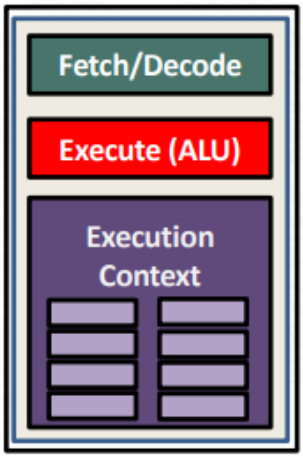
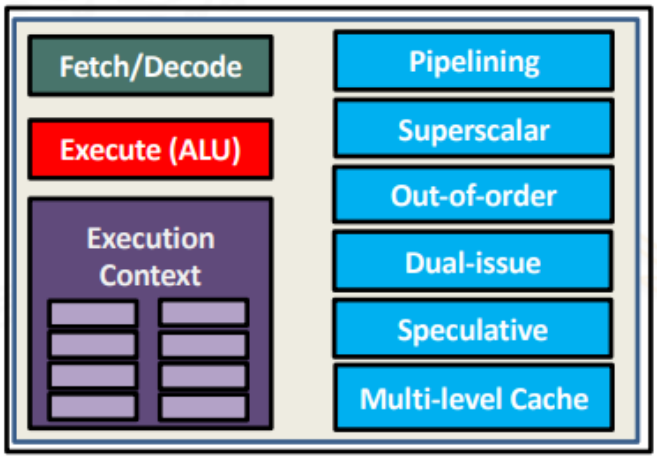
M6 GPU Archi + CUDA Programming

1. CPU Architecture

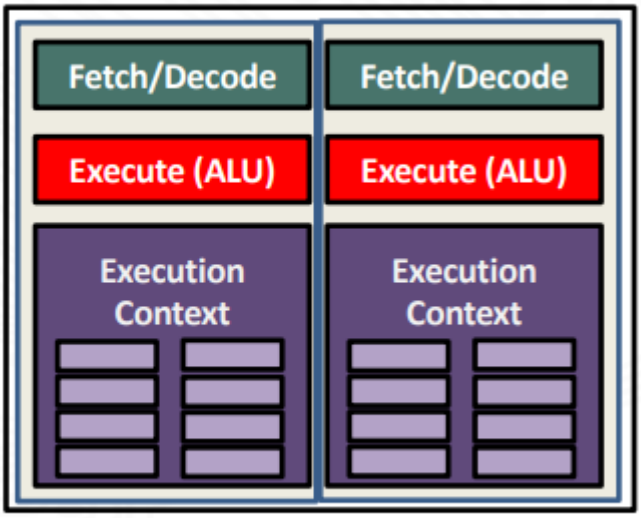
Single Core

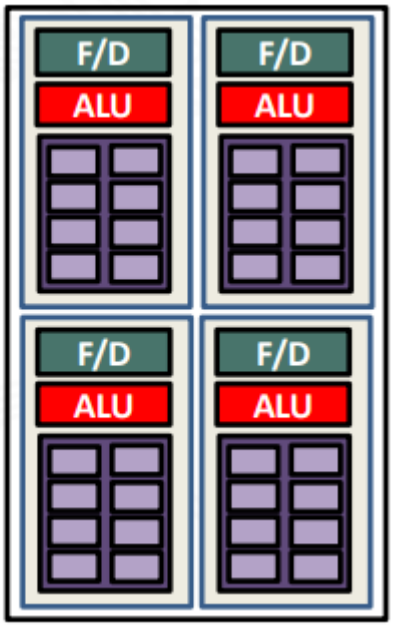




- Designed using various performance enhancement techniques through sophisticated processor logic circuitry for single-threaded code optimised for low latency

Multicore





- Adds more cores to enable execution of multiple instructions in parallel; effective if efficient concurrent code is written

Combining with SIMD (Single Instruction Multiple Data) ALU

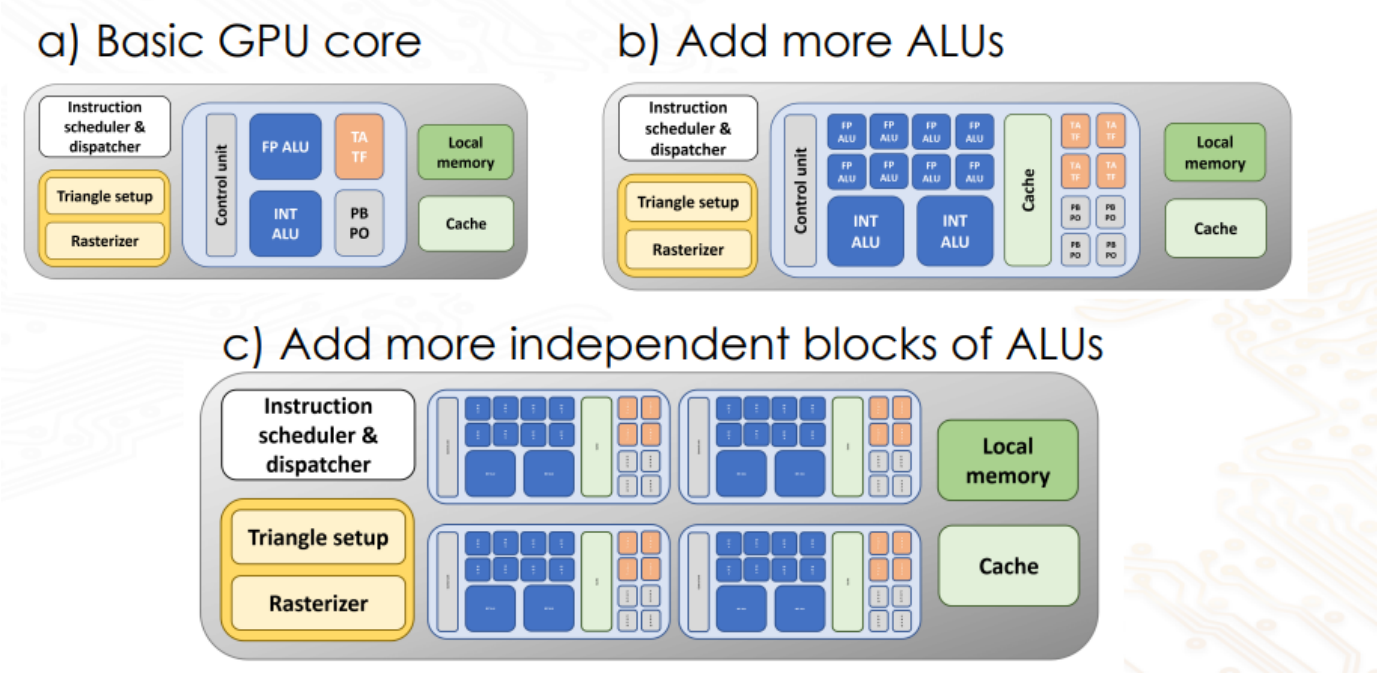
- ALU is duplicated within each core and all of them must execute the same instruction simultaneously to support SIMD operations

2. GPU Architecture

Basic GPU

- Consists of many streaming multiprocessors (SM) and compute units (CU), each with many streaming processors (SP; something like ALU) to support SIMT (SIMThreads) operations

Unified GPU



- Consists of processor array in the form on SMs, each containing multiple SP and can manage many concurrent SIMD threads

2.1 GPU Computing with CUDA Programming

- CUDA refers to general purpose parallel computing programming model where a problem is expressed in the form of multiple threads and executed in parallel using CUDA cores in GPU, enabling many complex computational problems to be solved in a very efficient way
  - Each sub problem is allocated to a thread block which is then further divided into finer pieces that can be solved cooperatively in parallel using multiple threads within the thread block
- A program will start and run by the host (CPU) and launch parallel threads that are executed on the device (GPU)
  - Program file is stored with a `.cu` extension which is compiled using NVIDIA compiler nvcc

Device code

- `__global__` declaration specifier is used to indicate that code should be ran on the device, which will then be launched as Kernel in CUDA and can be called from host by appending `<<<x,y>>>` to the back of the calling function

```
__global__ void hello_GPU(void){
  printf("Hello from GPU!");
  }
```

```
int main(void){
  hello_GPU<<<1,1>>>();
  printf("Hello from CPU!");
  return 0;
  }
```

(source file gets split into host and device components during compilation and are compiled separately)

2.2 Kernel and Parallel Threads

- Kernel is an extended C function and can be executed N times **in parallel** by N different CUDA threads when called
- Number of parallel CUDA threads launched for kernel is specified in the host code using execution configuration syntax (`<<<x,y>>>`)
  - `x` - number of thread blocks
  - `y` - number of threads in each block
  - Total threads $= x \times y =$ number of copies of kernel executed
- Different threads that executes a copy of the kernel is given a unique thread ID within the block and is accessible within the kernel through `threadIdx`
- Each block that executes the kernel also has a block ID accessible within the kernel through `blockIdx`
  - Threads among different blocks can be identified using `blockIdx` and `threadIdx` together with `blockDim` (number of threads per block)
  - Index of thread would be retrieved by using `blockIdx.x * blockDim.x + threadIdx.x`, where `blockIdx.x * blockDim.x` refers to the number of threads that are executing in parallel in previous blocks and `threadIdx.x` refers to the position of the thread in its own block
- Threads and Block indexes are extended with `.x` since they can be defined to be of 1D, 2D (with x, y) or 3D (with x,y,z) which are suitable for addressing complex problems described in multi-dimensions

2.3 Synchronisation

- `cudaDeviceSynchronize()` function is required when we need the host to wait for all threads to complete kernel execution

2.4 Parameter passing

- In cases like vector addition, it is most suitable to be executed by using the GPU using 3 threads (for 3D vector)
- Data for vectors can be passed from host to device using CUDA memory management functions `cudaMalloc(), cudaMemcpy(), cudaFree()`
- **Host code**

```
int main(void){
  int N = 3
  int a[N] = {7,2,3};
  int b[N] = {6,4,5};
  int c[N];

  int *d_a, *d_b, *d_c;
  cudaMalloc((void**)&d_a, sizeof(int)*N);
  :    : // repeat for d_b and d_c
  cudaMemcpy(d_a, a, sizeof(int)*N, cudaMemcpyHostToDevice);
  cudaMemcpy(d_b, b, sizeof(int)*N, cudaMemcpyHostToDevice);

  vector_add_cu<<<1,1>>>(d_c, d_a, d_b, N); // note: 1 thread

  cudaMemcpy(c, d_c, sizeof(int)*N, cudaMemcpyDeviceToHost);
  cudaFree(d_a);
  :    : // repeat for d_b and d_c
  }
```

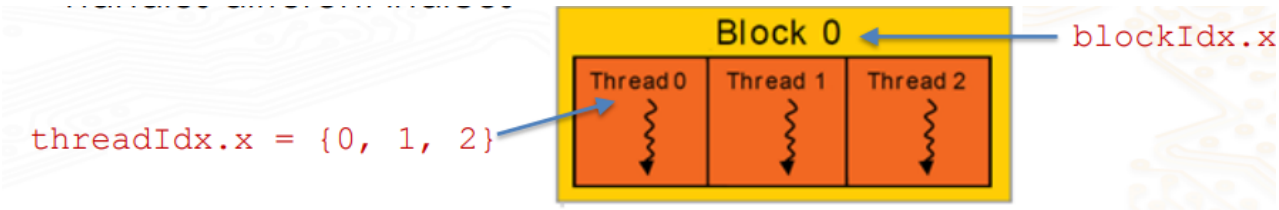
(in this image, note that one threadblock containing 1 thread is assigned)

- **Device code**

```
__global__
void vector_add_cu(int *d_c, int *d_a, int *d_b, int n){
   for (int i = 0; i<n; i++)
       d_c[i] = d_a[i] + d_b[i];
   }
```

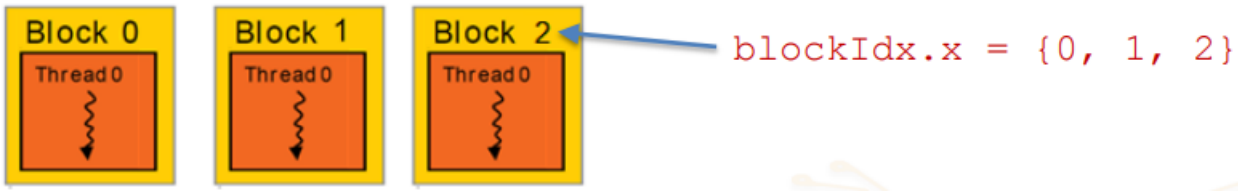
  - No parallel operation using the above host code since `<<<1,1>>>`
  - Both `<<<1,3>>>` or `<<<3,1>>>` can work but parallel threads within a block have the additional mechanisms to directly communicate and synchronise with each other. (i.e. `<<<1,3>>>` is preferred)

- When we have multiple threads running, there is a need to specify to each thread which element of the vector it should compute
  - If `<<<1,3>>>` is used, `threadIdx.x` is used to index the vector elements, otherwise `blockIdx.x` is used




```
__global__
void vector_add_cu(int *d_c, int *d_a, int *d_b, int n){
   d_c[threadIdx.x] = d_a[threadIdx.x] + d_b[threadIdx.x];
   }
```

31




```
__global__
void vector_add_cu(int *d_c, int *d_a, int *d_b, int n){
   d_c[blockIdx.x] = d_a[blockIdx.x] + d_b[blockIdx.x];
   }
```

- For dot product of vectors, where each element is multiplied to the corresponding element and the sum of all operations give the answer, it can be effectively computed by launching parallel threads

$$\vec{\mathbf{a}} \cdot \vec{\mathbf{b}} = \sum_{i=0}^{2} a_i b_i = a_0 * b_0 + a_1 * b_1 + a_2 * b_2$$

```
__global__
void dot_prod_cu(int *d_c, int *d_a, int *d_b){
  tmp[3];
  int i = ?
  tmp[i] = d_a[i] * d_b[i];

  if (i==0){
    int sum = 0;
    for (int j = 0; j < 3; j++)
      sum = sum + tmp[j];
    *d_c = sum;
    }
  }
```

40

  - Declaring `int i = blockIdx.x` or `threadIdx.x` will not work since threads need to access to each other's data; shared memory can be prepended with `__shared__` to allow access, and `__syncthreads()` is used to synchronise between threads when a thread is ready to share

data

```
__global__
void dot_prod_cu(int *d_c, int *d_a, int *d_b){
  __shared__ int tmp[3];
  int i = threadIdx.x;
  tmp[i] = d_a[i] * d_b[i];
  __syncthreads();
  if (i==0){
    int sum = 0;
    for (int j = 0; j < 3; j++)
      sum = sum + tmp[j];
    *d_c = sum;
    }
  }
```

- **variables passed into global function need to be pointers and are accessed with indexes (eg. `d_a[i]`)**

3. GPU Internal Operation

3.1 Software / Hardware Perspectives

- When a kernel is launched, each SP core will execute one thread
- **Streaming Multiprocessor Block Allocation**
  - During runtime each block of threads can be scheduled on any of the available SM within a GPU in any order, sequentially, or concurrently; but block cannot be split among multiple SMs (i.e. all threads in block must be scheduled to the same SM)
    - Each SM can only handle a maximum of 8 blocks with the actual number constrained by available resource

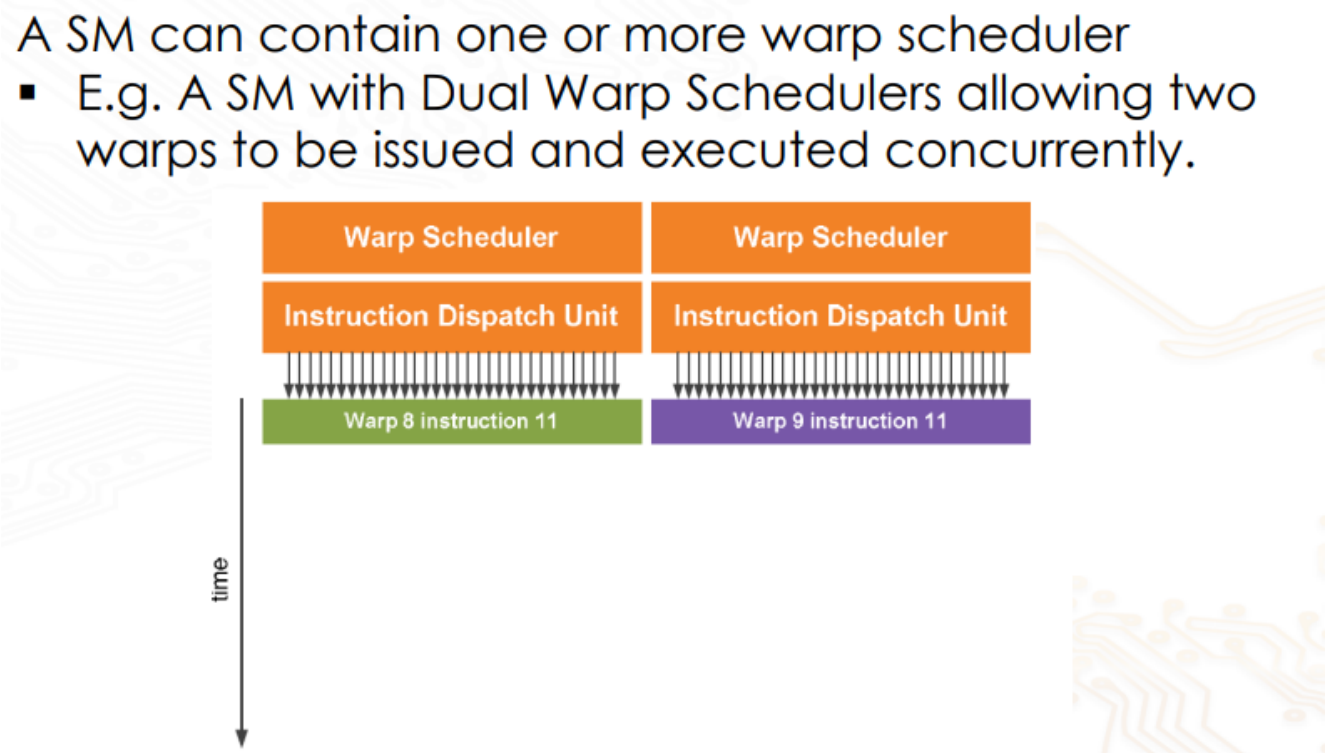
Blocks vs Threads

- In previous section, both using 1 block of multiple thread and multiple blocks of 1 thread are not ideal since
  - 1 block - only one SM is used per GPU
  - 1 thread - potentially only one core is used per SM if blocks are spread across different SM
- GPU parallel computing is only effective if massive amount of parallel data are available which also compensates for overhead of copying data between host and device

Warp

- When a SM is given one or more thread blocks to execute, the threads are partitioned into groups of 32 (Warp) with all threads in a warp executing the same instruction with different data (SIMD)
  - Threads in warp will have consecutive `threadIdx.x` (0-31 for first warp, 32-63 for second warp etc)

- Each warp then gets scheduled by a warp scheduler for execution since the number of CUDA cores in a SM is usually less than total number of threads that are assigned to it. At any one time, **only 1 warp** is selected for execution
- While a warp is waiting for the results of a previously executed instruction, a different warp that is ready is selected for execution (based on a certain prioritised scheduling policy), 'removing' the latency. $\Longrightarrow$ More efficient if many threads and many blocks



SIMT Architecture

- SM employs a unique architecture called Single Instruction Multiple Thread where a warp executes one common instruction for all its threads at a time.
- Within a single thread, instructions are pipelined to achieve instruction-level parallelism and are issued in order with **no branch prediction and speculative execution**
    - Individual threads in a warp start together at the same instruction address but each has its own instruction address counter and registers and are free to branch and execute independently when the thread diverges (due to data-dependent conditional execution and branch)

3.2 Hardware Resource Allocation

- On-chip resources are used to store the execution context (PC, registers, shared mem.) for each warp processed by a SM. Context is maintained during the entire lifetime of the warp, allowing fast switching between warps without the need of the conventional CPU's context switching

Resource Usage

- Resources are limited $\Longrightarrow$ number of threads that can be executed in a SM is capped for a given application
    - More resources a thread requires, the less number of threads that can simultaneously reside in SM

Example: Register Usage
Supposed a SM can accommodate upto 1536 threads and has 16384 registers.

To accommodate 1536 threads
- each thread can use no more than 16384/1536 = 10 registers.

If each threads requires 12 registers
- the number of threads that can simultaneously reside in the SM has to be reduced.

Reduction is done by removing a whole block
- if each block contains 128 threads
- reduction of threads will be done by reducing 128 threads at a time.

Another example: Shared memory usage
Suppose that a CUDA GPU has 24KB of shared memory per SM .

Suppose that each SM can support up to 8 blocks.
- each block must use no more than 3KB of shared memory.

If each block uses 5KB of shared memory for a particular application
- no more than 4 blocks can reside in a SM.

What happen if there are not enough registers or shared memory available per SM for at least one block?
- the kernel will fail to launch.

- Once a thread block is launched on a SM, all its warps are resident until executions are finished $\rightarrow$ no new block will be launched on an SM until there is sufficient number of free registers for all warps of the new block + there is enough free shared memory for the new block

Warp Issuing Efficiency

- Each scheduler must have at least one warp eligible to issue an instruction every clock cycle to hide latencies between instructions
    - Ineligible warps (not ready) due to input operands not ready / loading data from global memory / waiting at synchronising point
- During the execution of kernel, need to maintain as many active warps as possible to avoid situations where all warps and not ready and no instructions are issued.