GPGPU Programming

M. Sato

University of Tsukuba

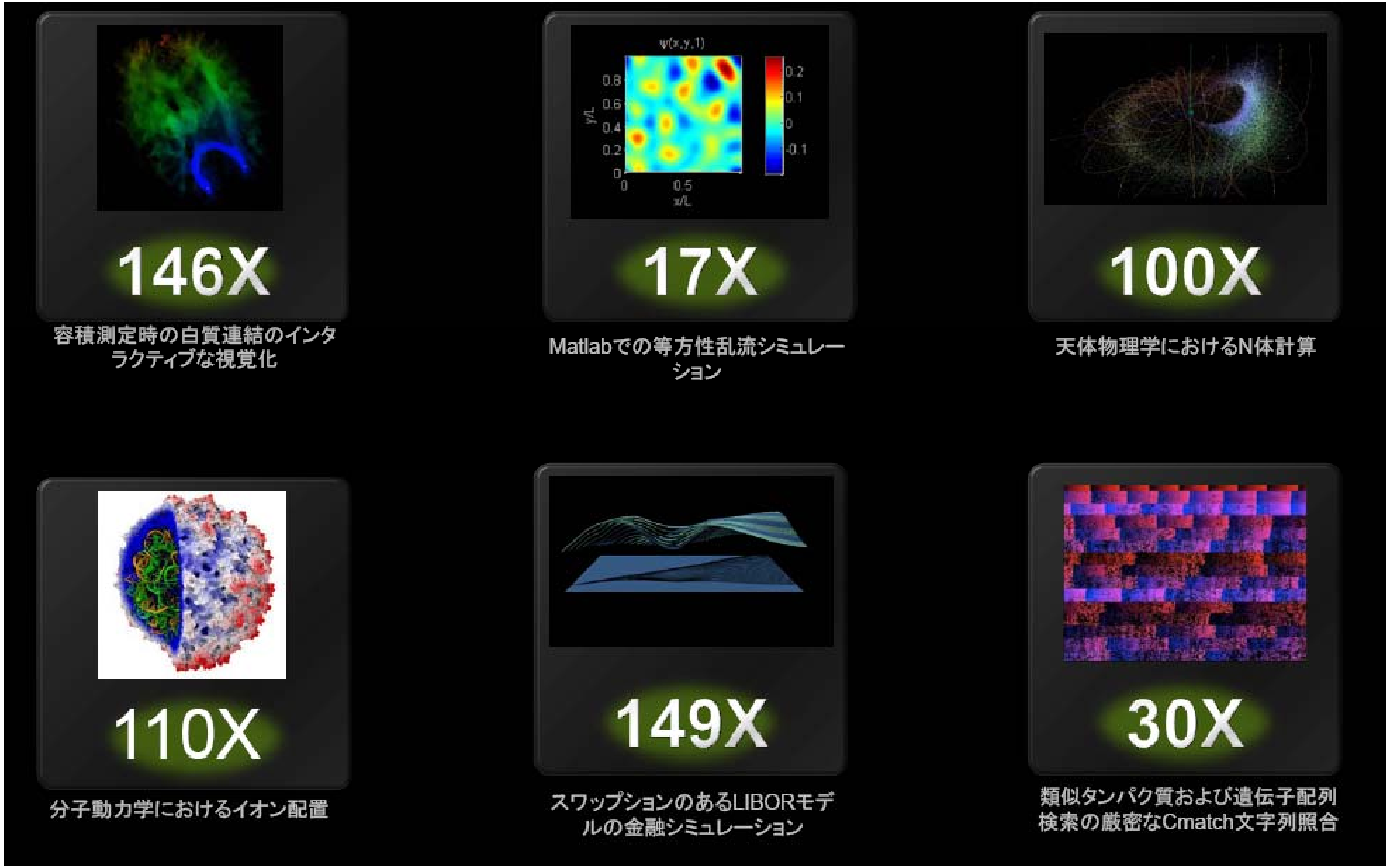
reference

- NVIDIAのCUDAの情報　Learn More about CUDA - NVIDIA
  – http://www.nvidia.co.jp/object/cuda_education_jp.html
  – 正式なマニュアルは、NVIDIA CUDA programming Guide
- わかりやすいCUDAのスライド
  – http://www.sintef.no/upload/IKT/9011/SimOslo/eVITA/2008/seland.pdf
- CUDAのコード例
  – http://tech.ckme.co.jp/cuda.shtml
- OpenCL NVIDIAのページ
  – http://www.nvidia.co.jp/object/cuda_opencl_jp.html
- 後藤弘茂のWeekly海外ニュース
  – スケーラブルに展開するNVIDIAのG80アーキテクチャ（2007年4月16日） http://pc.watch.impress.co.jp/docs/2007/0416/kaigai350.htm
  – KhronosがGDCでGPUやCell B.E.をサポートするOpenCLのデモを公開（2009年3月30日) http://pc.watch.impress.co.jp/docs/2009/0330/kaigai497.htm

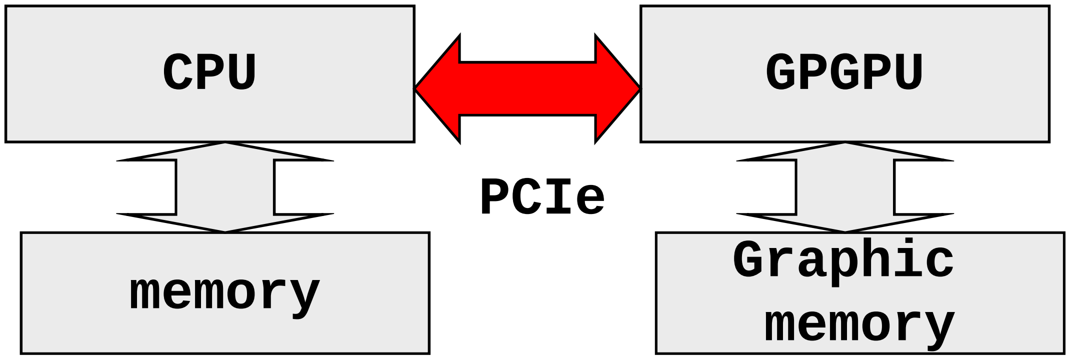
GPU Computing

- **GPGPU - General-Purpose Graphic Processing Unit**
  - **A technology to make use of GPU for general-purpose computing (scientific applications)**
- **CUDA （Compute Unified Device Architecture）**
  - **Co-designed Hardware and Software to exploit computing power of NVIDIA GPU for GP computing.**
  - **(In other words), at the moment, in order to obtain full performance of GPGPU, a program must be written in CUDA language.**
- **It is attracting many people’s interest since GPU enables great performance much more than that of CPU (even multi-core) in some scientific fields.**
- **Why GPGPU now?ーー price (cost-performance)!!!**

Applications（From NVIDIA's slides）

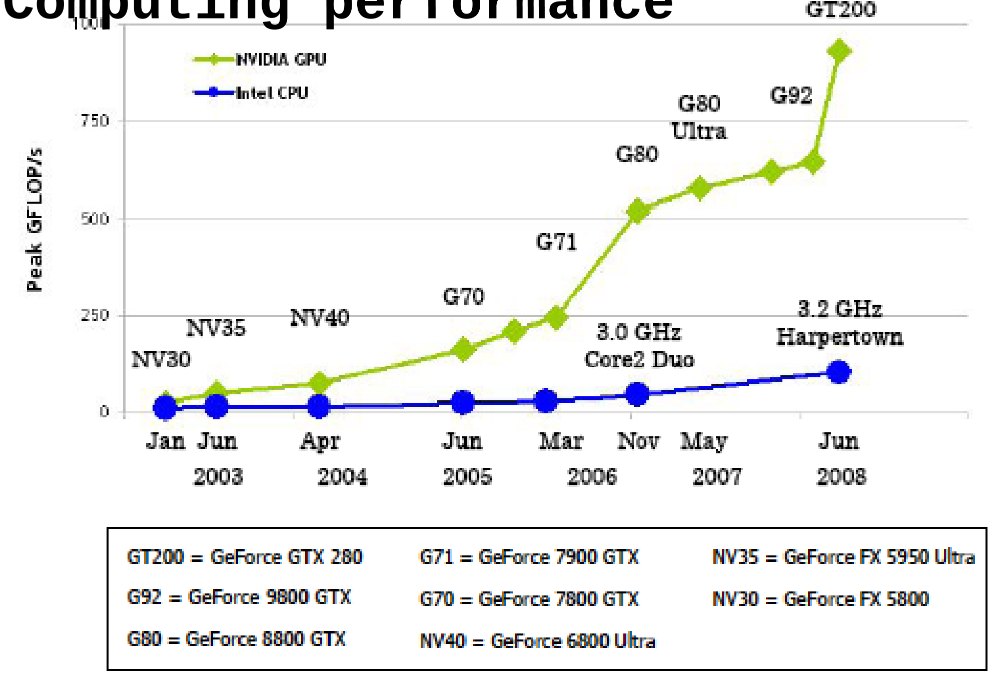


CPU vs. GPU

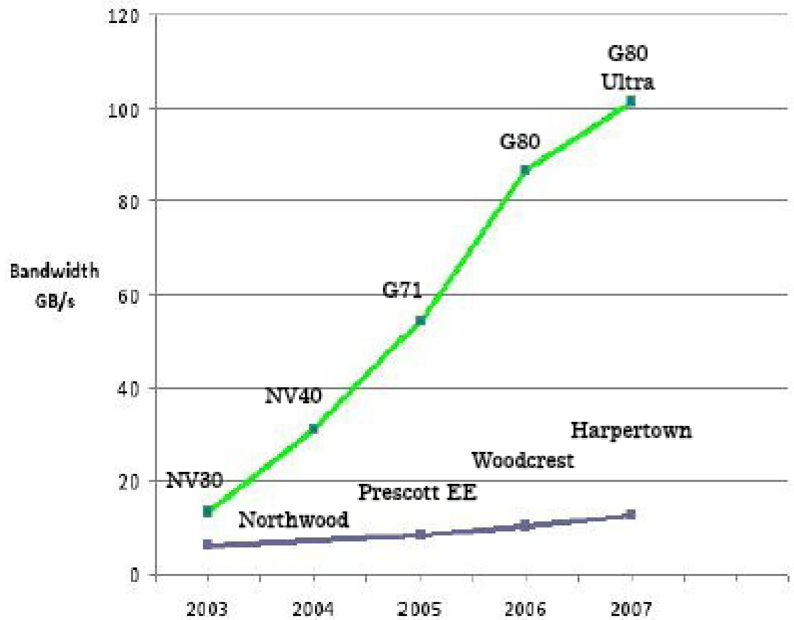


Connected via PCIexpress

Computing performance

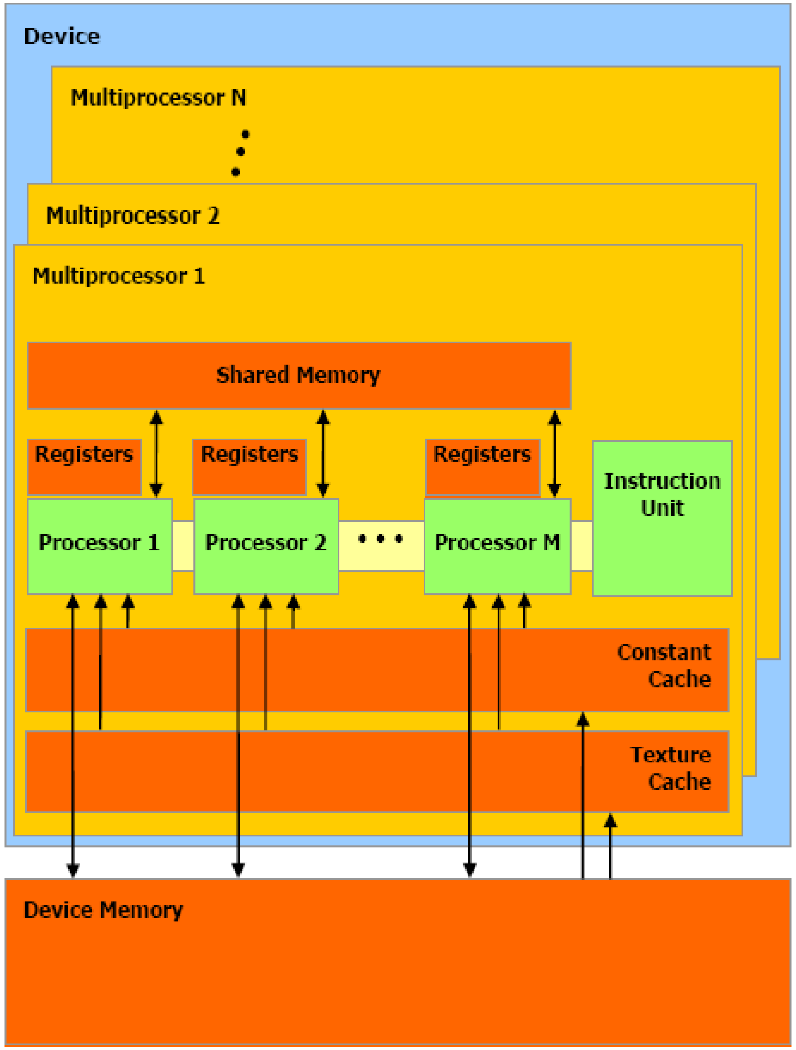


Memory bandwidth



NVIDIA GPGPU's architecture

- Many multiprocessor in a chip
  - eight Scalar Processor (SP) cores,
  - two special function units for transcendentals
  - a multithreaded instruction unit
  - on-chip shared Memory
- SIMT (single-instruction, multiple-thread).
  - The multiprocessor maps each thread to one scalar processor core, and each scalar thread executes independently with its own instruction address and register state.
  - creates, manages, schedules, and executes threads in groups of 32 parallel threads called warps.
- Complex memory hierarchy
  - Device Memory (Global Memory)
  - Shared Memory
  - Constant Cache
  - Texture Cache

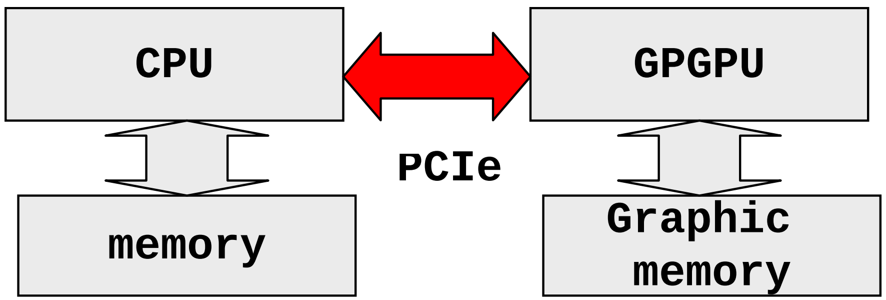


CUDA (Compute Unified Device Architecture)

- C programming language on GPUs
- Requires no knowledge of graphics APIs or GPU programming
- Access to native instructions and memory
- Easy to get started and to get real performance benefit
- Designed and developed by NVIDIA
- Requires an NVIDIA GPU (GeForce 8xxx/Tesla/Quadro)
- Stable, available (for free), documented and supported
- For both Windows and Linux

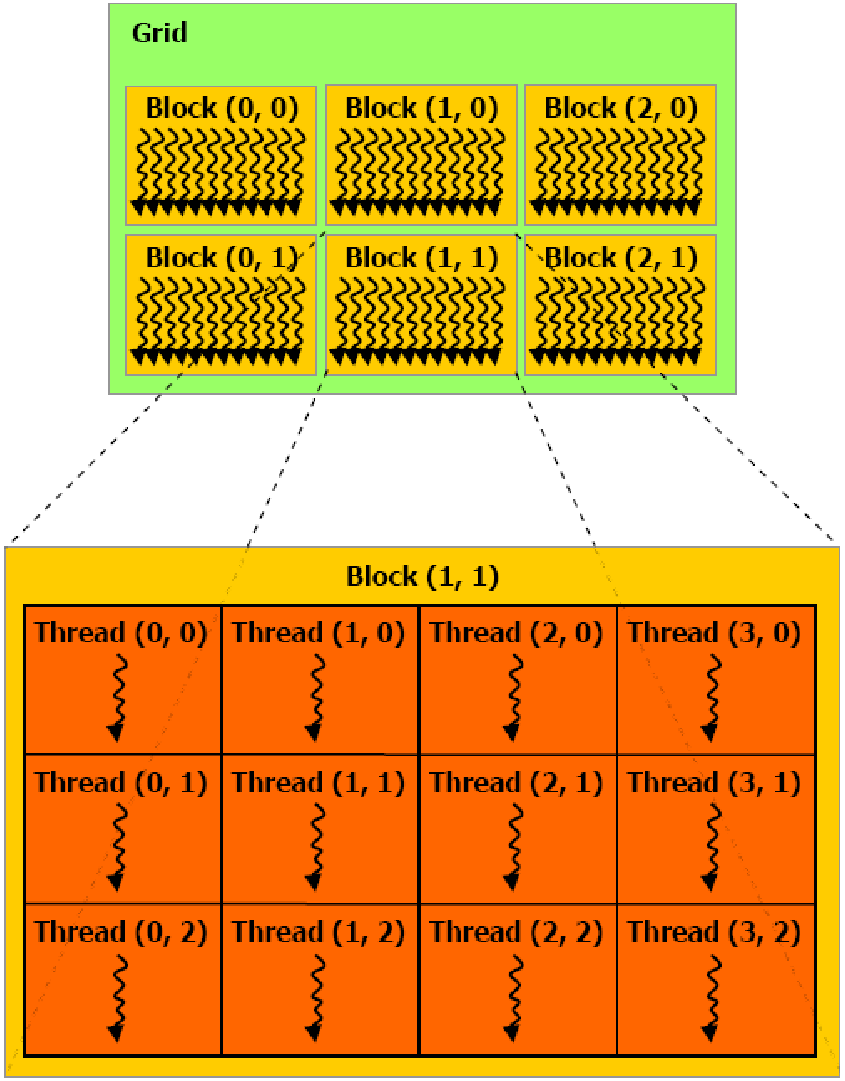
CUDA Programming model (1/2)

- GPU is programmed as a compute device working as co-processor from CPU(host).
  - Codes for data-parallel, compute intensive part are offloaded as functions to the device
  - Offload hot-spot in the program which is frequently executed on the same data
    - For example, data-parallel loop on the same data
  - Call "kernel" a code of the function compiled as a function for the device
  - Kernel is executed by multiple threads of device.
    - Only one kernel is executed on the device at a time.
  - Host (CPU) and device(GPU) has its owns memory, host memory and device memory
  - Data is copied between both memory.



CUDA Programming model (2/2)

- computational Grid is composed of multiple thread blocks
- thread block includes multiple threads
- Each thread executes kernel
  - A function executed by each thread called "kernel"
  - Kernel can be thought as one iteration in parallel loop
- computational Grid and block can have 1,2,3 dimension
- The reserved variable, blockID and threadID have ID of threads.



Example: Element-wise Matrix Add

```
void add_matrix
( float* a, float* b, float* c, int N ) {
  int index;
  for ( int i = 0; i < N; ++i )
  for ( int j = 0; j < N; ++j ) {
    index = i + j*N;
    c[index] = a[index] + b[index];
  }
}
int main() {
 add_matrix( a, b, c, N );
}
```

CPU program

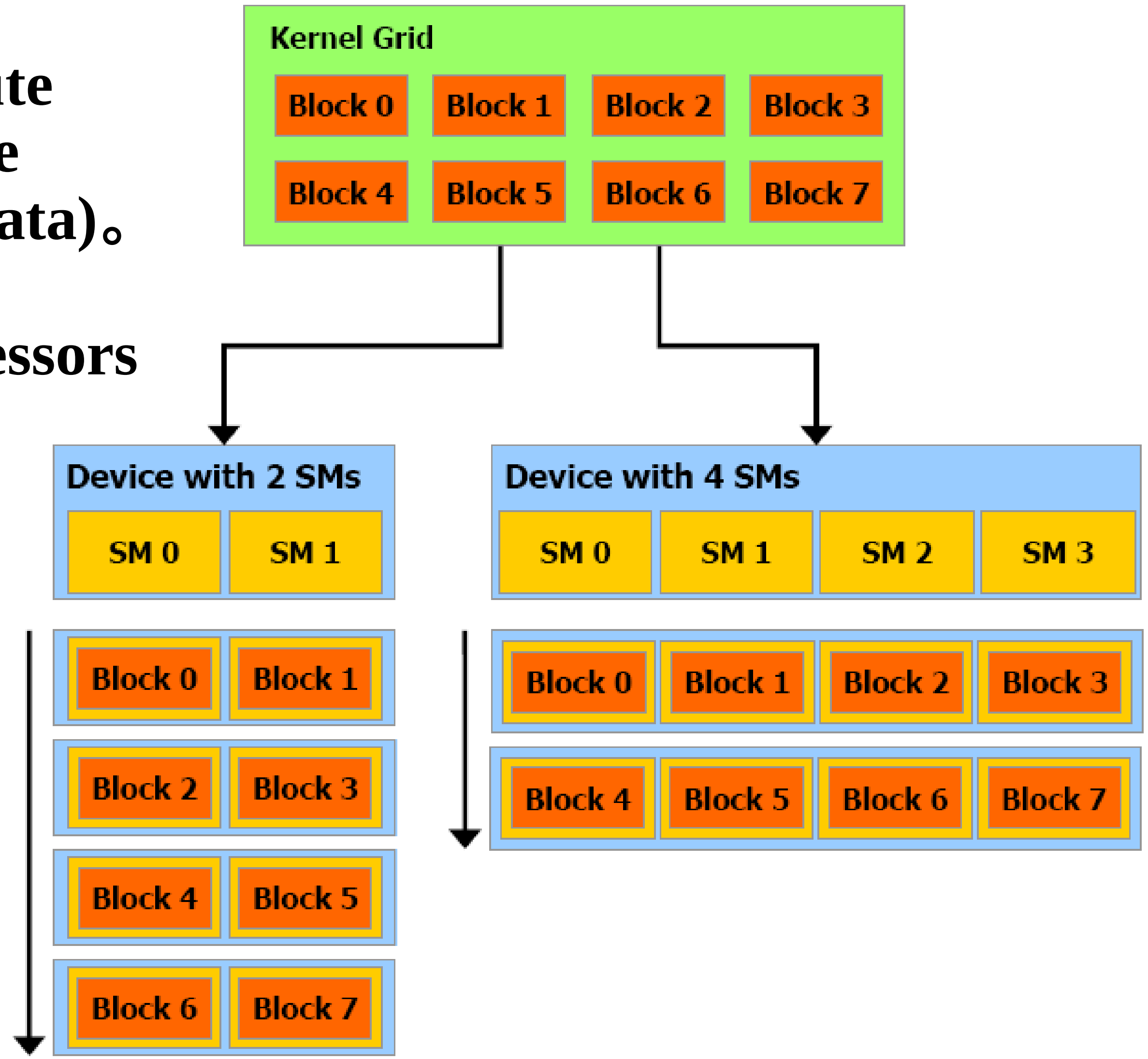
CUDA program

```
__global__ add_matrix
( float* a, float* b, float* c, int N ) {
 int i = blockIdx.x * blockDim.x + threadIdx.x;
 int j = blockIdx.y * blockDim.y + threadIdx.y;
 int index = i + j*N;
 if ( i < N && j < N )
  c[index] = a[index] + b[index];
}
int main() {
  dim3 dimBlock( blocksize, blocksize );
  dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );
  add_matrix<<<dimGrid, dimBlock>>>( a, b, c, N );
}
```

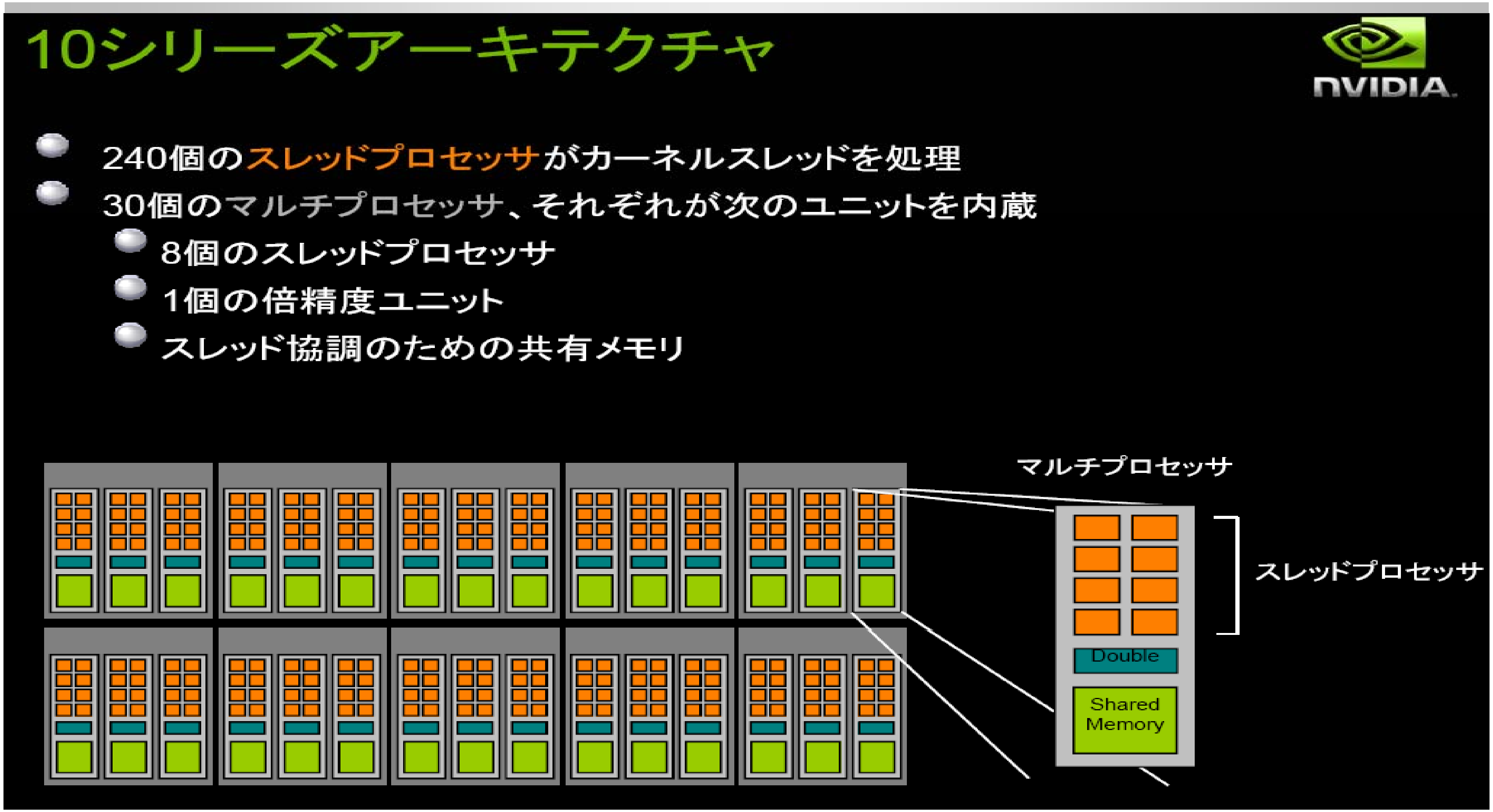
The nested for-loops are replaced with an implicit grid

How to be executed

- SM (Streaming Multiprocessor) execute blocks in SIMD (single instruction/multiple data)。
- SM consists of 8 processors



An example of GPGPU configuration



Lecture on Programming Environment

	Number of Multiprocessors (1 Multiprocessor = 8 Processors)	Compute Capability
GeForce GTX 295	2x30	1.3
GeForce GTX 285, GTX 280	30	1.3
GeForce GTX 260	24	
GeForce 9800 GX2	2x16	
GeForce GTS 250, GTS 150, 9800 GTX, 9800 GTX+, 8800 GTS 512	16	
GeForce 8800 Ultra, 8800 GTX	16	
GeForce 9800 GT, 8800 GT, GTX 280M, 9800M GTX	14	
GeForce GT 130, 9600 GSO, 8800 GS, 8800M GTX, GTX 260M, 9800M GT	12	

Tesla C1060

コア数： 240コア
プロセッサ周波数： 1.3GHz
搭載メモリ： 4GB
単精度浮動小数点演算性能： 933GFlops (ピーク)
倍精度浮動小数点演算性能： 78GFlops (ピーク)
メモリ帯域： 102GB/sec
標準電力消費量： 187.8W
浮動小数点演算： IEEE 754 単精度/倍精度
ホスト接続： PCI Express x16 (PCI-E2.0対応)

Tesla S1070	4x30	1.3
Tesla C1060	30	1.3
Tesla S870	4x16	1.0
Tesla D870	2x16	1.0
Tesla C870	16	1.0
Quadro Plex 2200 D2	2x30	1.3
Quadro Plex 2100 D4	4x14	1.1
Quadro Plex 2100 Model S4	4x16	1.0

Invoke (Launching) Kernel

- Host processor invoke the execution of kernel in this form similar to function call:

```
kernel<<<dim3 grid, dim3 block, shmem_size>>>(…)
```

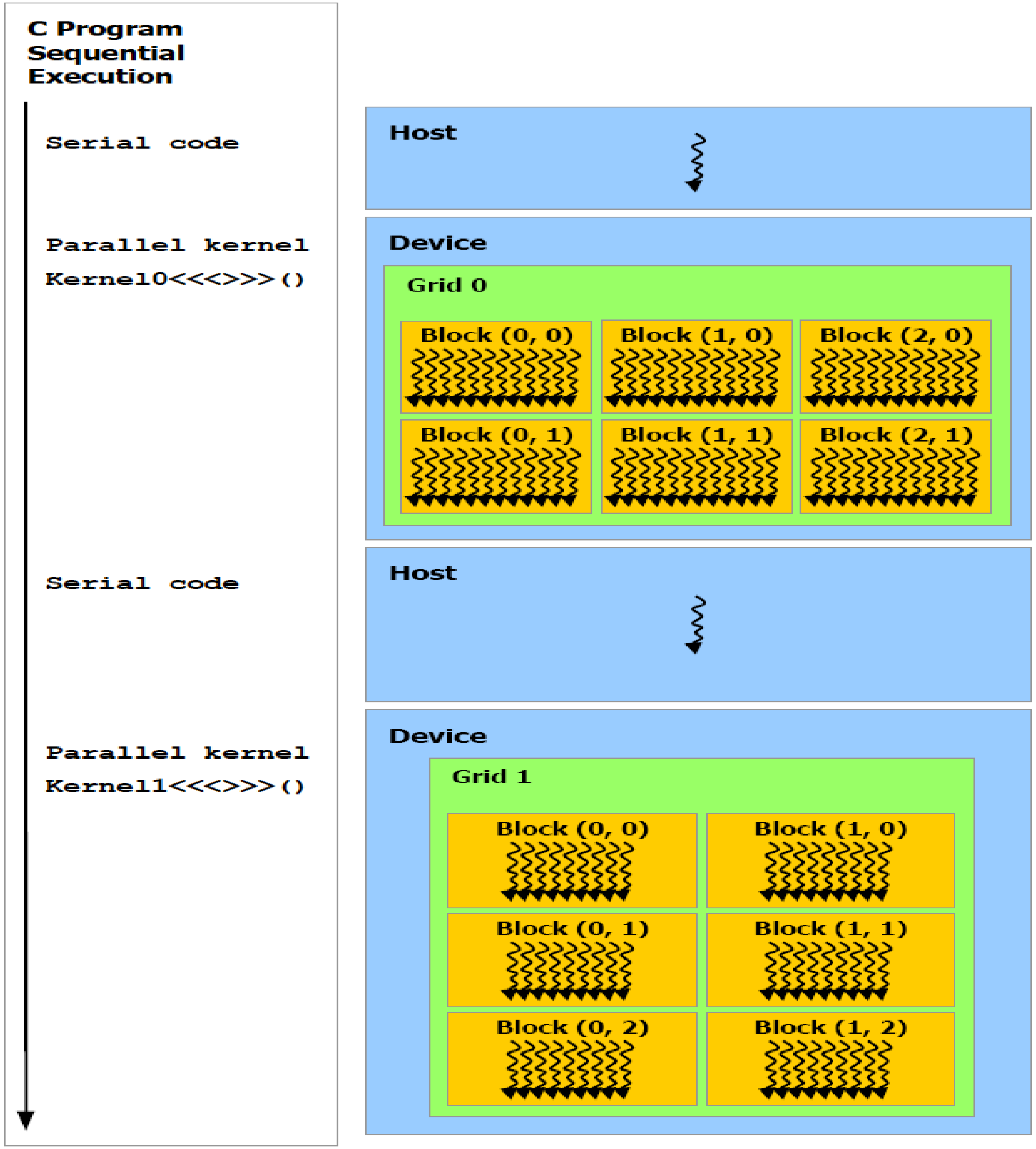
- Execution Configuation (“<<< >>>”)
  - Dimension of computational grid : x and y
  - Dimension of thread block: x、y、z

```
dim3 grid(16 16);
dim3 block(16,16);
kernel<<<grid, block>>>(...);
kernel<<<32, 512>>>(...);
```

CUDA kernel and thread

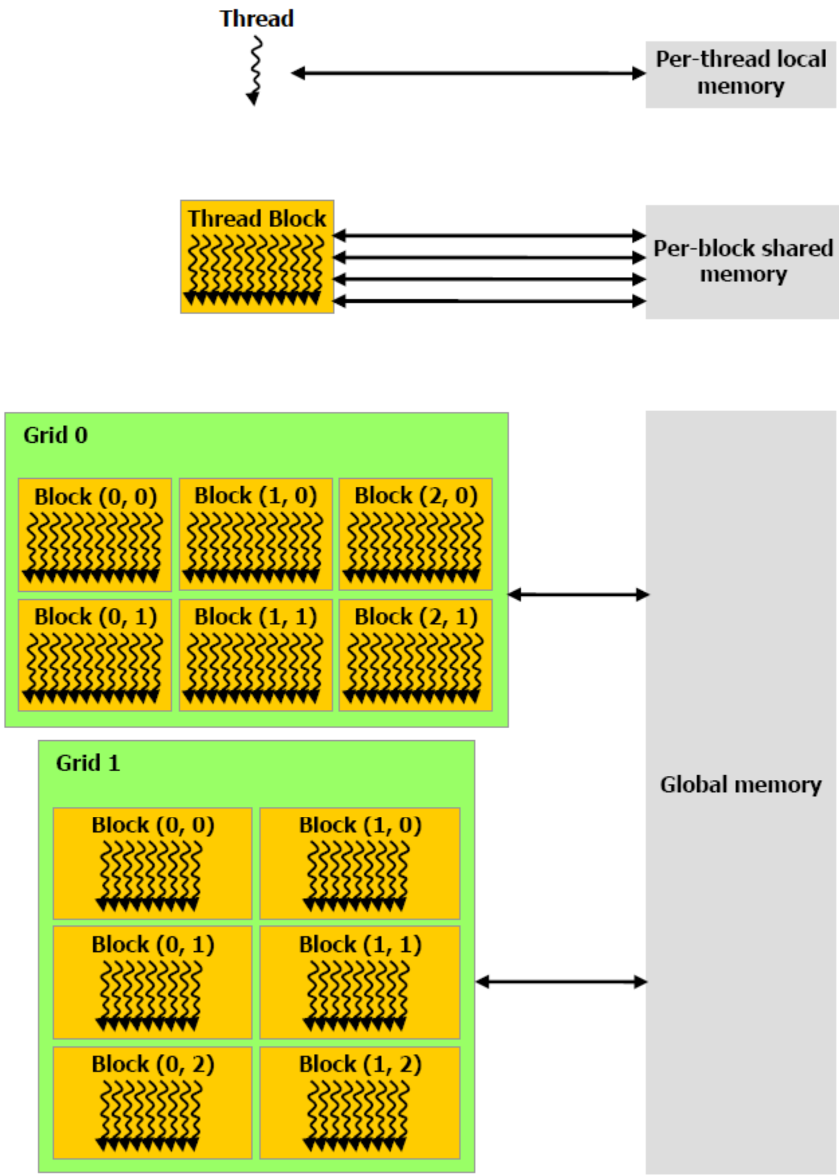
- Parallel part of applications are executed as a kernel of CUDA on the device
  - One kernel is executed at a time
  - Many threads execute kernel function in parallel.
- Difference between CUDA thread and CPU thread
  - CUDA thread is a very light-weight thread
    - Overhead of thread creation is very small
    - Thread switching is also very fast since it is supported by hardware.
  - CUDA exploit its performance and efficient execution by a thousands of threads.
    - Conventional Multicore supports only a few threads (by software)

Execution of CPU Code and Kernel code by Device



Grid, Block, thread and Memory hierarchy

- Thread can access local memory (per-thread)
- Thread can access "shared memory" on chip, which is attached for each thread block (SM).
- Thread in Computational Grid access and share a global memory.



Memory management (1/2)

- **CPU and GPU have different memory space.**
- **Hosts（CPU）manages device (GPU）memory**

Allocation and Deallocation of GPU memory

  - `cudaMalloc(void ** pointer, size_t nbytes)`
  - `cudaMemset(void * pointer, int value, size_t count)`
  - `cudaFree(void* pointer)`

```
int n = 1024;
int nbytes = 1024*sizeof(int);
int *d_a = 0;
cudaMalloc( (void**)&d_a nbytes );
cudaMemset( d_a, 0, nbytes);
cudaFree(d_a);
```

Memory management (2/2)

- **Data copy operation between CPU and device**
  - `cudaMemcpy(void *dst, void *src, size_t nbytes, enum cudaMemcpyKind direction);`
    - **Direction specifies how to copy from src to dst , see below**
    - **Block a caller of CPU thread (execution) until the memory transfer completes.**
    - **Copy operation starts after previous CUDA calls.**
  - `enum cudaMemcpyKind`
    - `cudaMemcpyHostToDevice`
    - `cudaMemcpyDeviceToHost`
    - `cudaMemcpyDeviceToDevice`

Executing Code on the GPU

- Kernels are C functions with some restrictions
  - Can only access GPU memory
  - Must have void return type
  - No variable number of arguments ("varargs")
  - Not recursive
  - No static variables
  - Function arguments

- Function arguments automatically copied from CPU to GPU memory

Function Qualifiers

- `__global__` : invoked from within host (CPU) code,
  cannot be called from device (GPU) code must return void
- `__device__` : called from other GPU functions,
  cannot be called from host (CPU) code
- `__host__` : can only be executed by CPU, called from host
- `__host__` and `__device__` can be combined.
  - Sample use: overloading operators
  - Compiler will generate both CPU and GPU code

CUDA Built-in Device Variables

- **`__global__` and `__device__` functions have access to these automatically defined variables**
  - `dim3 gridDim;`
    - **Dimensions of the grid in blocks (at most 2D)**
  - `dim3 blockDim;`
    - **Dimensions of the block in threads**
  - `dim3 blockIdx;`
    - **Block index within the grid**
  - `dim3 threadIdx;`
    - **Thread index within the block**

A simple example

```
__global__ void minimal( int* d_a)
{
      *d_a = 13;
}
```

```
__global__ void assign( int* d_a, int value)
{
  int idx = blockDim.x * blockIdx.x + threadIdx.x;
  d_a[idx] = value;
}
```

A simple example

```
__global__ void assign2D(int* d_a, int w, int h, int value)
{
   int iy = blockDim.y * blockIdx.y + threadIdx.y;
   int ix = blockDim.x * blockIdx.x + threadIdx.x;
   int idx = iy * w + ix;
   d_a[idx] = value;
}
...
assign2D<<<dim3(64, 64), dim3(16, 16)>>>(...);
```

Example code to increment array elements

CPU code

```
void inc_cpu(int*a, intN)
{
  int idx;
  for (idx =0;idx<N;idx++)
    a[idx]=a[idx] + 1;
}

voidmain()
{
 ...
 inc_cpu(a, N);
}
```

CUDA codes

```
__global__ void
  inc_gpu(int*a_d, intN){
  int idx = blockIdx.x* blockDim.x
           +threadIdx.x;
  if (idx < N)
    a_d[idx] = a_d[idx] + 1;
}
void main()
{
   …
   dim3dimBlock (blocksize);
   dim3dimGrid(ceil(N/
              (float)blocksize));
   inc_gpu<<<dimGrid,
       dimBlock>>>(a_d, N);
}
```

Example (host-side program)

```
// allocate host memory
int numBytes = N * sizeof(float)
float* h_A = (float*) malloc(numBytes);

// allocate device memory
// float* d_A = 0;
cudaMalloc((void**)&d_A, numbytes);

// Copy data from host to device
cudaMemcpy(d_A, h_A, numBytes, cudaMemcpyHostToDevice);

// Execute kernel
increment_gpu<<< N/blockSize, blockSize>>>(d_A, b);

// copy back data from device to host
cudaMemcpy(h_A, d_A, numBytes, cudaMemcpyDeviceToHost);

// Free device memory
cudaFree(d_A);
```

```
int main() {
  float *a = new float[N*N];
  float *b = new float[N*N];
  float *c = new float[N*N];

  for ( int i = 0; i < N*N; ++i ) {
   a[i] = 1.0f; b[i] = 3.5f; }

  float *ad, *bd, *cd;
  const int size = N*N*sizeof(float);
  cudaMalloc( (void**)&ad, size );
  cudaMalloc( (void**)&bd, size );
  cudaMalloc( (void**)&cd, size );

  cudaMemcpy( ad, a, size, cudaMemcpyHostToDevice );
  cudaMemcpy( bd, b, size, cudaMemcpyHostToDevice );

  dim3 dimBlock( blocksize, blocksize );
  dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );
  add_matrix<<<dimGrid, dimBlock>>>( ad, bd, cd, N );

  cudaMemcpy( c, cd, size, cudaMemcpyDeviceToHost );

  cudaFree( ad ); cudaFree( bd ); cudaFree( cd );
  delete[] a; delete[] b; delete[] c;
  return EXIT_SUCCESS;
}
```

CUDA Qualifiers for variable

- `_device_`
  - Allocated in device global memory（Large, high-latency, no cache）
  - Allocated by cudaMallocで（__device__ is default）
  - Access by every thread.
  - extent: during execution of application

- `__shared__`
  - Stored in on-chip “shared memory” (SRAM, low latency)
  - Allocated by execution configuration or at compile time
  - Accessible by all threads in the same thread block

- **Unqualified variables**
  - Scalars and built-in vector types are stored in registers
  - Arrays may be in registers or local memory *(registers are not addressable)*

How to use/specify shared memory

Compile time

```
__global__ void kernel(…)
{
…
__shared__ float sData[256];
…
}
int main(void)
{
…
kernel<<<nBlocks,blockSize>>>(…);
}
```

Invocation time

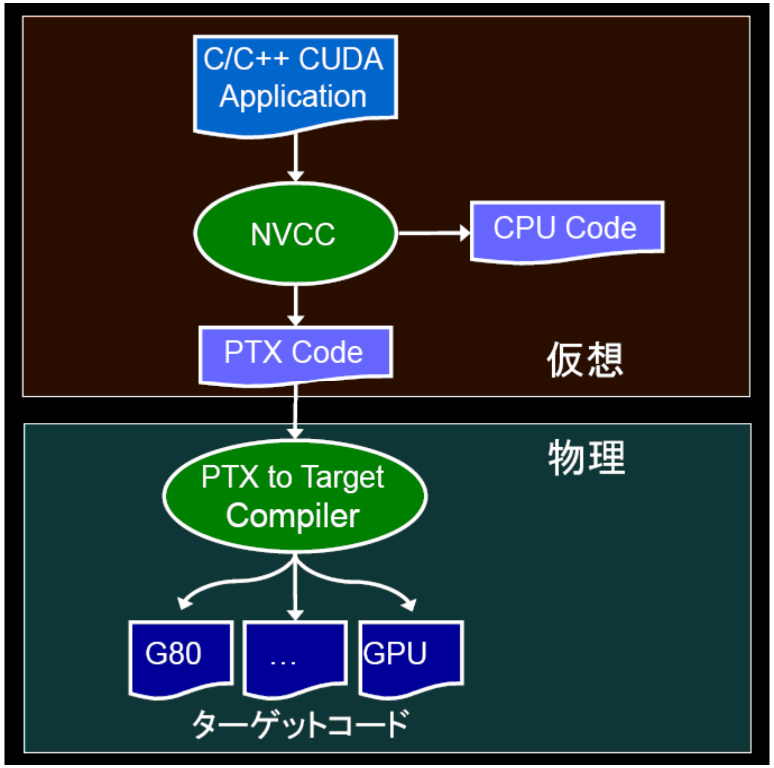
```
__global__ void kernel(…)
{
   …
   extern __shared__ float sData[];
   …
}

int main(void)
{
   …
   smBytes =
   blockSize*sizeof(float);
   kernel<<<nBlocks, blockSize,
                     smBytes>>>(…);
   …
}
```

GPU Thread Synchronization

- `void __syncthreads();`
  - Synchronizes all threads in a block
  - Generates barrier synchronization instruction
  - No thread can pass this barrier until all threads in the block reach it
  - Used to avoid RAW / WAR / WAW hazards when accessing shared memory
- Allowed in conditional code only if the conditional is uniform across the entire thread block
- Synchronization between blocks is not supported
  - Done by host-side

Compiler

- C Source program with CUDA is compiled by nvcc.
- Nvcc is a ccomile-driver:
  - Execute required tools and udacc、g++、cl
- Nvcc generates following codes:
  - C object code（CPU code）
  - PTX code for GPU
  - Glue code to call GPU from CPU
- Objects required to execute CUDA program
  - CUDA core library（cuda）
  - CUDA runtime library（cudart）



Optimization of GPU Programming

- **Maximize parallel using GPGPU**

- **Optimize/ avoid memory access to global memory**
  - Rather than storing data, re-computation may be cheaper in some cases
  - Coalescing memory access
  - Use cache in recent NVIDIA GPGPU

- **Optimize/avoid communication between CPU(host) and GPU (Device)**
  - Communication through PCI Express is expensive
  - Re-computing (redundant computing) may be cheaper than communications.

Optimization of Memory access

- **Coalescing global memory access**
  - Combine memory access to contiguous area
- **Make use of shared memory**
  - Much faster than global memory (several x 100 times faster)
    - On-chip Memory
    - Low latency
  - Threads in block share the memory.
  - All threads can share the data computed by other threads.
  - To load shared memory from global memory, coalesce the memory and use them
- **Use cache (shared memory) as in conventional CPU**
  - Recent GPGPU has a cache at the same level of shared memory

How to make use of different kinds of memory

- **Constant memory:**
  - Quite small, < 20K
  - As fast as register access if all threads in a warp access the same location
- **Texture memory:**
  - Spatially cached
  - Optimized for 2D locality
  - Neighboring threads should read neighboring addresses
  - No need to think about coalescing
- **Constraint:**
  - These memories can only be updated from the CPU

Access to Global memory

- 4 cycles to issue on memory fetch
- but 400-600 cycles of latency
  - The equivalent of 100 MADs
- Likely to be a performance bottleneck
- Order of magnitude speedups possible
  - Coalesce memory access　（結合メモリアクセス）
- Use shared memory to re-order non-coalesced addressing　（共有メモリの利用）

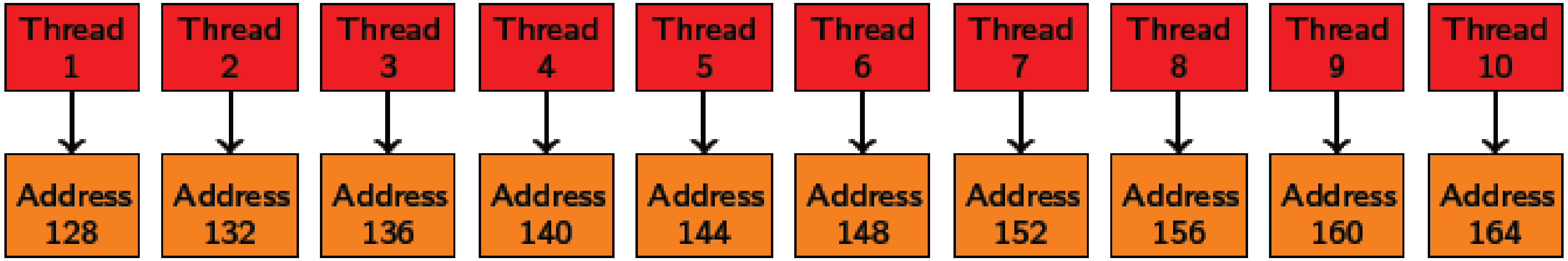
Coalesced Memory Access

To exploit performance, global memory access should be coalesced (combined).

- **A half warp（16t hread）memory access is colaesced.**
- **Contiguous memory access**
  - **64 bytes – each tread reads a single word（int、floatなど）**
  - **128bytes- each tread reads a double word （int2、float2など）**
  - **256バイト- each tread reads a quad word （int4、float4など）**
  - **Float3 is not aligned！！！**

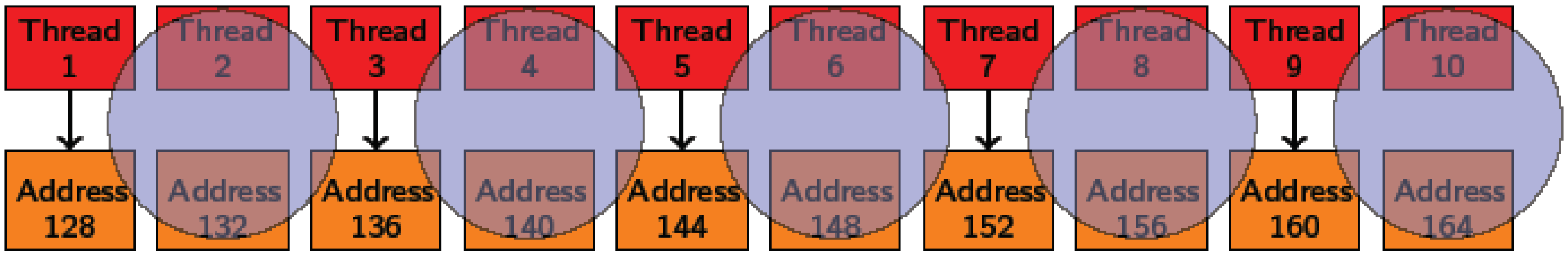
- **その他の制限**
  - **The start address of the contiguous area (Warp base address (WBA)) must be aligned the boundary of multiple of 数16*sizeof(type)**
  - **The k-th thread in half warp must access the k-th element of the block**
  - **All threads in half warp may not be access.**

Coalesced Memory Access



Coalesced memory access:

Thread k accesses $\texttt{WBA} + k$



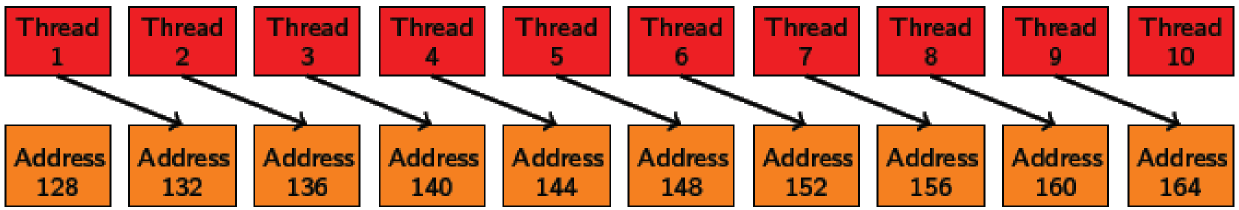
Coalesced memory access:

Thread k accesses $\texttt{WBA} + k$

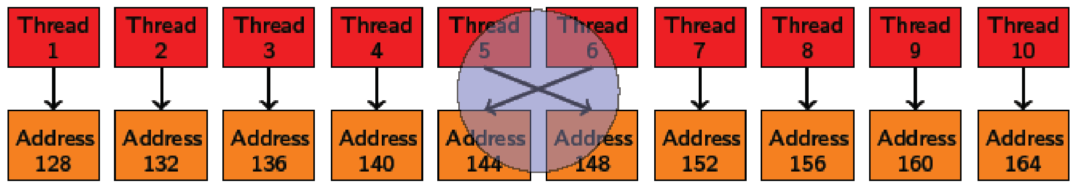
Not all threads need to participate

http://www.sintef.no/upload/IKT/9011/SimOslo/eVITA/2008/seland.pdf

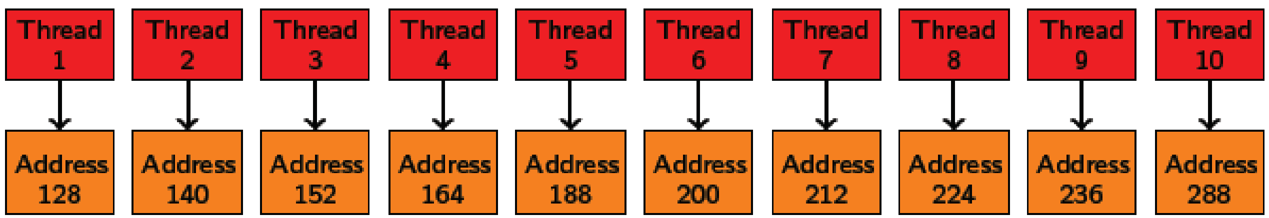
Case not coalesced



Non-Coalesced memory access:
Misaligned starting address



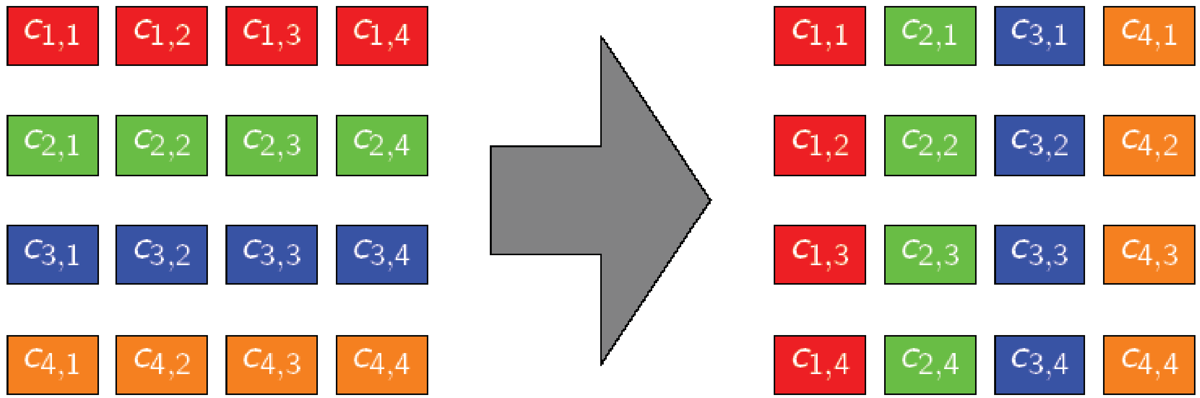
Non-Coalesced memory access:
Non-sequential access



Non-Coalesced memory access:
Wrong size of type

http://www.sintef.no/upload/IKT/9011/SimOslo/eVITA/2008/seland.pdf

Example of memory optimization : Matrix Transpose

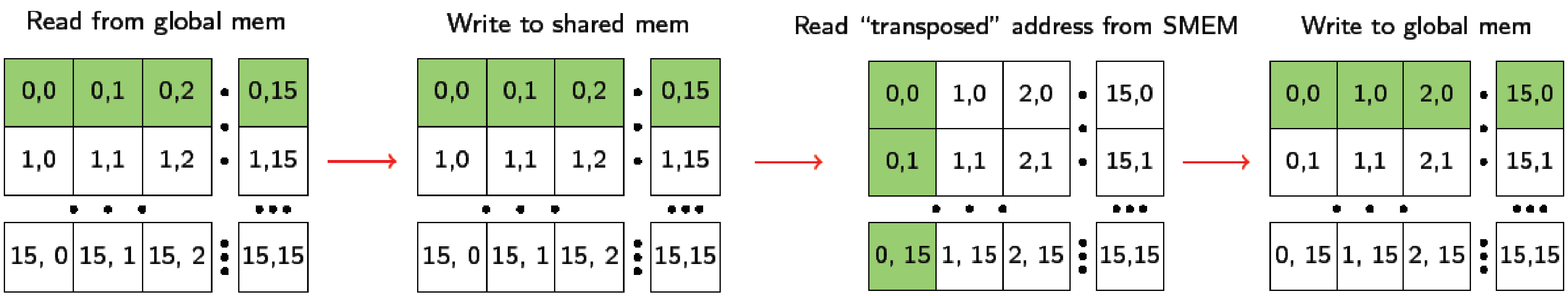



```
__global__ void
transpose_naive( float *out, float *in, int w, int h ) {
   unsigned int xIdx = blockDim.x * blockIdx.x + threadIdx.x;
   unsigned int yIdx = blockDim.y * blockIdx.y + threadIdx.y;

   if ( xIdx < w && yIdx < h ) {
      unsigned int idx_in = xIdx + w * yIdx;
      unsigned int idx_out = yIdx + h * xIdx;

      out[idx_out] = in[idx_in];
   }
}
```

read側(in)は、結合されるが、
write側(out)側は結合されない。

http://www.sintef.no/upload/IKT/9011/SimOslo/eVITA/2008/seland.pdf

Optimization of memory access



- By blocking, fetch block of data from shared memory, and store back the block of data to shared memory.

- The above example, thread block of 16 x 16 execute.
- Matrix is read and write for each 16 x 16 block
- When write back, write access is coalesced by contiguous memory address.

http://www.sintef.no/upload/IKT/9011/SimOslo/eVITA/2008/seland.pdf

Optimized code (Coaleased)

```
__global__ void
transpose( float *out, float *in, int w, int h ) {
  __shared__ float block[BLOCK_DIM*BLOCK_DIM];
  unsigned int xBlock = blockDim.x * blockIdx.x;
  unsigned int yBlock = blockDim.y * blockIdx.y;
  unsigned int xIndex = xBlock + threadIdx.x;
  unsigned int yIndex = yBlock + threadIdx.y;
  unsigned int index_out, index_transpose;
  if ( xIndex < width && yIndex < height ) {
    unsigned int index_in = width * yIndex + xIndex;
    unsigned int index_block = threadIdx.y * BLOCK_DIM + threadIdx.x;
    block[index_block] = in[index_in];
    index_transpose = threadIdx.x * BLOCK_DIM + threadIdx.y;
    index_out = height * (xBlock + threadIdx.y) + yBlock + threadIdx.x;
  }
  __synchthreads();
  if ( xIndex < width && yIndex < height ) {
    out[index_out] = block[index_transpose];
  }
}
```

http://www.sintef.no/upload/IKT/9011/SimOslo/eVITA/2008/seland.pdf

- **Example results**

Grid Size	Coalesced	Non-coalesced	Speedup
128×128	0.011 ms	0.022 ms	$2.0\times$
512×512	0.07 ms	0.33 ms	$4.5\times$
1024×1024	0.30 ms	1.92 ms	$6.4\times$
1024×2048	0.79 ms	6.6 ms	$8.4\times$

http://www.sintef.no/upload/IKT/9011/SimOslo/eVITA/2008/seland.pdf

Optimization of Host-device communication

- **The bandwidth between host and device is very narrow compared with the bandwidth of device memory.**
  - Peak bandwidth 4GB/s （PCIe x16 1.0） vs. 76 GB/s （Tesla C870）
- **Minimize the communication between host-device**
  - Intermediate results must be kept in device memory to avoid communications
- **Grouping communication**
  - Large chunk of communication is more efficient than several small chunk of communications
- **Asynchronous communication**
  - Make use of stream
  - cudaMemcpyAsync(dst, src, size, direction, 0);

Host Synchronization

- **All kernel launches are *asynchronous***
  - **control returns to CPU immediately**
  - **kernel executes after all previous CUDA calls have completed**

- **`cudaMemcpy()` is *synchronous***
  - **control returns to CPU after copy complete**
  - **copy starts after all previous CUDA calls have completed**

- **`cudaThreadSynchronize()`**
  - **blocks until all previous CUDA calls complete**

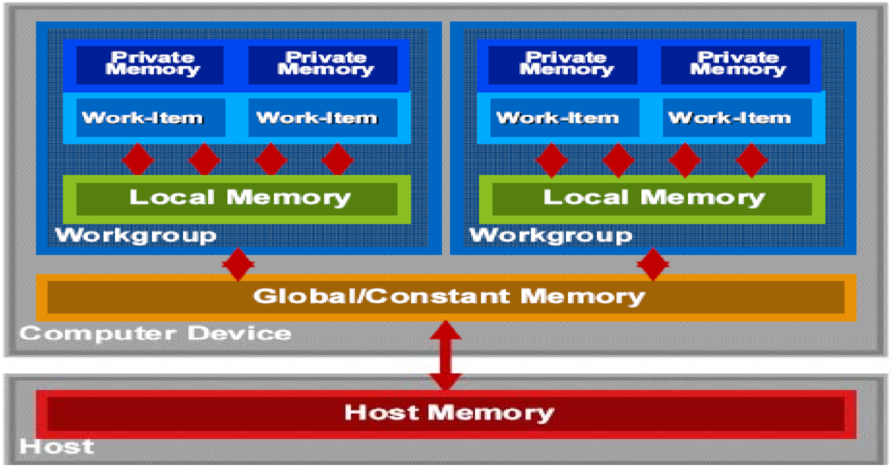
OpenCL

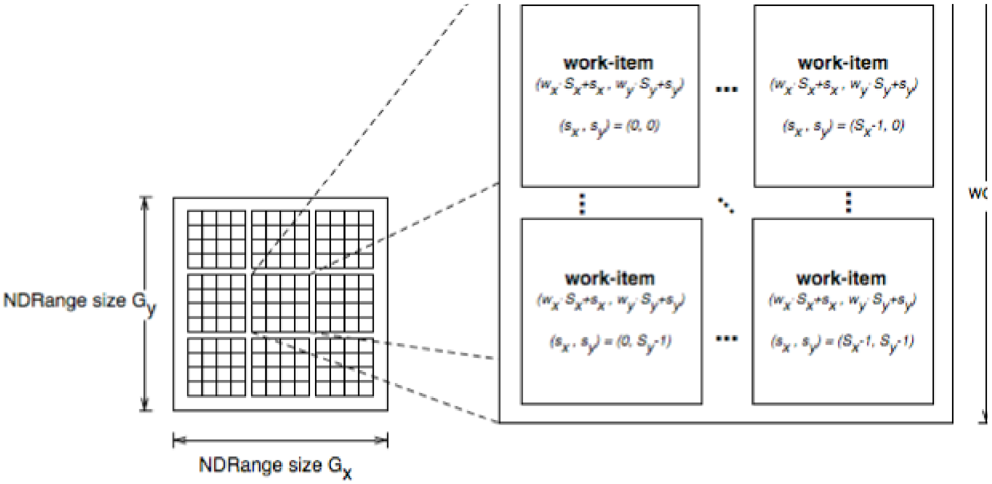
- Programming language for general purpose GPU computing.
- While C for CUDA is proprietary by NVIDIA, OpenCL is targeting cross-platform environments.
  - Only only for GPU such as NVIDIA and AMD(ATI), but also for conventional multicore CPU and many-core, such as Cell Broadband Engine(Cell B.E) and Intel MIC
- The point is that it targets for data parallel program by GPU and also for task-parallel of multi-core.
- What is different from CUDA? ： Similar programming mode for kernel, but different in execution environment.

Kernel and Memory model

OpenCL Memory Model

- Private Memory
  - Per work-item
- Local Memory
  - Shared within a workgroup (16Kb)
- Local Global/Constant Memory
  - Not synchronized
- Host Memory
  - On the CPU

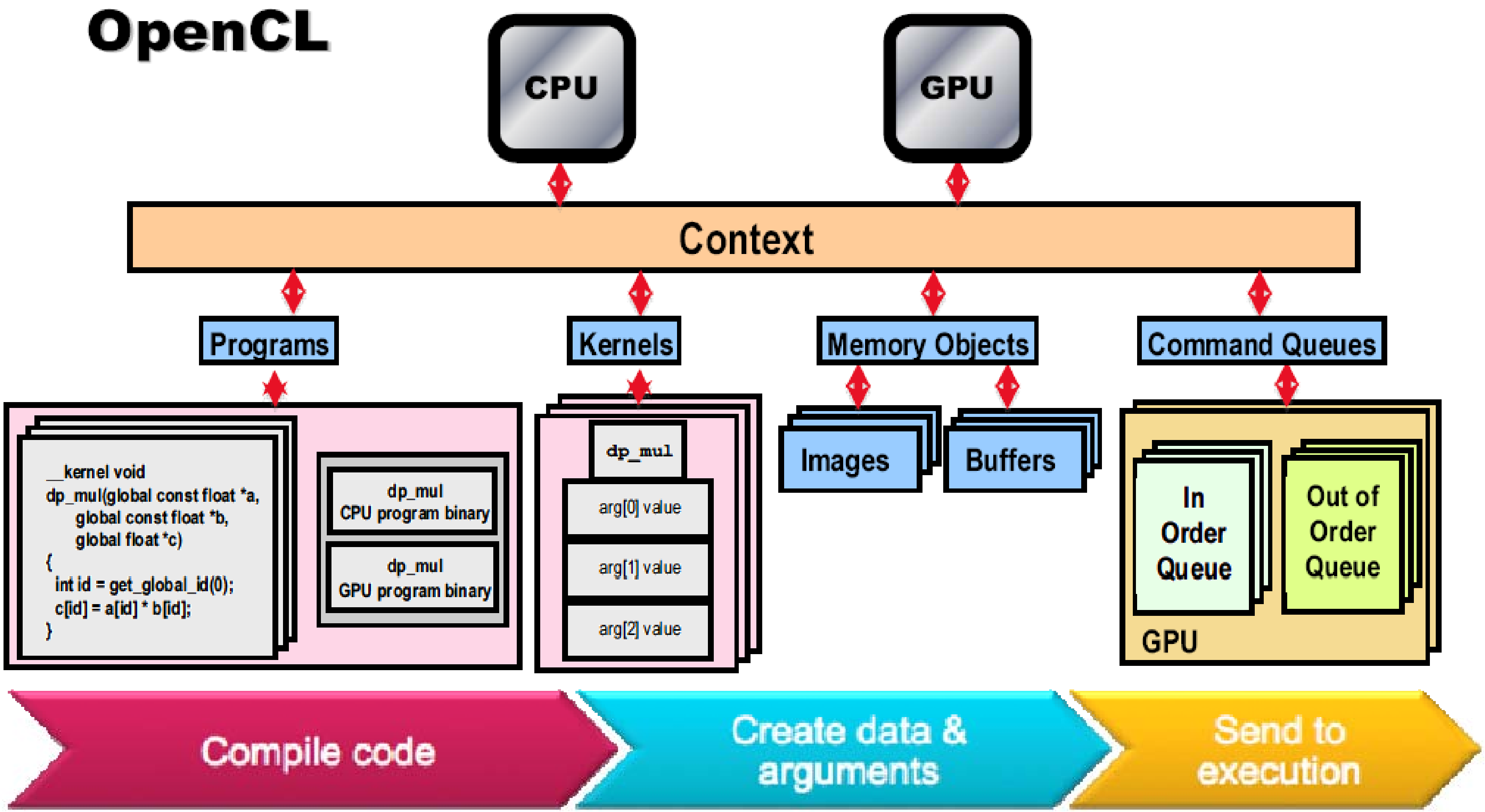




Data Parallel

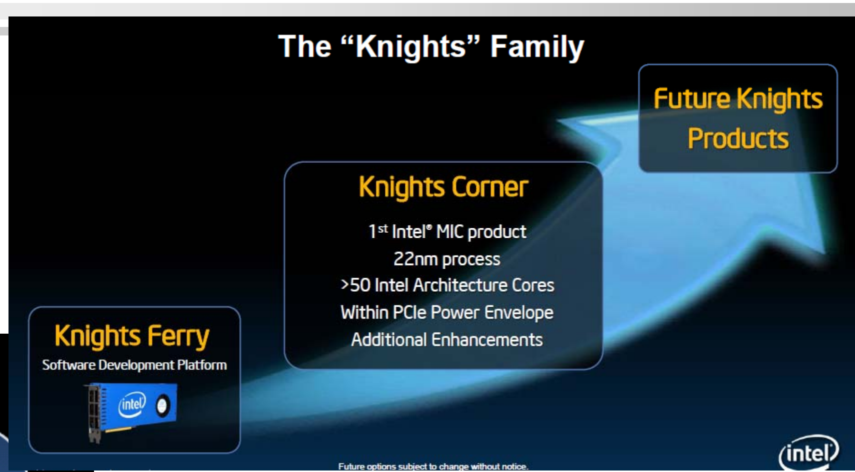
```
kernel void
dp_mul(global const float *a,
       global const float *b,
       global float *result)
{
  int id = get_global_id(0);

  result[id] = a[id] * b[id];
}
// execute dp_mul over "n" work-items
```

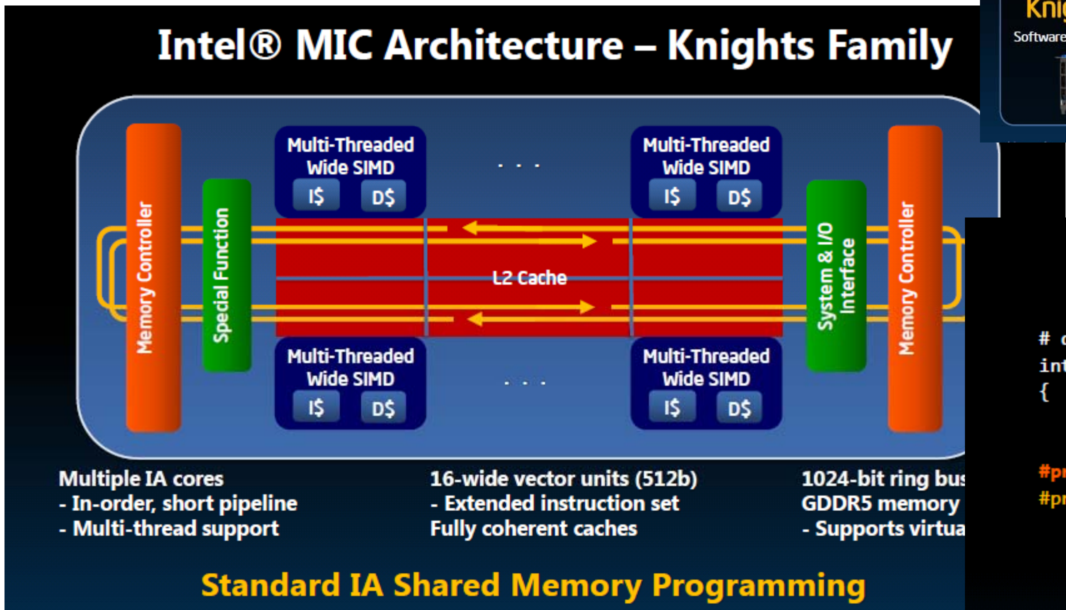
Execution Evnvironment of OpenCL

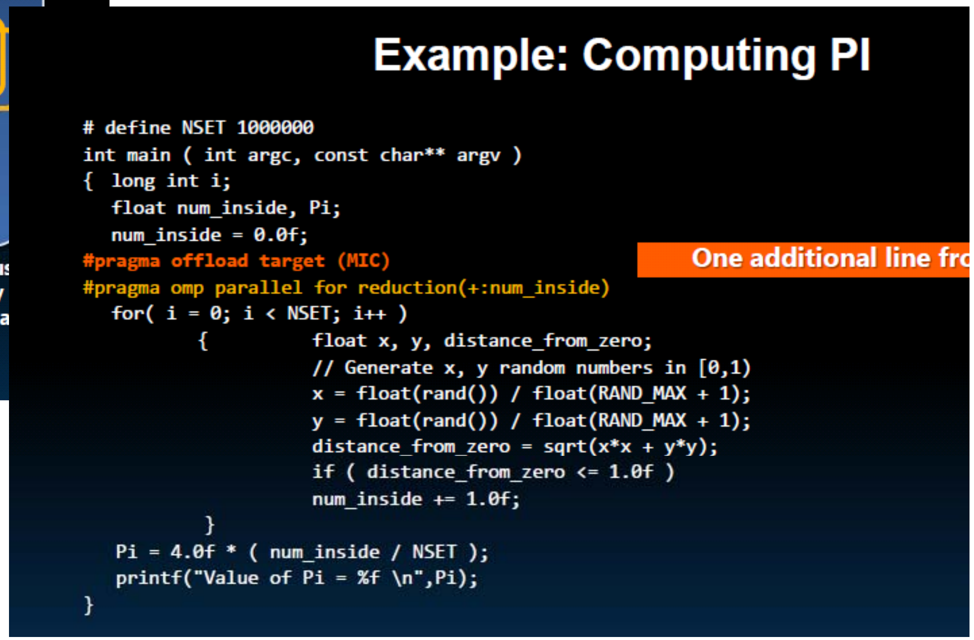


Intel MIC

- Intel Manycore architecture







OpenACC

- A spin-off activity from OpenMP ARB for supporting accelerators such as GPGPU and MIC

- NVIDIA, Cray Inc., the Portland Group (PGI), and CAPS enterprise

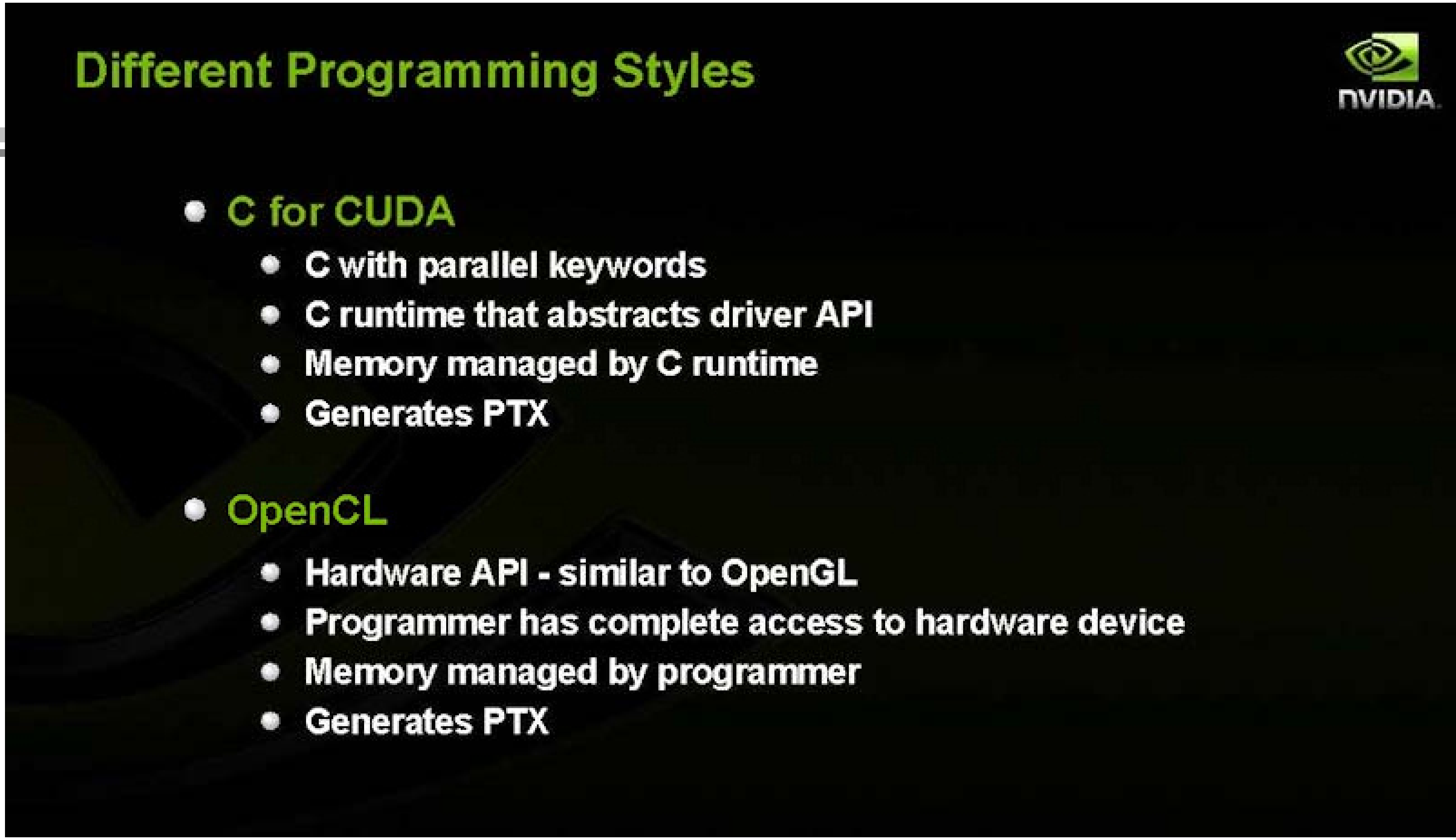
- Directive to specify the code offloaded to GPU.
  - #pragma acc region

```
!$acc region
      do k = 1,n1
       do i = 1,n3
        c(i,k) = 0.0
        do j = 1,n2
         c(i,k) = c(i,k) + a(i,j) * b(j,k)
        enddo
       enddo
      enddo
!$acc end region
```

```
float f(int n, float* v1, float* v2)
{
  int i;
  float sum = 0;
  #pragma acc region for
  for (i=0; i<n; i++)
  {
    // Do some heavy computations here!
  }
  return sum;
}
```

最後に

- GPGPUは、適合するアプリであれば非常に有望なソリューション
  - 特に、1GPUで１つのホストでやる場合
  - アプリケーションによってはだめな場合も…
- CUDAで簡単になったとはいえ、まだ、難しい
  - 特に、性能チューニング、メモリ配置、結合…
- 全体のコントロールフローは、ホスト側でやらなくてはならない
  - kernelは　local view プログラム
- １ノードを超えて、次の段階にいけるか？
  - マルチGPUーGPUを複数枚
  - マルチノードGPU -- クラスタにGPUをつけて並列計算
  - やはり、もうすこし、まともなプログラミング環境が必要？？？？



現状のC for CUDAとOpenCLでは、位置付けがずれる。
OpenCLがミドルウェアの土台としての色彩が濃いローレベルAPIであるのに対して、C for CUDAの方が抽象化の度合いが高くアプリケーションを書きやすい

http://pc.watch.impress.co.jp/docs/2009/0330/kaigai497.htm