软件学院　杨伟光

CUDA programming

Menu

HPC&IP

- Programming Model
- Execution Model
- Memory Model
- Optimization
- Streams
- CUDA Libraries

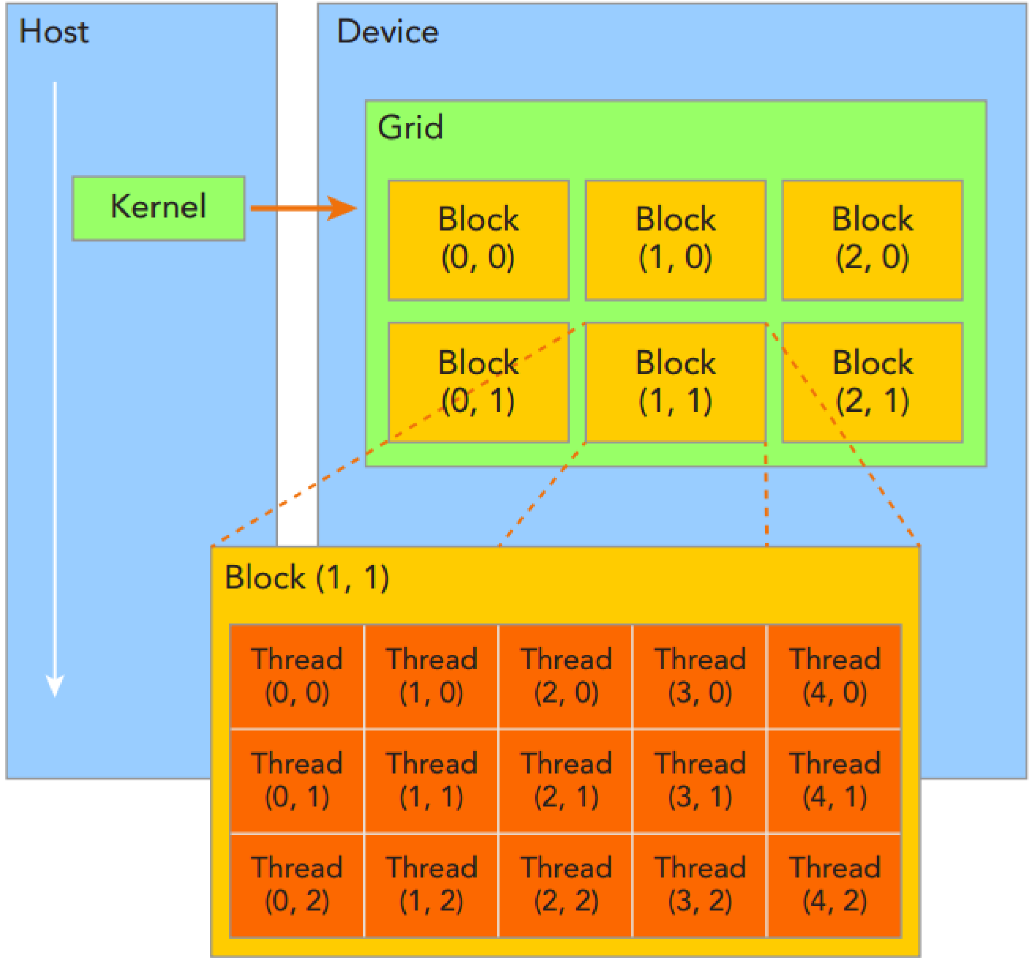
Menu

CUDA Tools

Experience to Learn CUDA

Programming Model

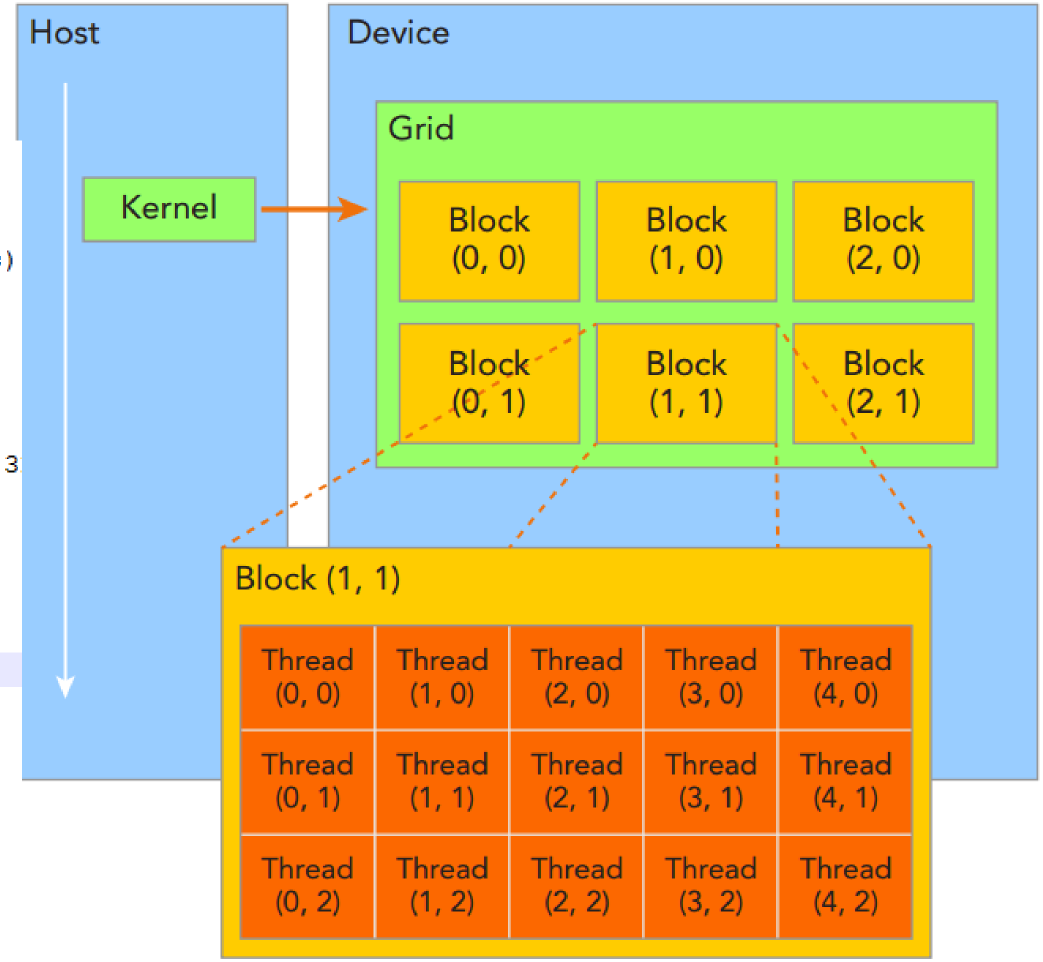
Grid: all threads in one kernel
Block: one grid including several blocks



Programming Model

Grid: all threads in one kernel
Block: one grid including several

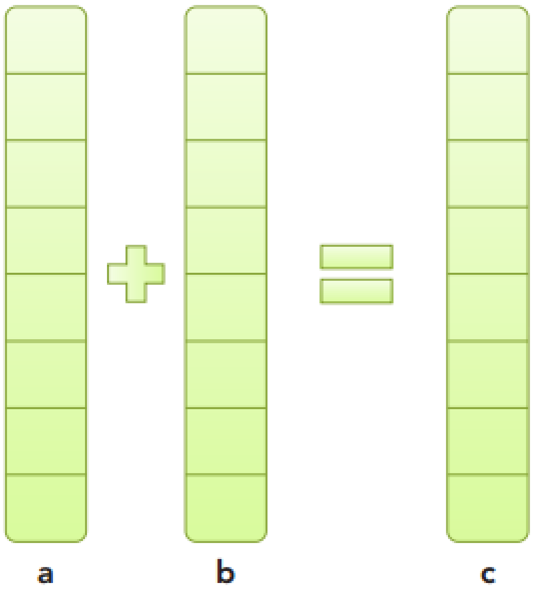
```
Device 0: "Tesla K40c"
  CUDA Driver Version / Runtime Version          7.0 / 7.0
  CUDA Capability Major/Minor version number:    3.5
  Total amount of global memory:                 11520 MBytes (12079136768 bytes)
  (15) Multiprocessors, (192) CUDA Cores/MP:     2880 CUDA Cores
  GPU Max Clock rate:                            745 MHz (0.75 GHz)
  Memory Clock rate:                             3004 Mhz
  Memory Bus Width:                              384-bit
  L2 Cache Size:                                 1572864 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65536), 3
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Enabled
  Device supports Unified Addressing (UVA):      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 1 / 0
```



Programming Model

- Example

vectorAdd



A typical processing flow of a CUDA program follows this pattern:
1. Copy data from CPU memory to GPU memory.
2. Invoke kernels to operate on the data stored in GPU memory.
3. Copy data back from GPU memory to CPU memory.

Programming Model

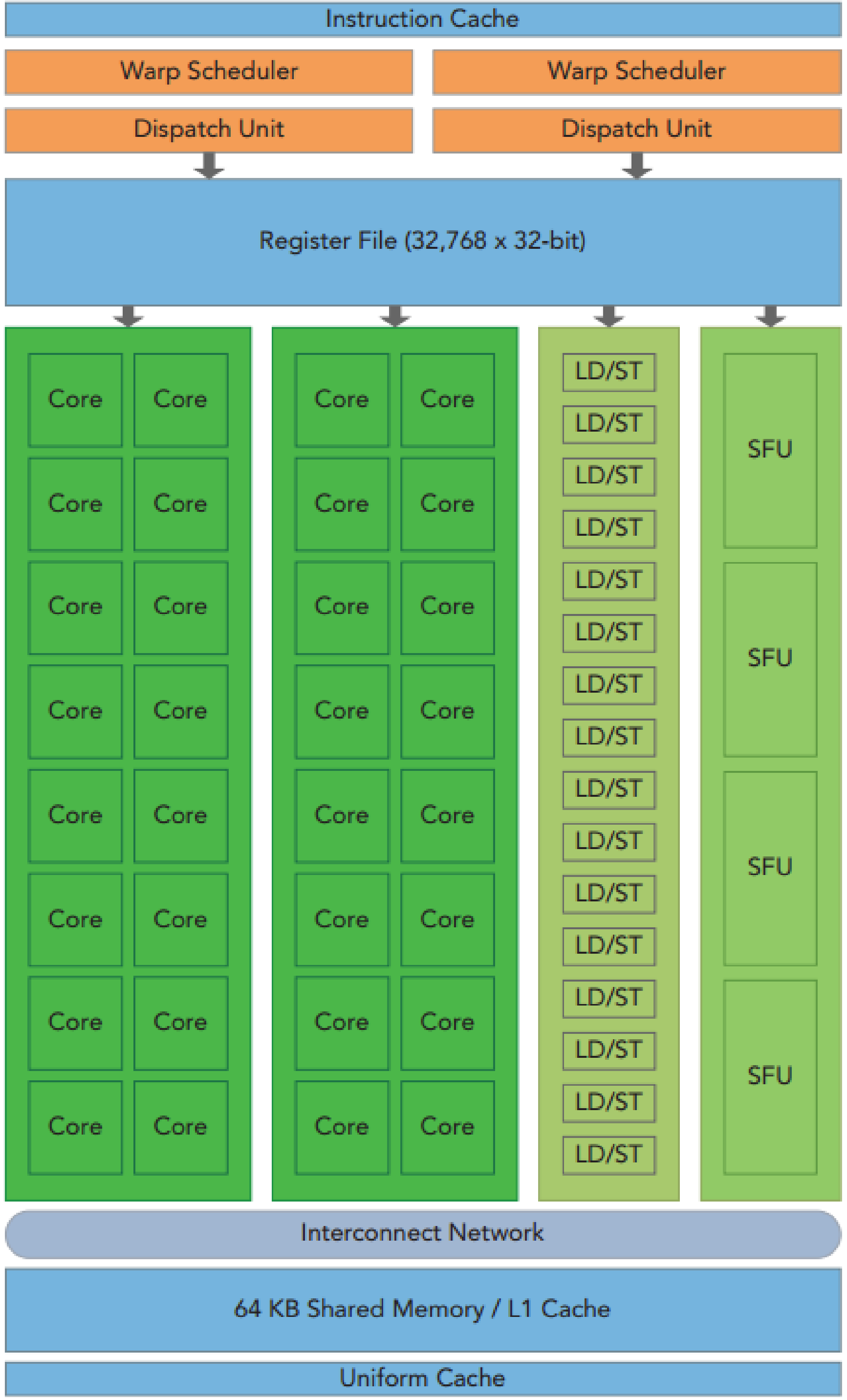
Timing Your Kernel

http://blog.csdn.net/litdaguang/article/details/50520549

Execution Model

the key components of a SM:

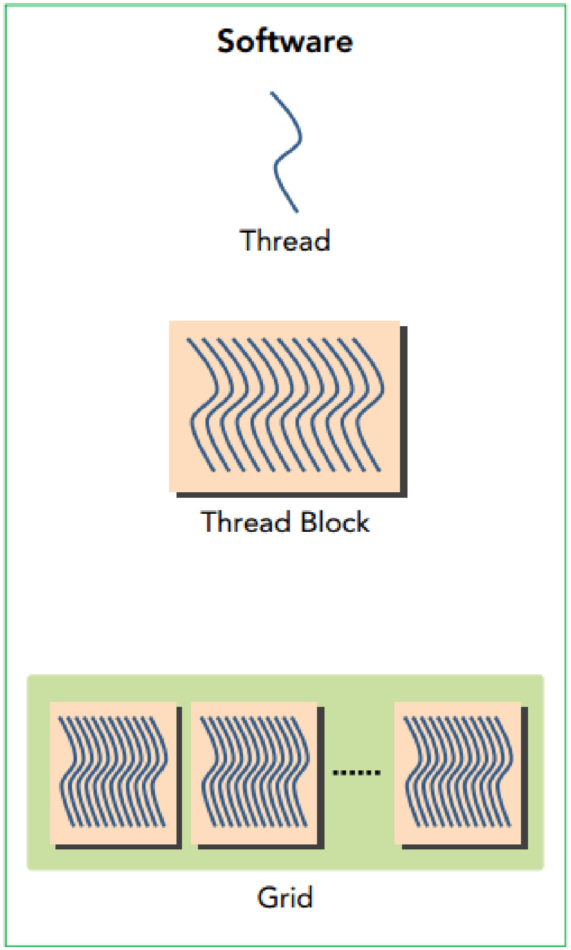
- CUDA Cores
- Shared Memory/L1 Cache
- Register File
- Load/Store Units
- Special Function Units
- Warp Scheduler

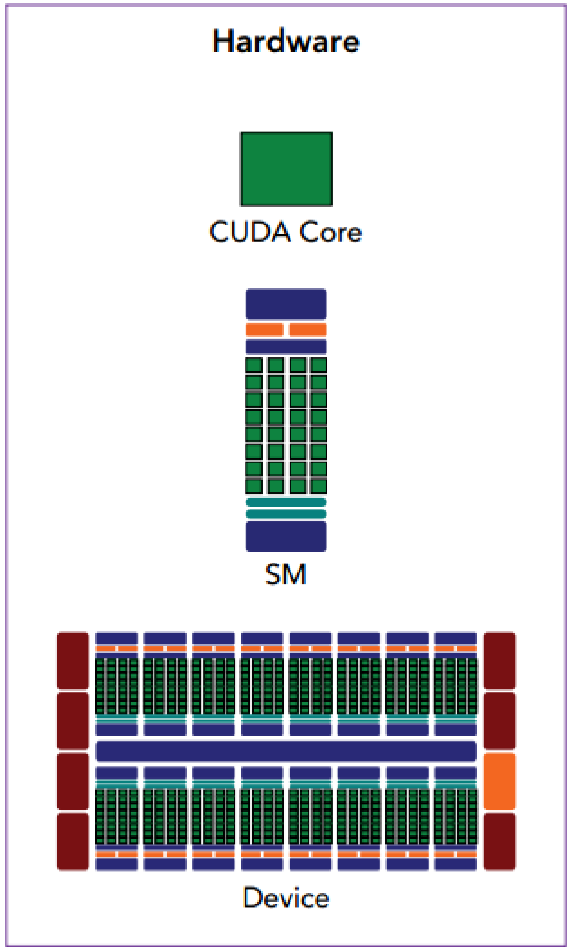


Execution Model

A thread block is scheduled on only one SM. Once a thread block is scheduled on an SM, it remains there until execution completes.

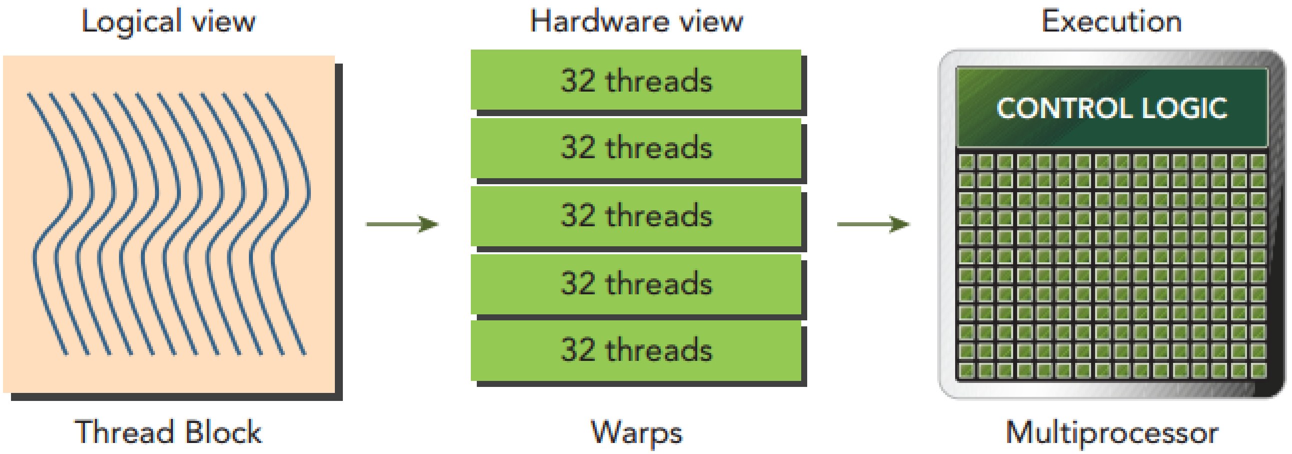
An SM can hold more than one thread block at the same time.

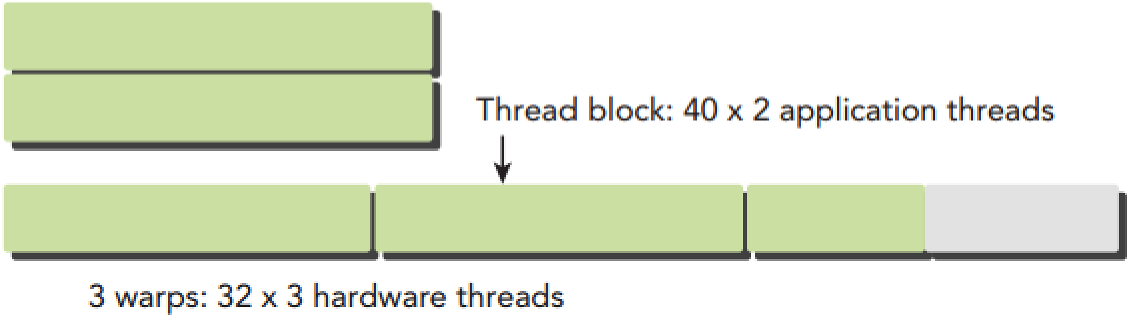




Execution Model

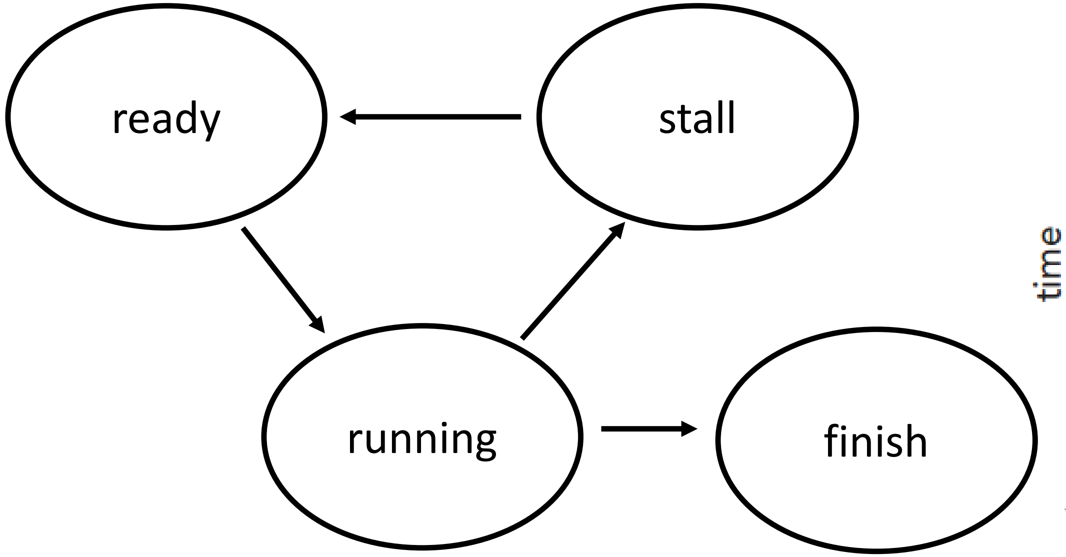
Each 32 threads in block organized into *warp*
warp is the execution unit in GPU

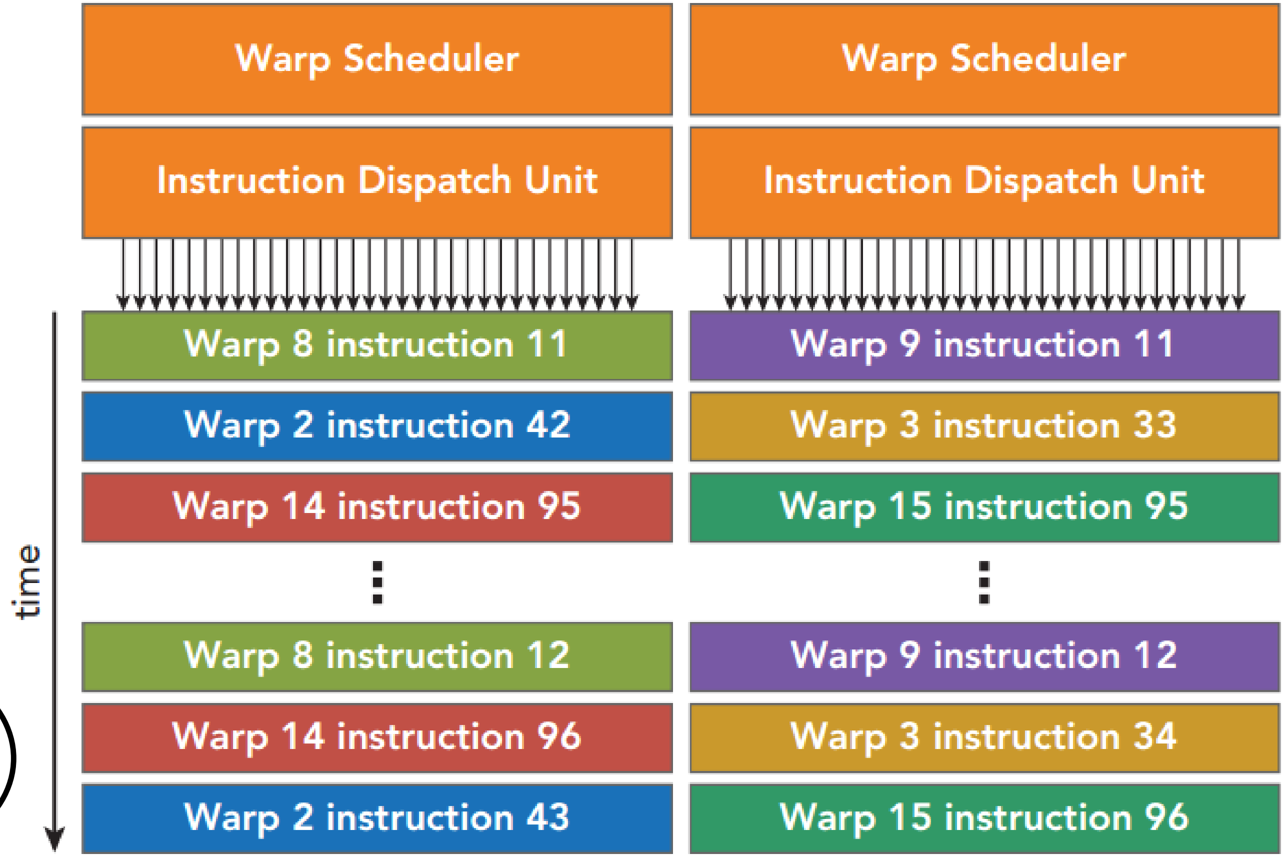




Execution Model

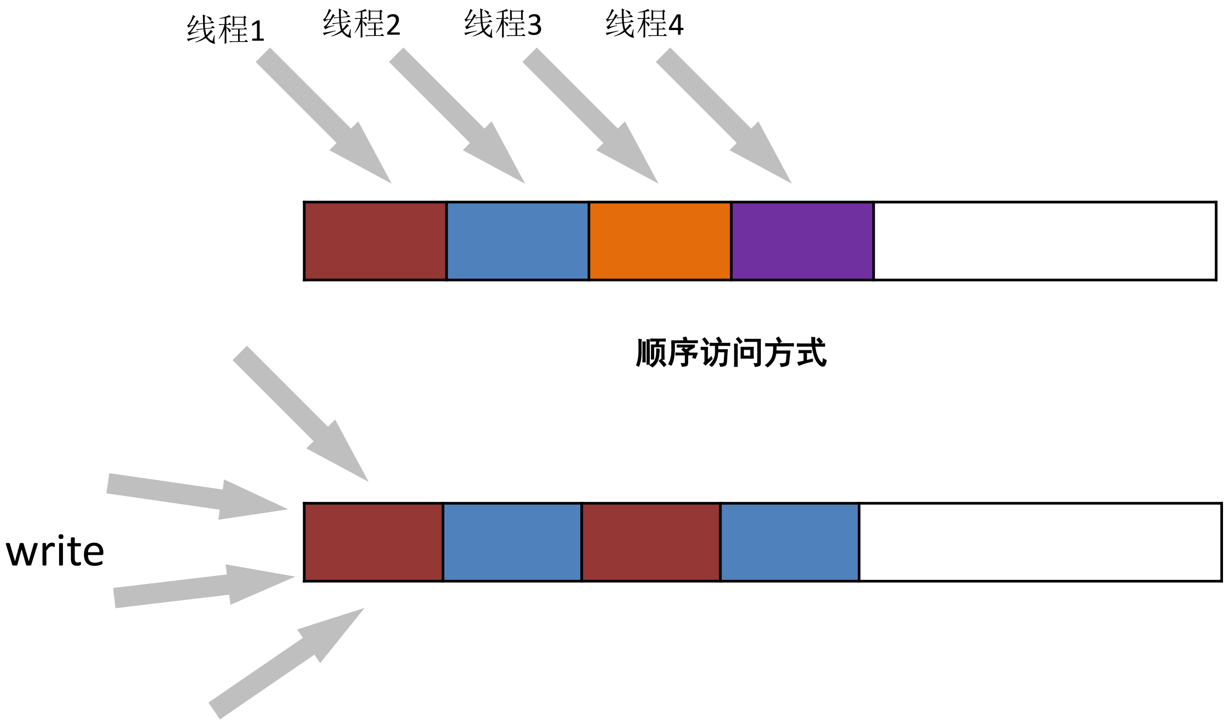
Out-of-order execution model





Execution Model

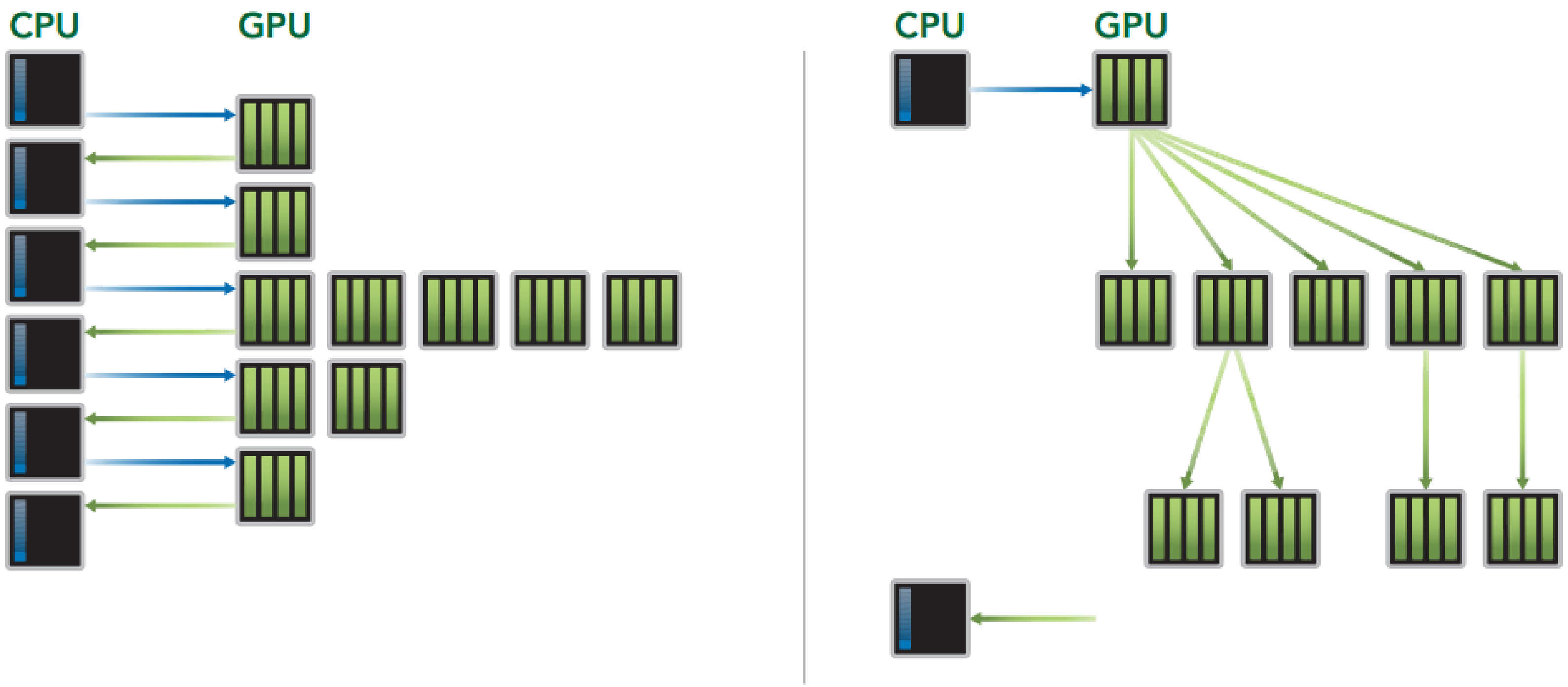
Atomic Instruction



An atomic instruction performs a mathematical operation, but does so in a single uninterruptable operation with no interference from other threads.

Execution Model

Dynamic Parallelism



Dynamic Parallelism is a new feature introduced with Kepler GPUs that allows the GPU to dynamically launch new grids.

/NVIDIA_CUDA-7.0_Samples/0_Simple/cdpSimpleQuicksort

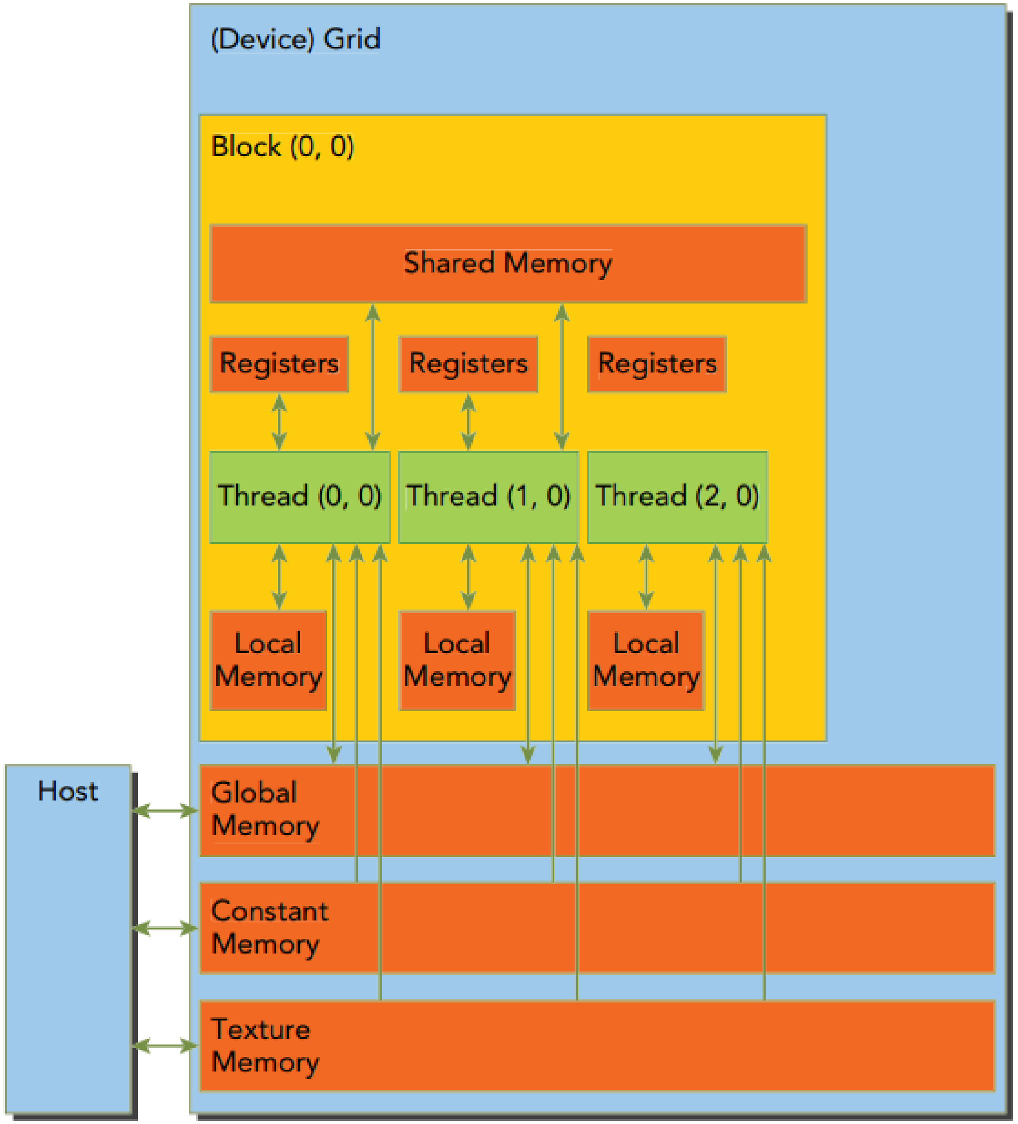
Memory Model

- ➤ Registers
- ➤ Shared memory

Onchip

- ➤ Local memory
- ➤ Constant memory
- ➤ Texture memory
- ➤ Global memory

Offchip



Memory Model

Registers

(1) the fastest memory space on a GPU

(2) there is a hardware limit of registers per thread. [Fermi:63] [Kepler:255]

```
__global__ void sumArraysZeroCopy(float *A, float *B, float *C, const int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) C[i] = A[i] + B[i];
}
```

Memory Model

Local memory

Variables in a kernel that are eligible for registers but cannot fit into the register space allocated for that kernel will spill into local memory.

- ➢ Local arrays referenced with indices whose values cannot be determined at compile-time.
- ➢ Large local structures or arrays that would consume too much register space.
- ➢ Any variable that does not fit within the kernel register limit.

```
__global__ void test(int size)
{
    int *arr1 = (int*)malloc(size);
    float arr2[200];
}
```

Memory Model

Shared memory

(1) Variables decorated with "__shared__" in a kernel are stored in shared memory

(2) Static shared memory and dynamic shared memory

(3) Each SM has a limited amount of shared memory. [48K]

(4) Shared memory shares its lifetime with a thread block. [inter-thread communication]

(5) The L1 cache and shared memory for an SM use the same 64 KB of on-chip memory, which is statically partitioned but can be dynamically configured at runtime.

```
cudaError_t cudaFuncSetCacheConfig(const void* func, enum cudaFuncCache  cacheConfig);
cudaFuncCachePreferNone: no preference (default)
cudaFuncCachePreferShared: prefer 48KB shared memory and 16KB L1 cache
cudaFuncCachePreferL1: prefer 48KB L1 cache and 16KB shared memory
cudaFuncCachePreferEqual: Prefer equal size of L1 cache and shared memory, both 32KB
```

Memory Model

Shared memory Example

```
static __global__
void _d1DSharedStepKer(DATA_TYPE *data1D, DATA_TYPE* dev_out, int step,
                        int am_num, int size, int copy_num_per_thread)
{
    // 获得线程索引
    int index = blockIdx.x * blockDim.x + threadIdx.x;

    if (index >= size)
        return ;

    // 将数据拷贝到共享内存中
    extern __shared__ DATA_TYPE sharedData[];
    // 一个线程拷贝 (data_size + T - 1) / T 个数据
    for (int i = 0; i < copy_num_per_thread; ++i) {
        // 计算要拷贝数据的下标
        int copt_index = index * copy_num_per_thread + i;
        if (copt_index < size)
            sharedData[copt_index] = data1D[copt_index];
    }
    __syncthreads();

    for (int i = 0; i < am_num; ++i)
        dev_out[index] += sharedData[(index + i * step) % size];
}
```

Memory Model

Shared memory Example

Shared memory

bank conflicts

```
static __global__
void _d1DSharedStepKer(DATA_TYPE *data1D, DATA_TYPE* dev_out, int step,
                        int am_num, int size, int copy_num_per_thread)
{
    // 获得线程索引
    int index = blockIdx.x * blockDim.x + threadIdx.x;

    if (index >= size)
        return ;

    // 将数据拷贝到共享内存中
    extern __shared__ DATA_TYPE sharedData[];
    // 一个线程拷贝 (data_size + T - 1) / T 个数据
    for (int i = 0; i < copy_num_per_thread; ++i) {
        // 计算要拷贝数据的下标
        int copt_index = index * copy_num_per_thread + i;
        if (copt_index < size)
            sharedData[copt_index] = data1D[copt_index];
    }
    __syncthreads();

    for (int i = 0; i < am_num; ++i)
        dev_out[index] += sharedData[(index + i * step) % size];
}
```

Memory Model

Constant memory

(1) Variables decorated with "__constant__" in a kernel are stored in constant memory

(2) Each GPU has a limited amount of shared memory. [64K]

(3) The throughput of constant memory is 4B per clock per SM. Unless an entire warp reads the same address, replays are needed.

(4) Constant memory is statically declared and visible to all kernels

```
// 全局内容的大小必须提前设置
__constant__ DATA_TYPE constant_data1D[1];
```

```
cuerrcode = cudaMemcpyToSymbol(constant_data1D, this->data1D, sizeof(DATA_TYPE) * this->size);
if (cuerrcode != cudaSuccess) {
    // 数据拷贝出错，返回错误代码前释放申请的空间
    // free(this->data1D);
    return -1;
}
```

Memory Model

HPC&IP

Texture memory

(1) Hardware interpolation

(2) Texture objects can interoperate graphics(OpenGL, DirectX)

(3) Format conversion {char, short, int} -> float

Memory Model

Global memory

Dynamic Global Memory:

- ➤ Allocate and deallocate device memory
- ➤ Transfer data between the host and device

```
#include <cuda_runtime.h>
#include <stdio.h>

int main(int argc, char **argv) {
    // set up device
    int dev = 0;
    cudaSetDevice(dev);

    // memory size
    unsigned int isize = 1<<22;
    unsigned int nbytes = isize * sizeof(float);

    // get device information
    cudaDeviceProp deviceProp;
    cudaGetDeviceProperties(&deviceProp, dev);
    printf("%s starting at ", argv[0]);
    printf("device %d: %s memory size %d nbyte %5.2fMB\n", dev,
           deviceProp.name,isize,nbytes/(1024.0f*1024.0f));

    // allocate the host memory
    float *h_a = (float *)malloc(nbytes);

    // allocate the device memory
    float *d_a;
    cudaMalloc((float **)&d_a, nbytes);

    // initialize the host memory
    for(unsigned int i=0;i<isize;i++) h_a[i] = 0.5f;

    // transfer data from the host to the device
    cudaMemcpy(d_a, h_a, nbytes, cudaMemcpyHostToDevice);

    // transfer data from the device to the host
    cudaMemcpy(h_a, d_a, nbytes, cudaMemcpyDeviceToHost);

    // free memory
    cudaFree(d_a);
    free(h_a);

    // reset device
    cudaDeviceReset();
    return EXIT_SUCCESS;
}
```

```
cudaMemcpyHostToHost
cudaMemcpyHostToDevice
cudaMemcpyDeviceToHost
cudaMemcpyDeviceToDevice
```

Memory Model

Global memory

Dynamic Global Memory:

- ➢ Allocate and deallocate device memory
- ➢ Transfer data between the host and device

1. Copy data from CPU memory to GPU memory.
2. Invoke kernels to operate on the data stored in GPU memory.
3. Copy data back from GPU memory to CPU memory.

```
#include <cuda_runtime.h>
#include <stdio.h>

int main(int argc, char **argv) {
   // set up device
   int dev = 0;
   cudaSetDevice(dev);

   // memory size
   unsigned int isize = 1<<22;
   unsigned int nbytes = isize * sizeof(float);

   // get device information
   cudaDeviceProp deviceProp;
   cudaGetDeviceProperties(&deviceProp, dev);
   printf("%s starting at ", argv[0]);
   printf("device %d: %s memory size %d nbyte %5.2fMB\n", dev,
          deviceProp.name,isize,nbytes/(1024.0f*1024.0f));

   // allocate the host memory
   float *h_a = (float *)malloc(nbytes);

   // allocate the device memory
   float *d_a;
   cudaMalloc((float **)&d_a, nbytes);

   // initialize the host memory
   for(unsigned int i=0;i<isize;i++) h_a[i] = 0.5f;

   // transfer data from the host to the device
   cudaMemcpy(d_a, h_a, nbytes, cudaMemcpyHostToDevice);

   // transfer data from the device to the host
   cudaMemcpy(h_a, d_a, nbytes, cudaMemcpyDeviceToHost);

   // free memory
   cudaFree(d_a);
   free(h_a);

   // reset device
   cudaDeviceReset();
   return EXIT_SUCCESS;
}
```

```
cudaMemcpyHostToHost
cudaMemcpyHostToDevice
cudaMemcpyDeviceToHost
cudaMemcpyDeviceToDevice
```

Memory Model

Global memory

Static Global Memory:

```
#include <cuda_runtime.h>
#include <stdio.h>

__device__ float devData;

__global__ void checkGlobalVariable() {
   // display the original value
   printf("Device: the value of the global variable is %f\n",devData);

   // alter the value
   devData +=2.0f;
}

int main(void) {
   // initialize the global variable
   float value = 3.14f;
   cudaMemcpyToSymbol(devData, &value, sizeof(float));
   printf("Host:   copied %f to the global variable\n", value);

   // invoke the kernel
   checkGlobalVariable <<<1, 1>>>();

   // copy the global variable back to the host
   cudaMemcpyFromSymbol(&value, devData, sizeof(float));
   printf("Host:   the value changed by the kernel to %f\n", value);

   cudaDeviceReset();
   return EXIT_SUCCESS;
}
```

Memory Model

- Registers
- Shared memory
- Local memory
- Constant memory
- Texture memory
- Global memory

Read-only Cache, Pinned Memory, Zero-Copy Memory, Unified virtual addressing

Shared Memory: bank conflict, data layout

Global Memory access patterns: Aligned and Coalesced Access

L1/L2 Cache

Warp shuffle instruction

Optimization

Three Bottle-Necks： Memory / Instruction / Latency

(1) Memory bandwidth bound: GDDR/L2/TEX/L1/Shared Memory bandwidth etc.

(2) Instruction throughput bound: single/double-precision/LDST/SFU throughput etc.

(3) Latency bound: No enough warp to switch in while waiting.

Optimization

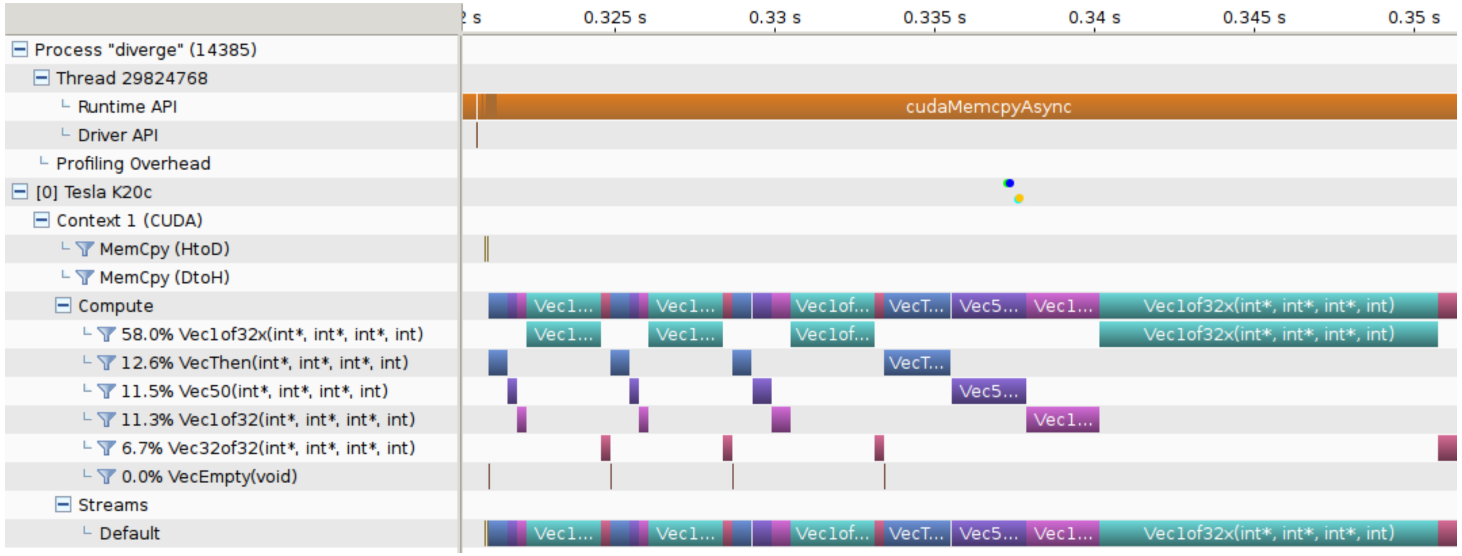
How to find bottle-necks?

http://docs.nvidia.com/cuda/profiler-users-guide/index.html

(1) NVIDIA visual profiler (NVVP)

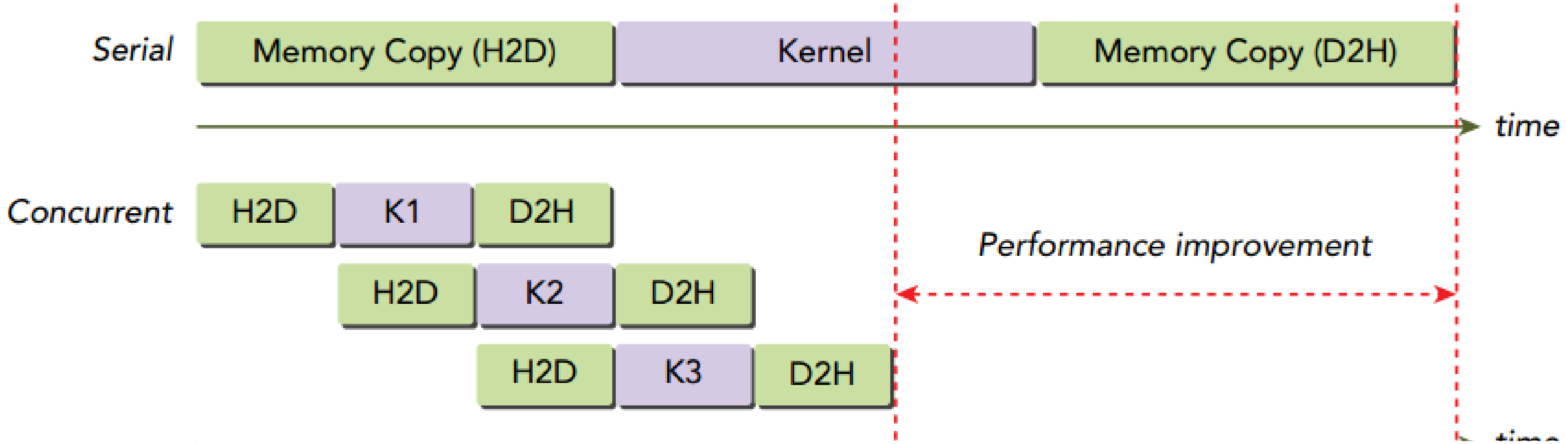
(2) NVPROF

(3) Command Line Profiler



Streams

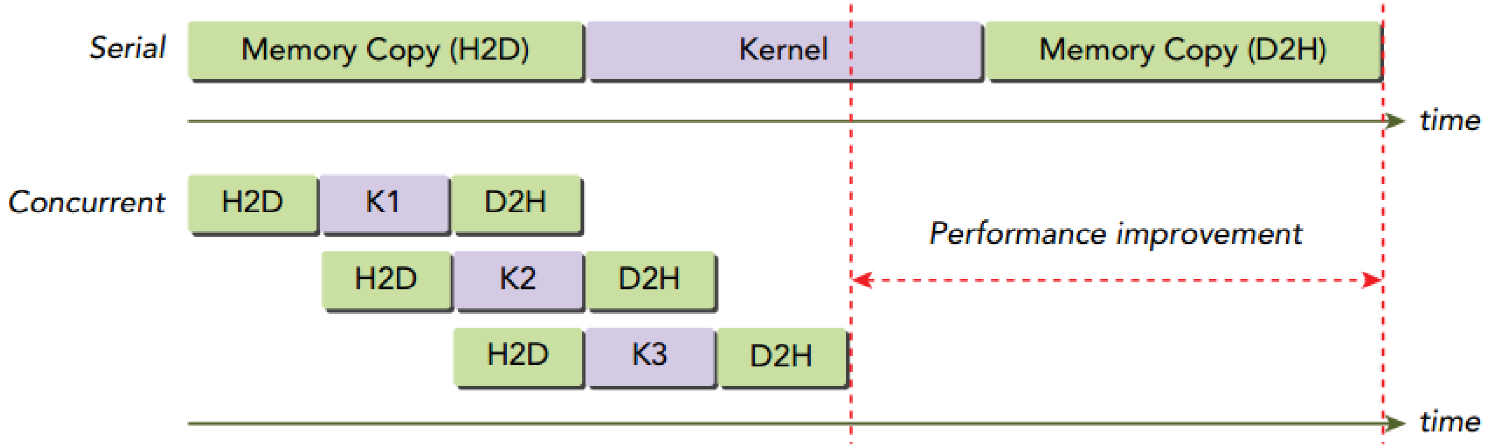
```
cudaMemcpy(..., cudaMemcpyHostToDevice);
kernel<<<grid, block>>>(...);
cudaMemcpy(..., cudaMemcpyDeviceToHost);
```




```
for (int i = 0; i < nStreams; i++) {
  int offset = i * bytesPerStream;
  cudaMemcpyAsync(&d_a[offset], &a[offset], bytePerStream, streams[i]);
  kernel<<grid, block, 0, streams[i]>>(&d_a[offset]);
  cudaMemcpyAsync(&a[offset], &d_a[offset], bytesPerStream, streams[i]);
}

for (int i = 0; i < nStreams; i++) {
    cudaStreamSynchronize(streams[i]);
}
```

Streams



Stream operations

```
cudaStream_t stream;
cudaStreamCreate(&stream);
cudaStreamDestroy(cudaStream_t stream);

cudaStreamSynchronize(cudaStream_t stream);
cudaStreamQuery(cudaStream_t stream);
```

cudaMemcpy()

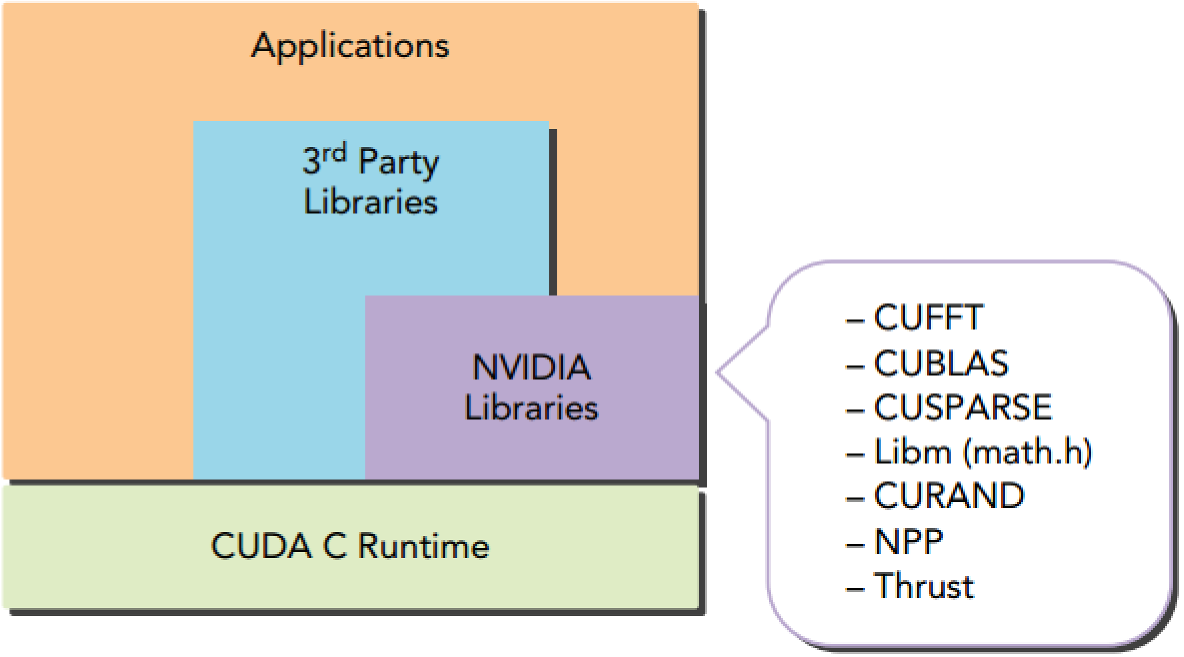
```
cudaMemcpyAsync(void* dst, const void* src, size_t count,cudaMemcpyKind kind, cudaStream_t stream = 0);
```

kernel<<< .. , .. >>>

```
kernel_name<<<grid, block, sharedMemSize, stream>>>(argument list);
```

CUDA Libraries

https://cudazone.nvidia.cn/gpu-accelerated-libraries/



CUDA Tools

- 编辑器：

sublime text + cuda snippets 插件

Nsight Eclipse Edition

- 调试器：cuda-gdb，Nsight，CUDA-MEMCHECK

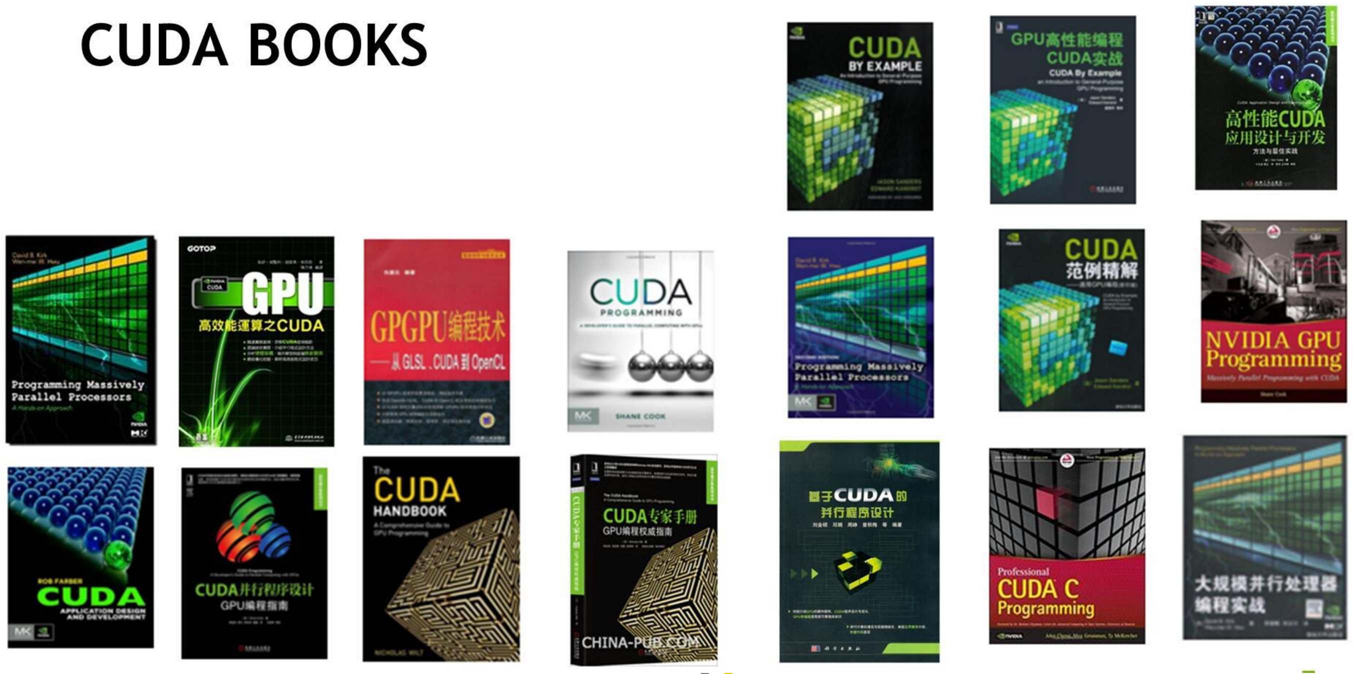
- 性能分析工具：NVIDIA Visual Profiler(NVVP)， nvprof

- cuda docs (http://docs.nvidia.com/cuda/index.html)

Experience to Learn CUDA

- 官网入门文档：<<CUDA C Programming Guide>>
- 代码：cuda sdk example, openCUDA (https://github.com/LitLeo/OpenCUDA)
- CUDA群：CUDA Professional (45157483)
- Stack Overflow (http://stackoverflow.com/)
- CUDA 书籍

CUDA BOOKS



Thanks