GPGPUプログラミング入門

佐藤

筑波大学

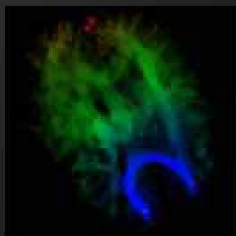
参考にした資料

- NVIDA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

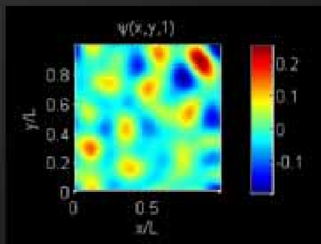
- GPGPU - General-Purpose Graphic Processing Unit
  - グラフィック専用だったGPUを、汎用の計算に使う技術
- CUDA （Compute Unified Device Architecture）
  - GPUコンピューティングのためのNVIDIA GPUの計算能力を発揮するために共同デザインされた、ハードウェアとソフトウェア
  - 逆に言えば、現時点で、GPGPUの性能を引き出すにはCUDAのようなプログラミング言語で書かなくてはならない
- ある特定の科学技術アプリケーションにおいては、CPU（マルチコアでさえも）の性能を大きくしのぐ性能を得ることができる、ため注目を集めている。
- なぜ、いま、GPGPUか。ーー price!!!

アプリケーション例（NVIDIAのスライドから）



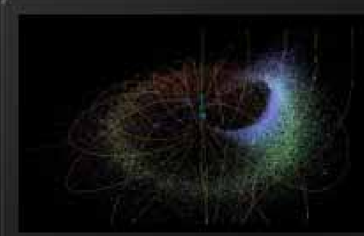
146X

容積測定時の白質連結のインタラクティブな視覚化



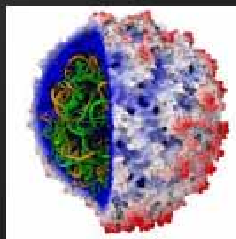
17X

Matlabでの等方性乱流シミュレーション



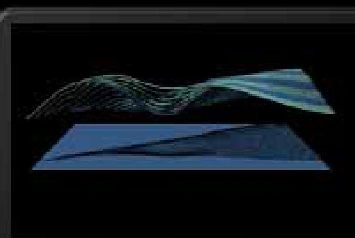
100X

天体物理学におけるN体計算



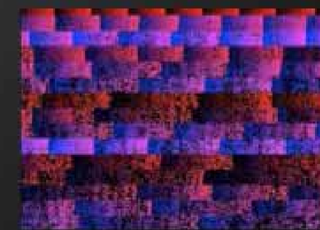
110X

分子動力学におけるイオン配置



149X

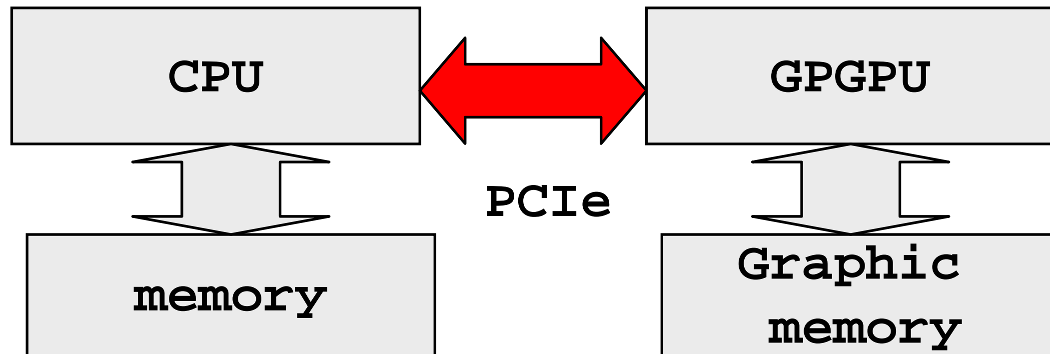
スワップションのあるLIBORモデルの金融シミュレーション



30X

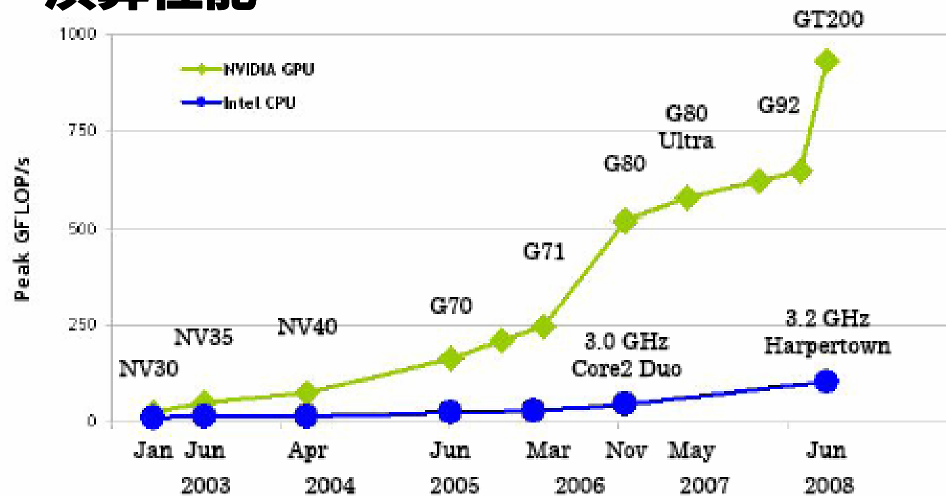
類似タンパク質および遺伝子配列検索の厳密なCmatch文字列照合

CPUとGPU

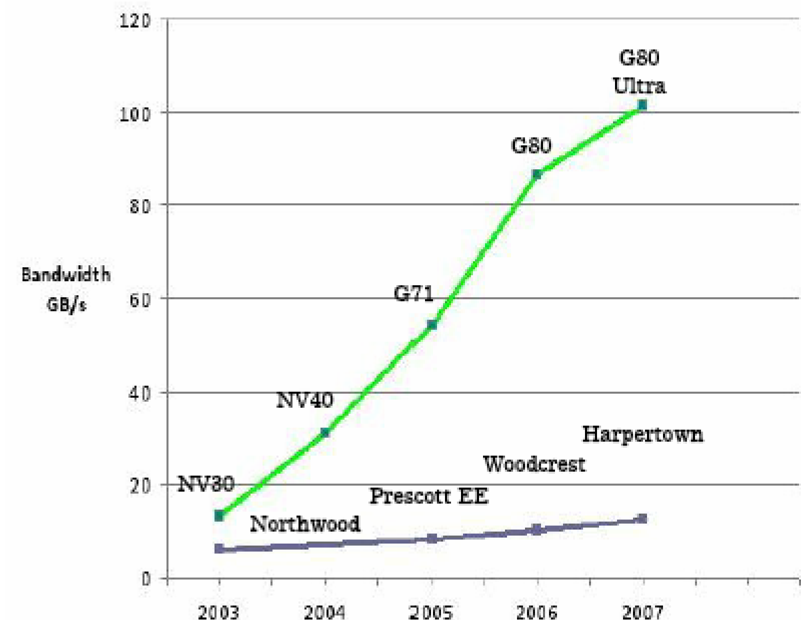


PCIexpress
で接続されている

演算性能

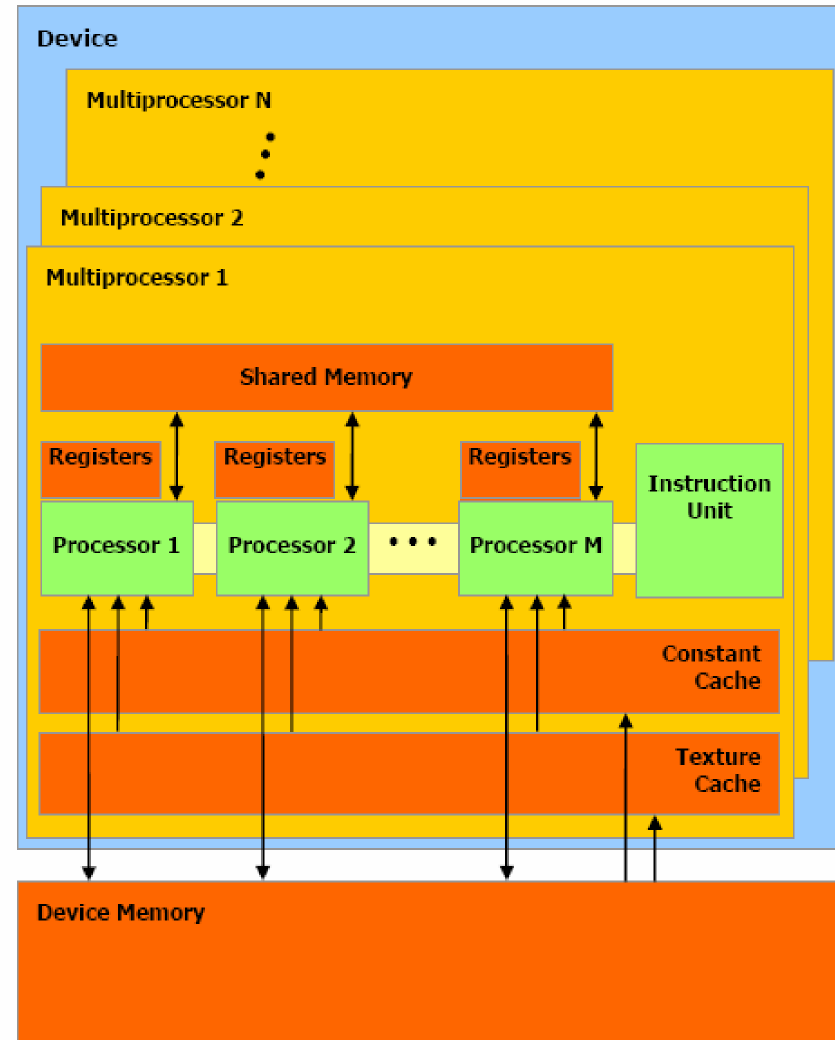


メモリバンド幅



NVIDIA GPGPUのアーキテクチャ

- 一つのチップの中に、複数のmultiprocessorがある
  - 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.
- 複雑なメモリ階層
  - Device Memory (Global Memory)
  - Shared Memory
  - Constant Cache
  - Texture Cache

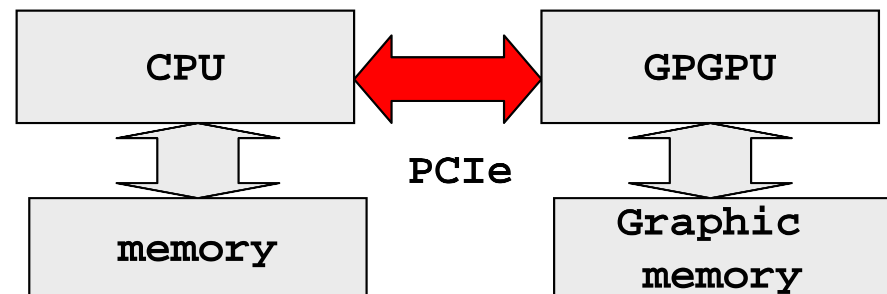


CUDA (Compute Unified Device Architecture)

- 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

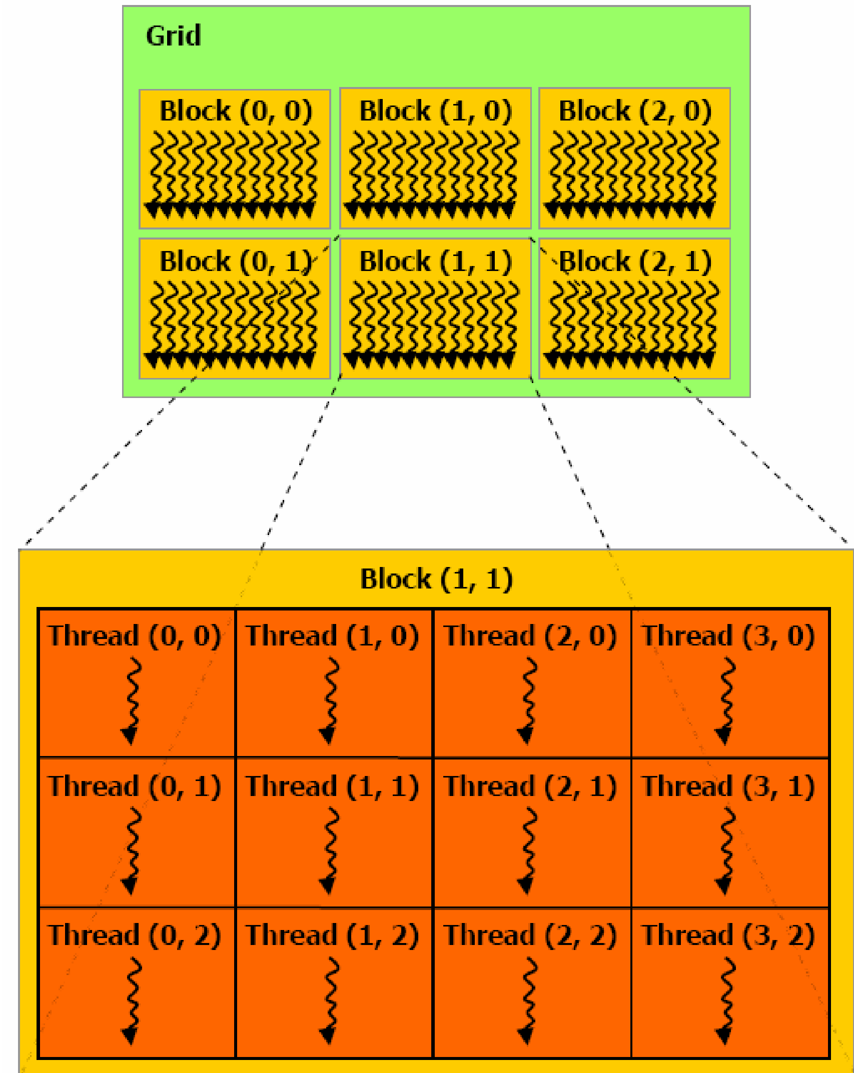
CUDAのプログラミングモデル (1/2)

- GPUは、CPU(host)からはco-processorとして動くcompute deviceとして見える
  - データ並列、compute intensiveなプログラムの一部を、関数として、deviceにoff-loadする。
  - 頻繁に実行される部分で、個別のデータで計算する部分
    - 例えば、ループのbodyの部分
  - deviceの関数としてコンパイルされる部分を、kernelと呼ぶ
  - kernelは、複数のスレッドでdevice上で実行される
    - 同時には１つのkernelだけが、device上で実行される
  - host (CPU)とdevice(GPU)は、それぞれのメモリ、host memory とdevice memoryを管理しなくてはならない
  - データは、その間でコピーを行う



CUDAのプログラミングモデル (2/2)

- 計算グリッド(computational Grid)はいくつかのthread Blockからなる
- thread blockはいくつかのスレッドからなる。
- 各スレッドは、kernelを実行する。
  - 各スレッドで実行される関数をkernelと呼ぶ
- computational Gridとblockは、1,2,3次元でもいい。
- blockIDとthreadIDは、予約変数で、現在実行中のスレッドの番号がわかる



例 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

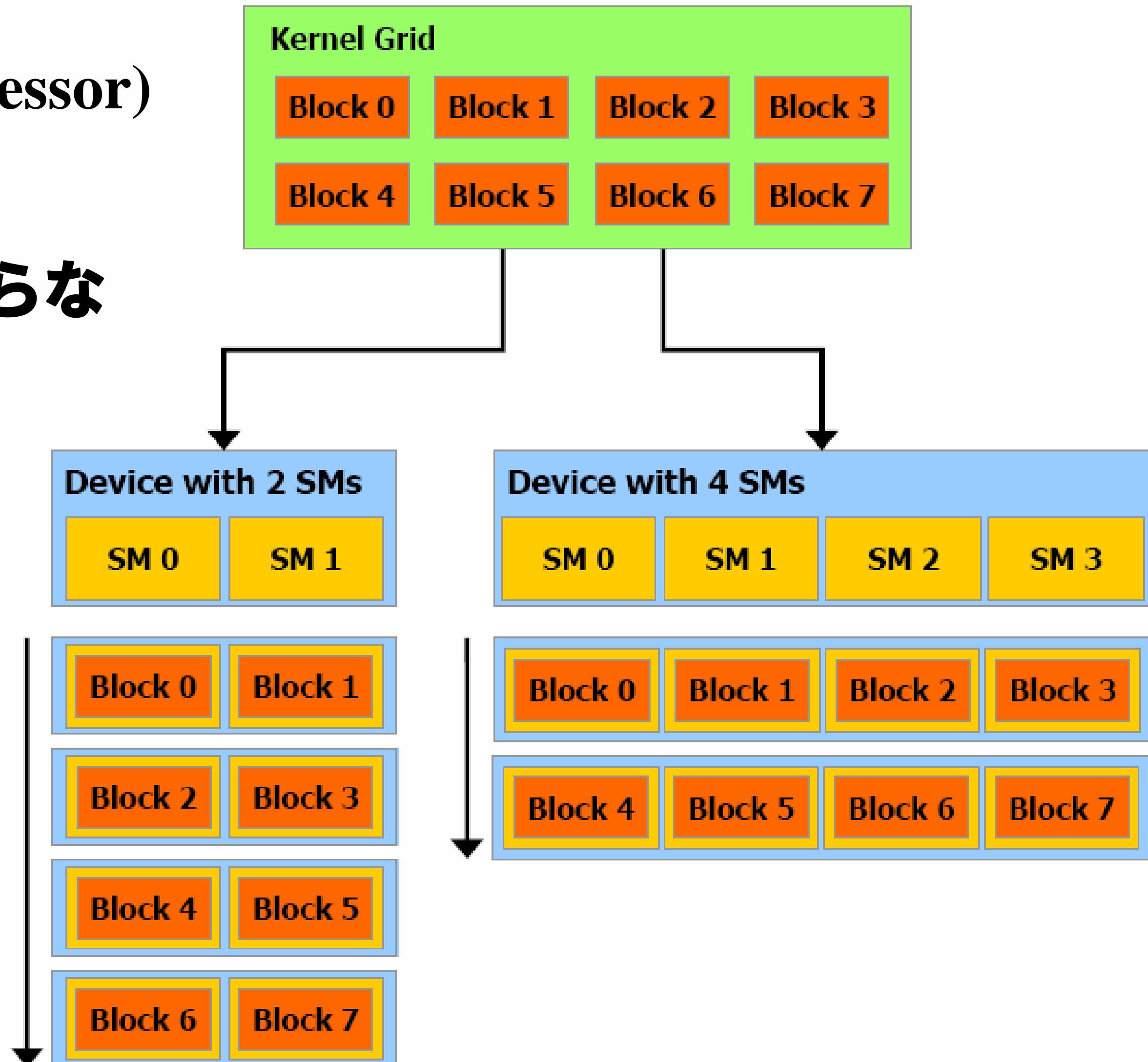
CUDA program

```
__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

ハードウエアでの実行

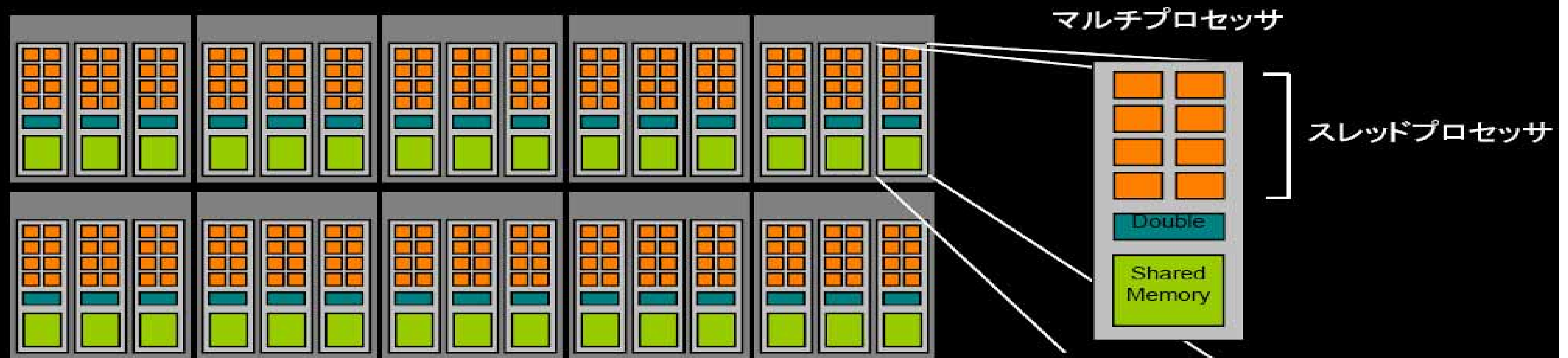
- ブロックごとにSM (Streaming Multiprocessor) で実行される。
- SMは、8processorからなる



GPGPUの例

10シリーズアーキテクチャ

- 240個のスレッドプロセッサがカーネルスレッドを処理
- 30個のマルチプロセッサ、それぞれが次のユニットを内蔵
  - 8個のスレッドプロセッサ
  - 1個の倍精度ユニット
  - スレッド協調のための共有メモリ



プログラミング環境特論

	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

カーネルの起動

- ホストからの呼び出しはＣ関数呼び出し構文の変形

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

- 実行の構成（“<<< >>>”）
  - グリッドの次元: xとy
  - スレッドブロックの次元: x、y、z

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

CUDAカーネルとスレッド

- アプリケーションの並列部分はカーネルとしてデバイス上で実行される
  - 1度に実行されるカーネルは1つ
  - 多数のスレッドが各カーネルを実行する

- CUDAスレッドとCPUスレッドの違い
  - CUDAスレッドは非常に軽量
    - 作成のオーバーヘッドがほとんどない
    - 瞬時に切り替えが可能
  - CUDAは数千ものスレッドによって高い効率性を実現
    - マルチコアCPUで使用できるスレッドは、数個のみ

プログラミング環境特

ホストとデバイスでの実行の関係

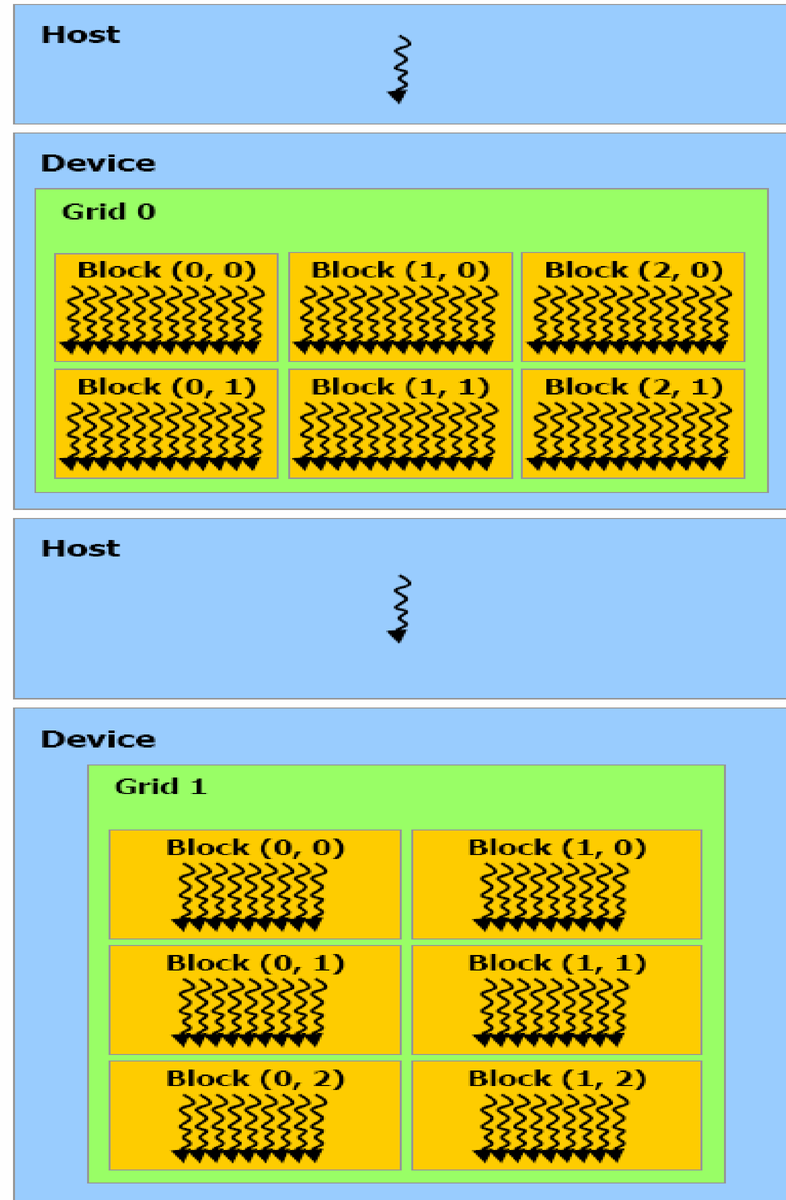
C Program
Sequential
Execution

Serial code

Parallel kernel
Kernel0<<<>>>()

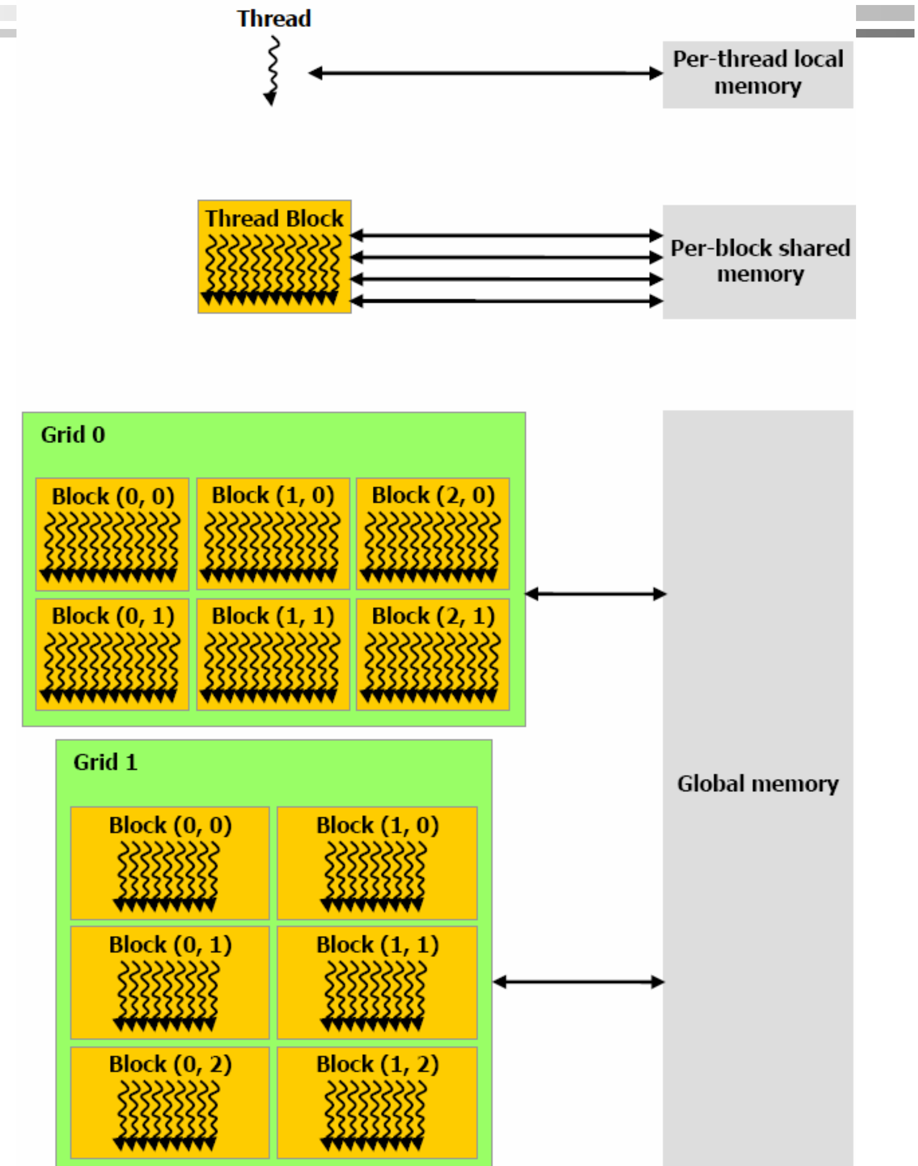
Serial code

Parallel kernel
Kernel1<<<>>>()



グリッド、ブロック、スレッドとメモリ階層

- スレッドは、ローカルメモリにアクセスできる
- スレッドは、ブロックに対応した共有メモリにアクセスできる
- グリッドは、グローバルメモリにアクセスする。



メモリ管理

- CPUとGPUは別のメモリ空間を持つ
- ホスト（CPU）コードがデバイス（GPU）メモリを管理する

- **CPUメモリの割り当て／解放**
  - 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);
```

メモリ管理

◆データのコピー

- cudaMemcpy(void *dst, void *src, size_t nbytes, enum cudaMemcpyKind direction);
  - directionは、（ホストまたはデバイスの）srcとdstの位置を指定
  - CPUスレッドをブロック: コピーが完了したら戻す
  - 前のCUDA呼び出しが完了するまでコピーは開始されない
- enum cudaMemcpyKind
  - cudaMemcpyHostToDevice
  - cudaMemcpyDeviceToHost
  - cudaMemcpyDeviceToDevice

GPUでのコードの実行

- カーネルは何らかの制限のあるC関数
  - GPUメモリのみにアクセスできる
  - 戻り型voidが必要
  - 可変個引数（varargs）なし
  - 再帰的にはできない
  - static変数なし
- 関数の引数はCPUからGPUメモリに自動的にコピーされる

関数の修飾子

- `__global__` : ホスト（CPU）コード内で呼び出し。デバイス（GPU）コードからは呼び出せない。戻り型voidが必要
- `__device__` : 別のGPU関数から呼び出される。ホスト（CPU）コードからは呼び出せない
- `__host__` : CPUでのみ実行可能。ホストから呼び出される
- `__host__`と`__device__`の修飾子は組み合わせられる
  – 使用例: 演算子のオーバーロード
  – コンパイラはCPUコードとGPUコードの両方を生成する

CUDA組み込みデバイス変数

- `__global__`と`__device__`のすべての関数は、これらの自動的に定義された変数にアクセスできる
  - 予約された変数、スレッドの位置を計算するために用いる
  - `dim3 gridDim;`
    - ブロックのグリッドの次元（最大で2次元）
  - `dim3 blockDim;`
    - スレッドのブロックの次元
  - `dim3 blockIdx;`
    - グリッド内のブロックのインデックス
  - `dim3 threadIdx;`
    - ブロック内のスレッドのインデックス

簡単な例

```
__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;
}
```

簡単な例

```
__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)>>>(...);
```

配列のインクリメントの例

CPUのコード

```
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のプログラム

```
__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);
}
```

簡単な例（ホスト側）

```
// ホストメモリを割り当て
int numBytes = N * sizeof(float)
float* h_A = (float*) malloc(numBytes);

// デバイスメモリを割り当て
// float* d_A = 0;
cudaMalloc((void**)&d_A, numbytes);

// ホストからデバイスにデータをコピー
cudaMemcpy(d_A, h_A, numBytes, cudaMemcpyHostToDevice);

// カーネルを実行
increment_gpu<<< N/blockSize, blockSize>>>(d_A, b);

// デバイスからホストにデータをコピーバック
cudaMemcpy(h_A, d_A, numBytes, cudaMemcpyDeviceToHost);

// デバイスメモリを解放
cudaFree(d_A);
```

プログラミ

```
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;
}
```

変数の修飾子

- __device__
  - デバイスメモリに保存する（大容量、高いレイテンシ、キャッシュなし）
  - cudaMallocで割り当てられる（__device__修飾子は暗黙）
  - すべてのスレッドからアクセス可能
  - 生存期間: アプリケーション

- __shared__
  - 内蔵の共有メモリに保存する（レイテンシは非常に低い）
  - 実行の構成またはコンパイル時に割り当てられる
  - 同じブロック内のすべてのスレッドからアクセス可能
  - 生存期間: カーネルの実行中

- 非修飾の変数
  - スカラと組み込みのベクトル型はレジスタに保存される
  - 要素が5以上ある配列はデバイスメモリに保存される

共有メモリの使い方

コンパイル時にサイズを指定

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

カーネルの起動時に指定

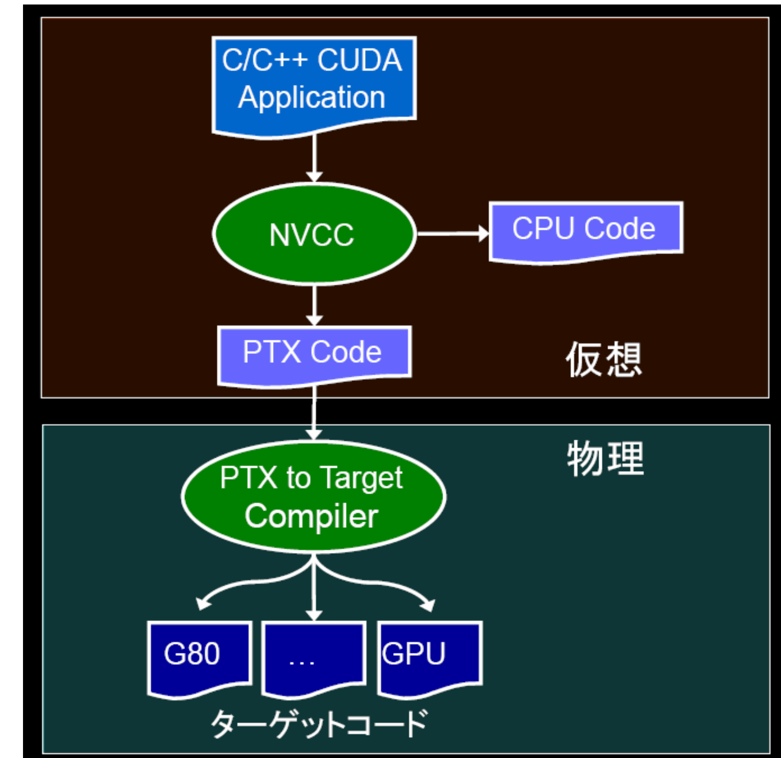
```
__global__ void kernel(…)
{
   …
   extern __shared__ float sData[];
   …
}

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

GPUスレッドの同期

- `void __syncthreads();`
  - ブロック内のすべてのスレッドを同期する
  - 境界同期命令を生成する
  - ブロック内のすべてのスレッドに達するまで、どのスレッドもこの境界を越えることはできない
  - 共有メモリへのアクセスでRAW／WAR／WAWハザードを避けるために使用
- 条件がスレッドブロック全体で一律の場合に限り、条件コードに記述できる
- ブロック間のスレッドの同期はできない
  - ホスト側で行う。

コンパイラ

- CUDA言語拡張が記述されているソースファイルは、nvccでコンパイルする必要がある
- nvccはコンパイラドライバ
  - すべての必要なツールとcudacc、g++、clなどのコンパイラを起動させることで動作する
- nvccは以下のいずれかを出力できる
  - Cコード（CPUコード）
  - 別のツールを使用して、アプリケーションの残りといっしょにコンパイルする必要がある
  - PTXオブジェクトコードに直接
- CUDAコードの実行可能プログラムで必要なもの
  - CUDAコアライブラリ（cuda）
  - CUDAランタイムライブラリ（cudart）
  - ランタイムAPIを使用する場合CUDAライブラリをロードする



GPUでのアルゴリズムの最適化

- 独立した並列性を最大化する
- 計算集約度を最大にする（計算／帯域幅）
- キャッシュするよりも再計算するほうが適している場合もある
  - GPUはメモリでなく論理演算装置でトランジスタを消費
- GPUでの計算を多くして、負荷の高いデータ転送を避ける
  - 並列性が低い計算でもホストとの転送を行うよりも速い場合がある

メモリアクセスの最適化

- ホストーデバイス間のデータ転送の最適化
- グローバルデータアクセスの結合
- 共有メモリの活用
  - グローバルメモリよりも数百倍高速
  - スレッドは共有メモリを介して協調できる
  - 1つまたは少数のスレッドを使用して、すべてのスレッドに共有されるデータのロードや計算を実行する
  - 非結合アクセスを避けるために使用する
    - 結合vs. 非結合=桁違いの差
  - 共有メモリはオンチップで帯域幅が非常に広い
    - 低いレイテンシ
    - ユーザー管理のマルチプロセッサごとのキャッシュと同様

いろいろなメモリの利用

- **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

グローバルメモリへのアクセス

- 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）

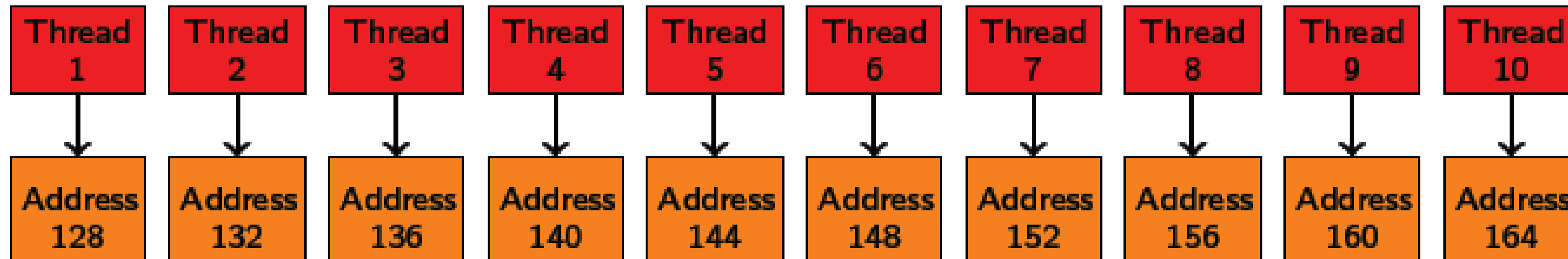
メモリアクセスの結合（coalesce）

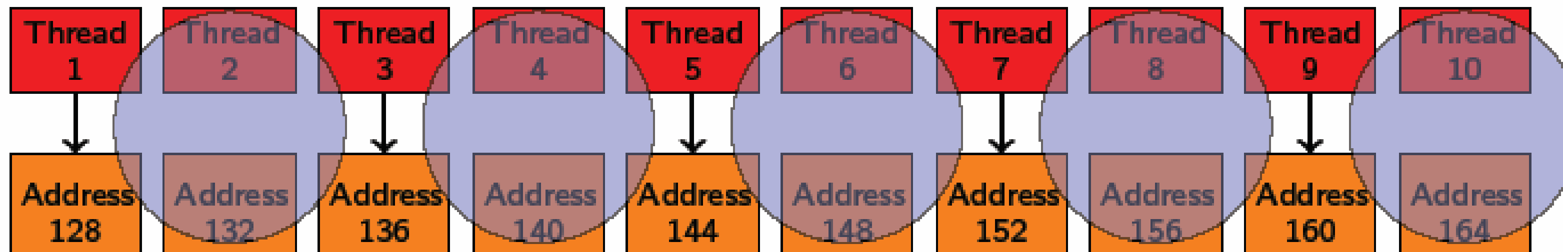
性能を上げるためには、グローバルメモリは結合されなくてはならない

- ハーフワープ（16スレッド）で読み込みを協調
- グローバルメモリの連続した領域:
  - 64バイト- 各スレッドはシングルワード（int、floatなど）を読み込む
  - 128バイト- 各スレッドはダブルワード（int2、float2など）を読み込む
  - 256バイト- 各スレッドはクワッドワード（int4、float4など）を読み込む
  - float3はalignされない！！！

- その他の制限
  - 領域の開始アドレス(Warp base address (WBA))は領域サイズの倍数16*sizeof(type)でなくてはならない
  - ハーフワープのk番目のスレッドは読み込まれるスレッドのk番目の要素にアクセスしなくてはならない
  - 全部のスレッドはアクセスしなくてもいい

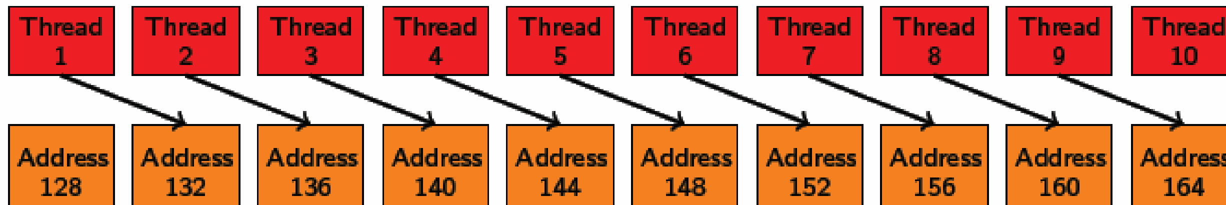
結合される例



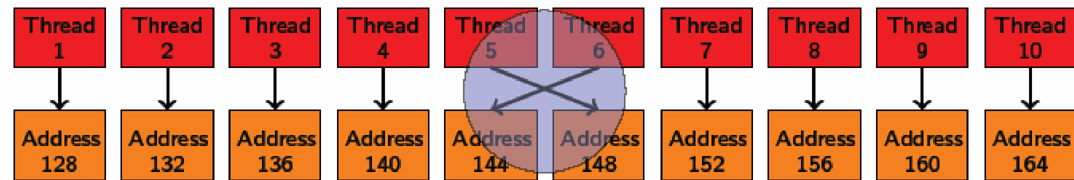


http://www.sintef.no/upload/IKT/9011/SimOslo/eVITA/2008/seland.pdf

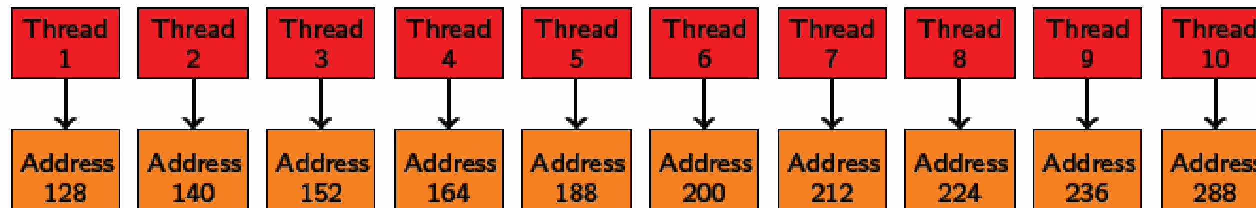
結合されない例



Non-Coalesced memory access:
Misaligned starting address



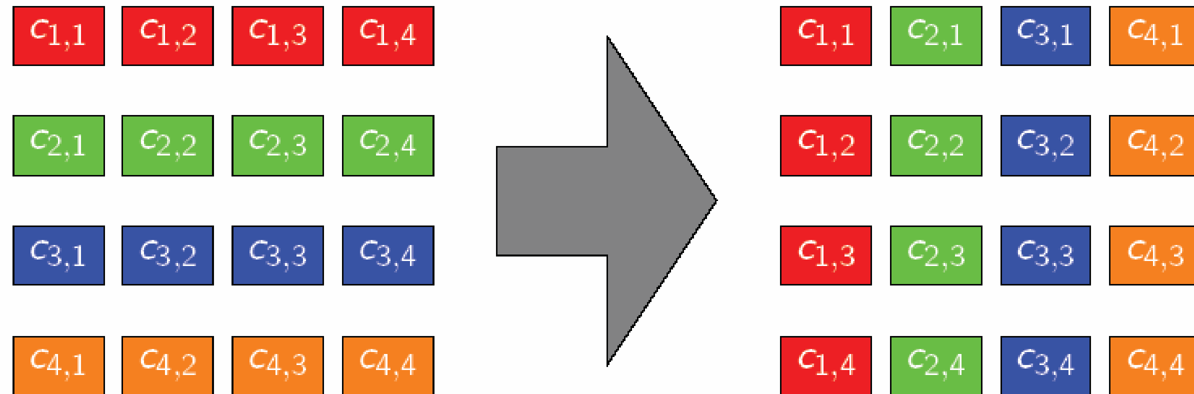
Non-Coalesced memory access:
Non-sequential access



Non-Coalesced memory access:
Wrong size of type

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

メモリ最適化の例：Matrix Transpose

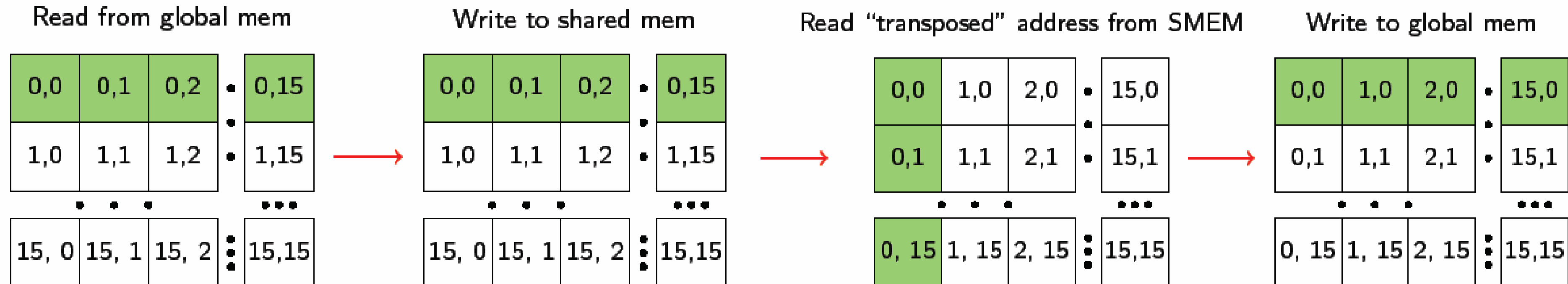



```
__global__ void
transpose_naive( float *out, float *in, int w, int h ) {
  unsigned int xIdx = blockDim.x * blockIdx.x + threadIdx.x;
  unsigned int yIdx = blockDim.y * blockIdx.y + threadIdx.y;

  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

最適化の方針



- ブロック化し、ブロックごとに連続に共有メモリに読み込み、ブロックから連続にグローバルメモリに書き戻す。

- 例では、16 x 16の thread blockで実行
- Matrix は、16 x 16 の単位で、読み書きされる。
- write側の書き戻すときには連続になるために、結合される

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

最適化されたコード(メモリアクセス結合)

```
__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

◆ 結果の例

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

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

ホストーデバイス間のデータ転送

- デバイスメモリからホストメモリの帯域幅はデバイスメモリの帯域幅に比べて非常に狭い
  - ピーク時4GB/s （PCIe x16 1.0） vs.ピーク時76 GB/s （Tesla C870）
- 転送の最小化
  - ホストメモリにコピーすることなく、中間のデータ構造で割り当て、演算、解放できるようにする
- 転送のグループ化
  - 細かい多数の転送よりも、一度の大きい転送のほうがよい
- 非同期通信の利用
  - ストリームの利用
  - cudaMemcpyAsync(dst, src, size, direction, 0);

ホストとの同期

- すべてのカーネルは非同期で起動される
  - 即座にCPUに制御が戻る
  - カーネルは前のCUDA呼び出しがすべて完了してから実行される
- `cudaMemcpy()`は同期
  - コピーの完了後CPUに制御が戻る
  - 前のCUDA呼び出しがすべて完了してからコピーが開始される
- `cudaThreadSynchronize()`
  - 前のCUDA呼び出しがすべて完了するまでブロック

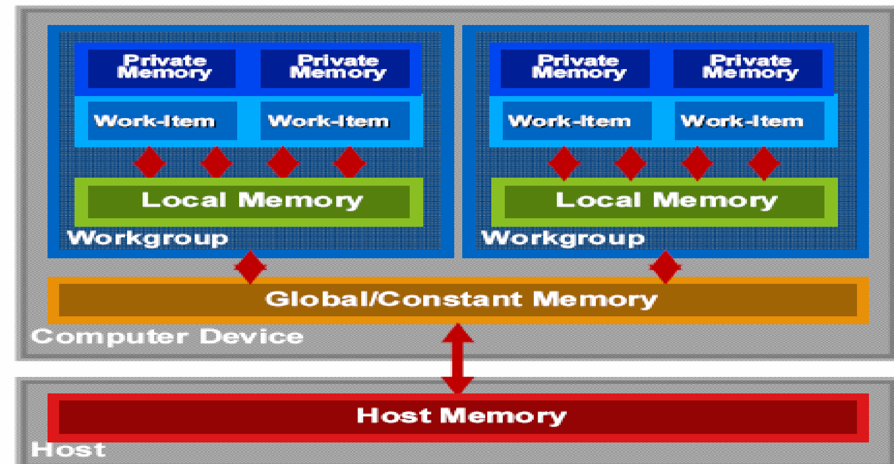
OpenCL

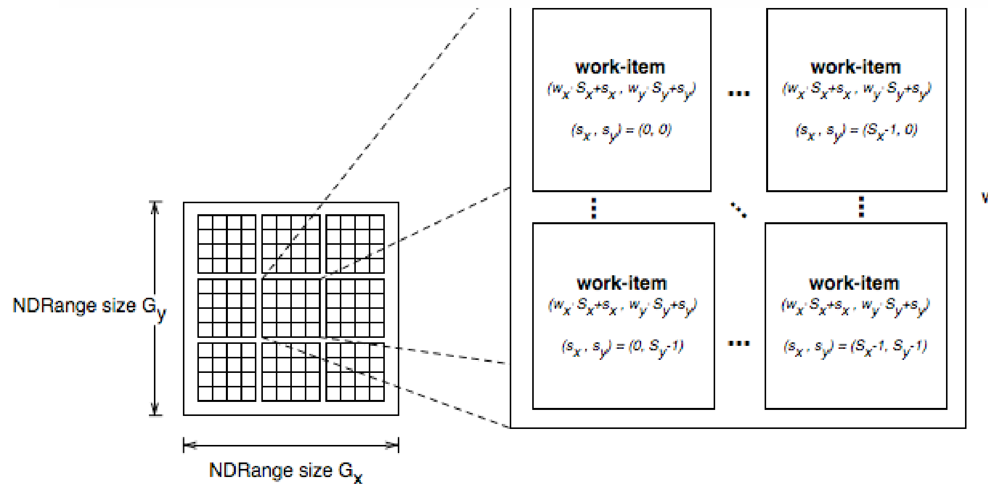
- GPUを汎用コンピューティングに使うことができるプログラミング言語
- NVIDIA独自のC for CUDAとは異なり、クロスプラットフォームのプログラミング環境
  - NVIDIAやAMD(ATI)のGPUだけでなく、マルチコアCPUやCell Broadband Engine(Cell B.E.)といった多様なプロセッサ(Larrabeeも含まれると推測される)をカバー
- 最大の特徴は、GPU型のデータ並列モデルに対応したことだが、同時にマルチコアCPUなどのタスク並列モデルもサポート
- CUDAとの違い：kernelのプログラミングモデルは同じようなものに見えるが、実行環境がずいぶん異なる

kernelとメモリモデル

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

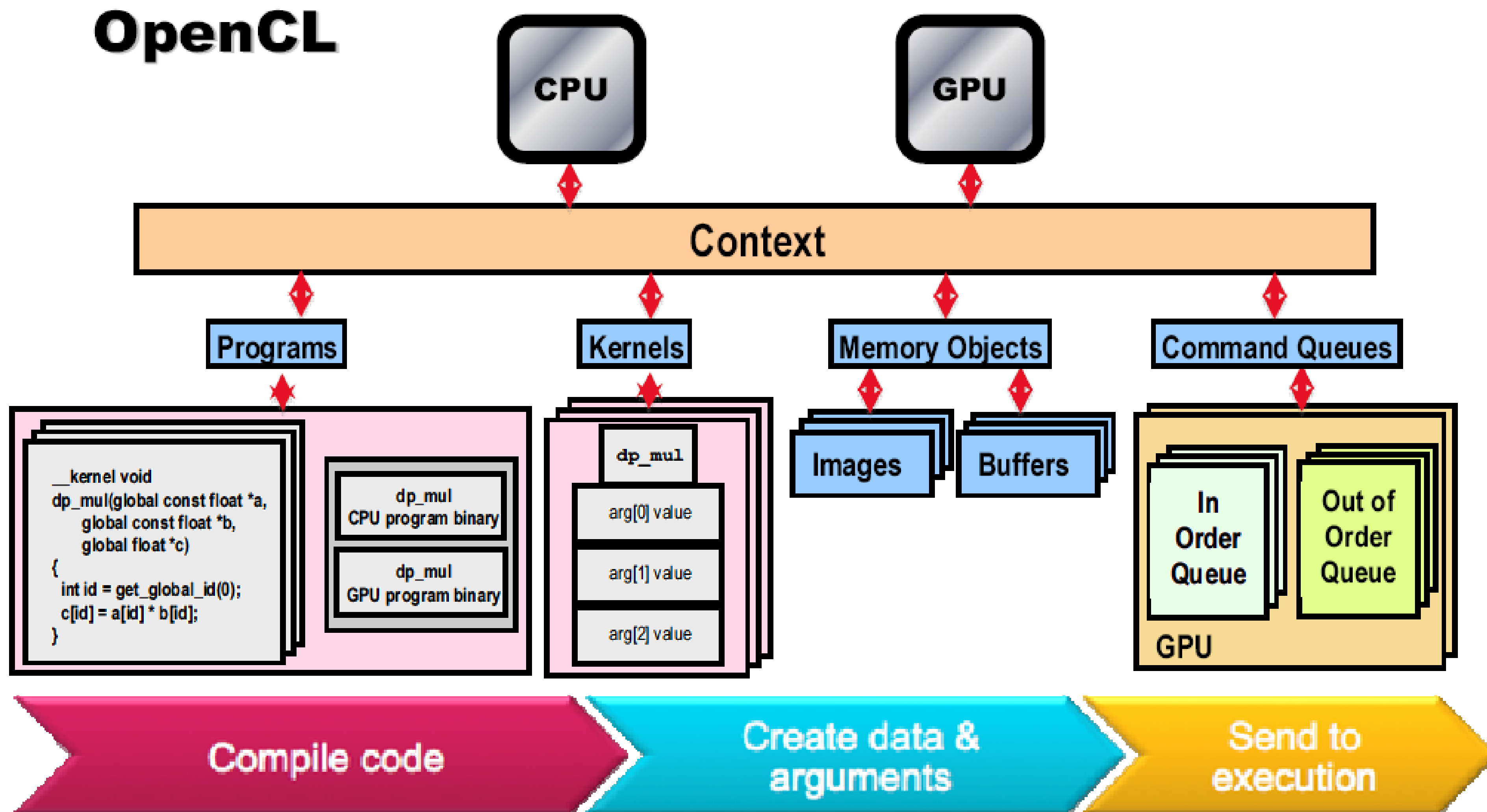




Data Parallel

```
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
```

OpenCLの実行環境



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？？？？