instruction
  - some become inactive when code path diverges
  - *hardware automatically handles divergence*
- **warps are the primitive unit of scheduling**
  - pick 1 of 32 warps for each instruction slot
  - note that warps may be running different programs/shaders!
- SIMT execution is an *implementation choice*
  - sharing control logic leaves more space for ALUs
  - largely invisible for programmer
  - must understand for performance, not correctness

Splitting parallel blocks

- **thread block**: 3-dimensional vector; may contain up to 1024 threads (4.2)
  - **blockDim.{x,y,z}**: 1/2/3-dimensional **thread block** space
  - **threadIdx.{x,y,z}**: thread index for each block dimension
  - two-dimensional block of size (D_x, D_y): thread ID of a thread of index (x, y) is $x + yD_x$
  - three-dimensional block of size (D_x, D_y, D_z): thread ID of a thread of index (x, y, z) is $x + yD_x + zD_xD_y$.
- **grid**: can have 1, 2-dimensional grid of thread blocks
  - **gridDim.{x,y}**: 1/2-dimensional **grid** space

Kernel function call

- **kernel**: `__global__` declaration specifier
- `add<<nblocks, nthreads_per_block>>(args)`
  - total # threads: `nblocks` $\times$ `nthread_per_block`
- `add<<<N, 1>>>(args)`: N threads all in one block
  - `int tid = blockIdx.x`
- `add<<<1, N>>>(args)`: one thread in N blocks (one thread in each block)
  - `int tid = threadIdx.x`
- given that we want 128 threads per block BUT (1) and/or N can be less than 128 and/or (2) N is not multiple of 128

  (1) **kernel call**: `add<<<(N+127)/128, 128>>>`

  (2) **guard inside kernel**:

```
if (tid < N) {
   // guard against !(N mod 128)
   c[tid] = a[tid] + b[tid];
}
```

Warp divergence

- **given**:

```
if (x < 0.0)
   z = z - 2.0;
else
   z = sqrt(x);
```

- **predication**: compute predicate first and then execute two predicated instructions

```
p = (x < 0.0);
(p):  z = x - 2.0; // single instruction
(!p): z = sqrt(x);
```

  - both branches are executed; so, large branch (which actually is taken by only one thread) code causes all other threads to execute it
- **warp voting**: if branches are big, **nvcc** inserts code to check if all threads in the warp take the same branch

Memory: Optimize memory access

- coalesced vs non-coalesced = order of magnitude
  - **coalesced memory transaction**: one in which all of the threads in a half-warp access global memory at the same time. e.g. if threads 0, 1, 2, 3 read global memory 0x0, 0x4, 0x8, 0xc, it is a coalesced read.
  - Example: given an array: 0 1 2 3 4 5 6 8 9 a b

```
CASE #1:
  thread 0: 0, 1, 2
  thread 1: 3, 4, 5
  thread 2: 6, 7, 8
  thread 3: 9, a, b
CASE #2:
  thread 0: 0, 4, 8
  thread 1: 1, 5, 9
  thread 2: 2, 6, a
  thread 3: 3, 7, b
```

    CASE #2 is a coalesced since the "first access" is 0, 1, 2, 3. If threads in a block are accessing consecutive global memory locations, then all accesses are combined into a **single request** (or **coalesced**) by the hardware.
- optimize for spatial locality in cached texture memory
- in shared memory, avoid *high-degree bank conflicts*

Memory: Take advantage of shared memory

- hundreds of times faster than global memory
- threads can cooperate via shared memory
- *use one/a-few threads to load/compute data shared by all threads*
- use it to avoid non-coalesced access: stage loads and stores in shared memory to re-order non-coalesceable addressing

References

[1] NVIDIA. NVIDIA CUDA C Programming Guide, version 4.2, April 2012.