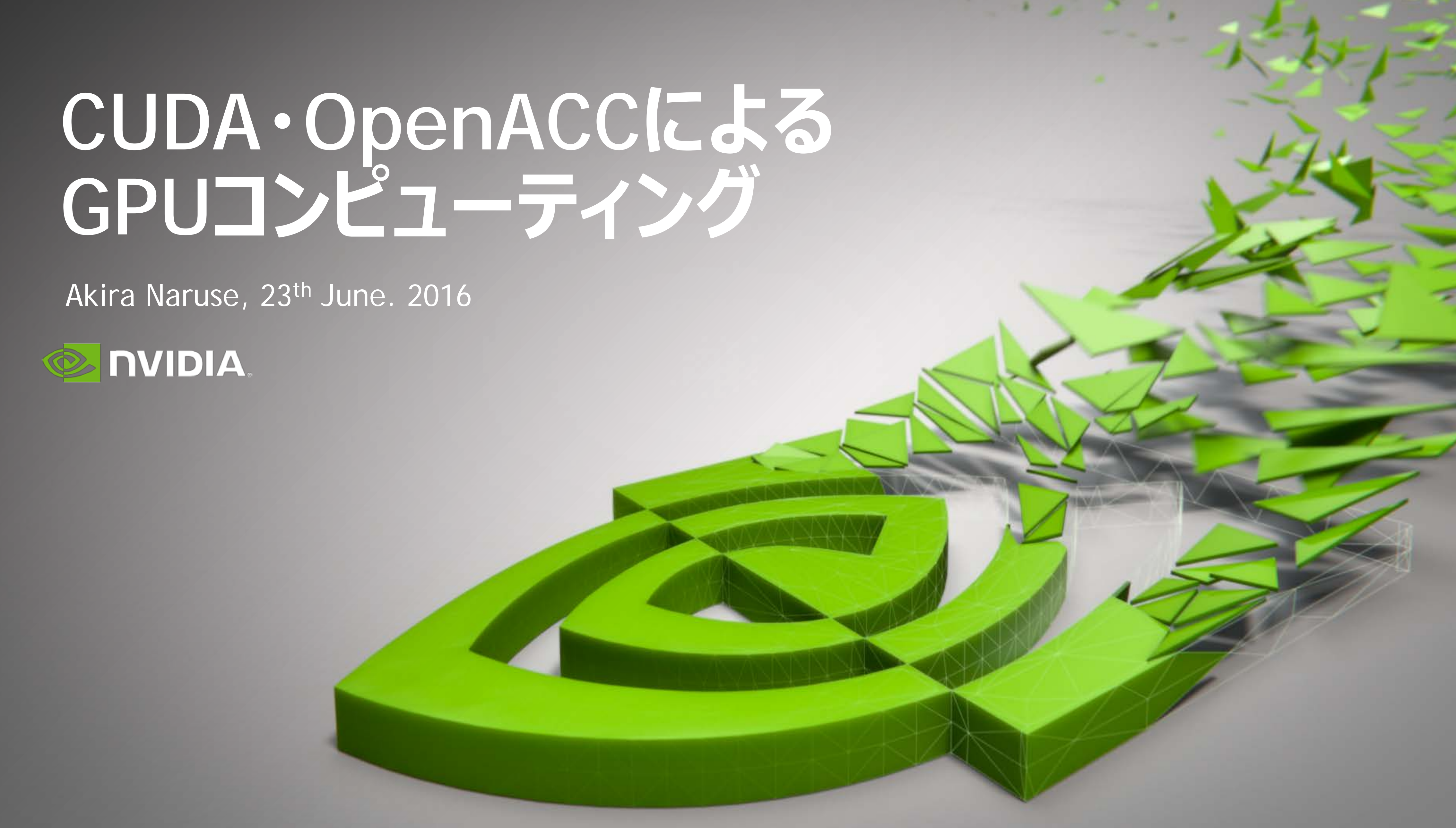
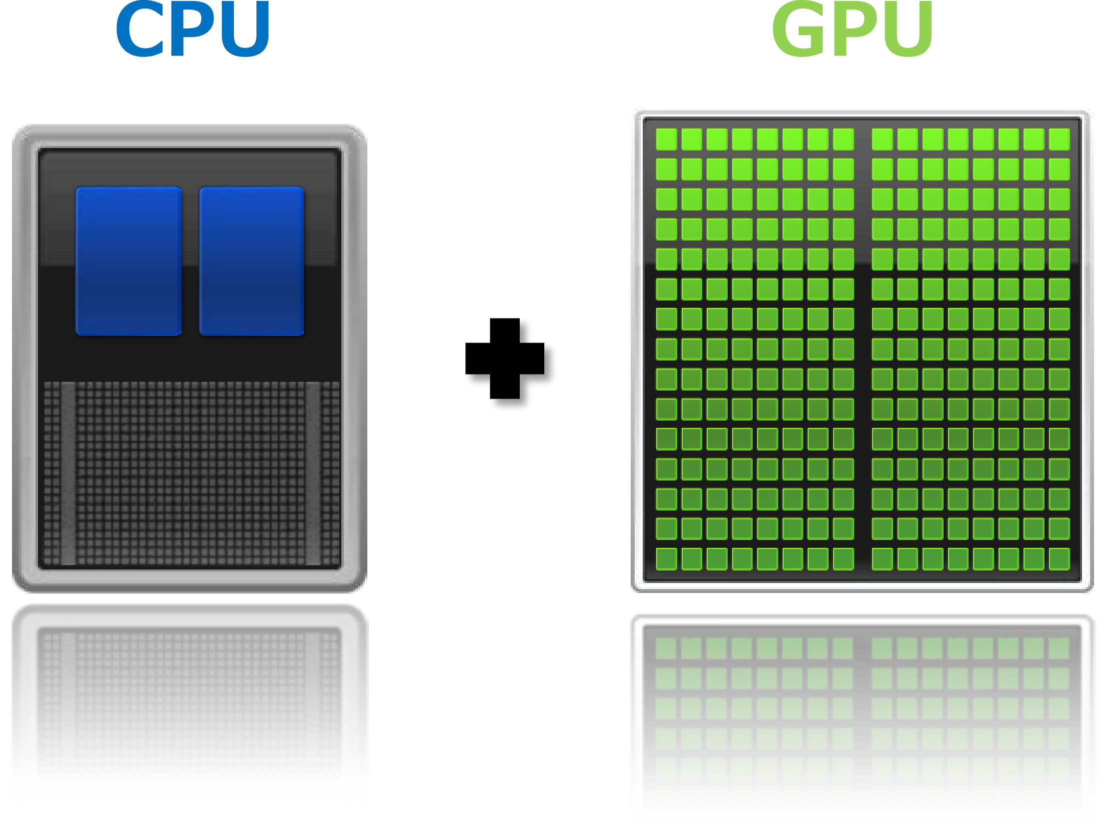
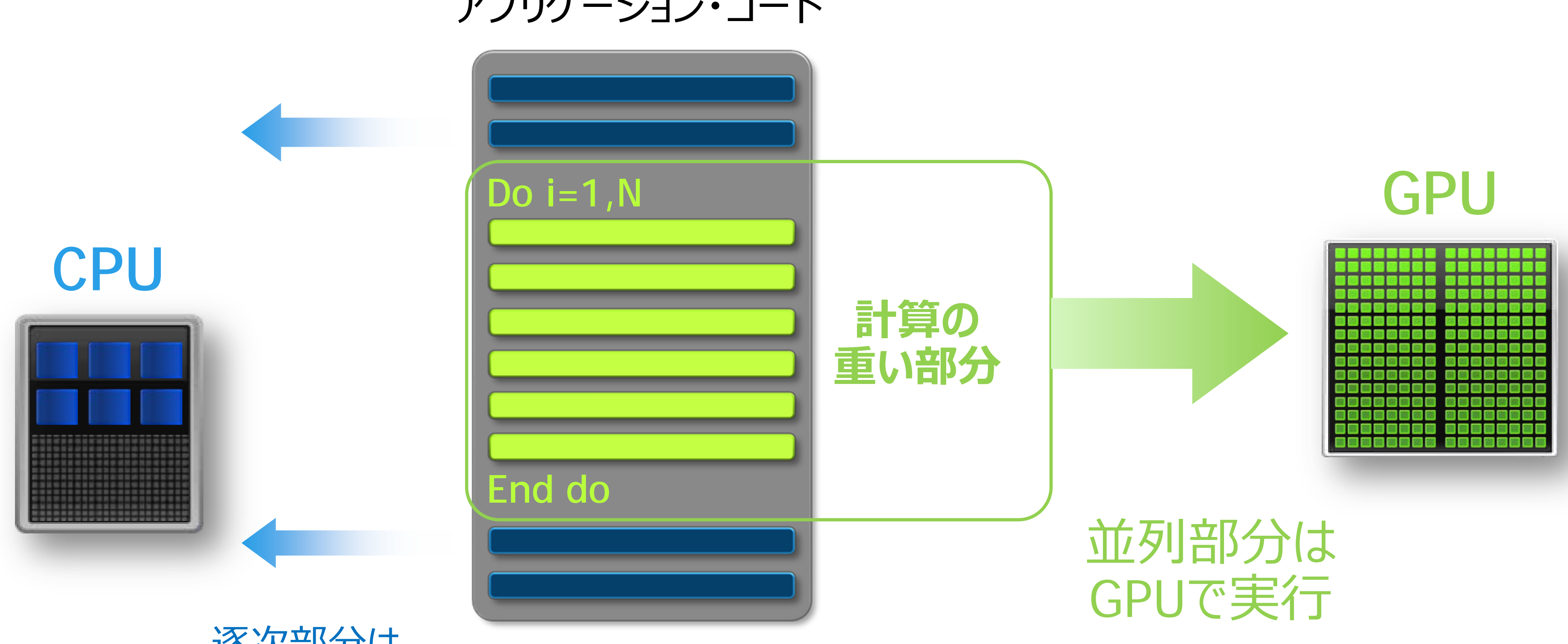

CUDA・OpenACCによる
GPUコンピューティング
Akira Naruse, 23th June. 2016
NVIDIA

GPUコンピューティング

Low latency + High throughput



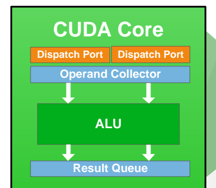
アプリケーション実行



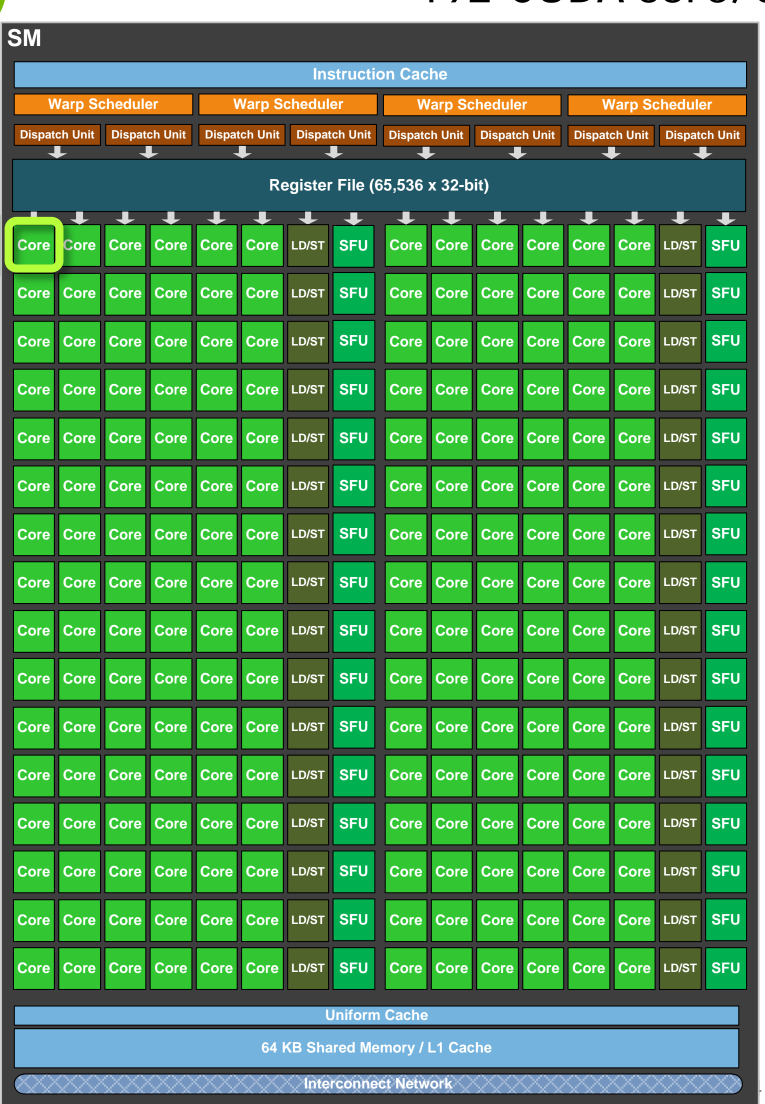
GPUの構造(TESLA K40)

192 CUDA core/SM

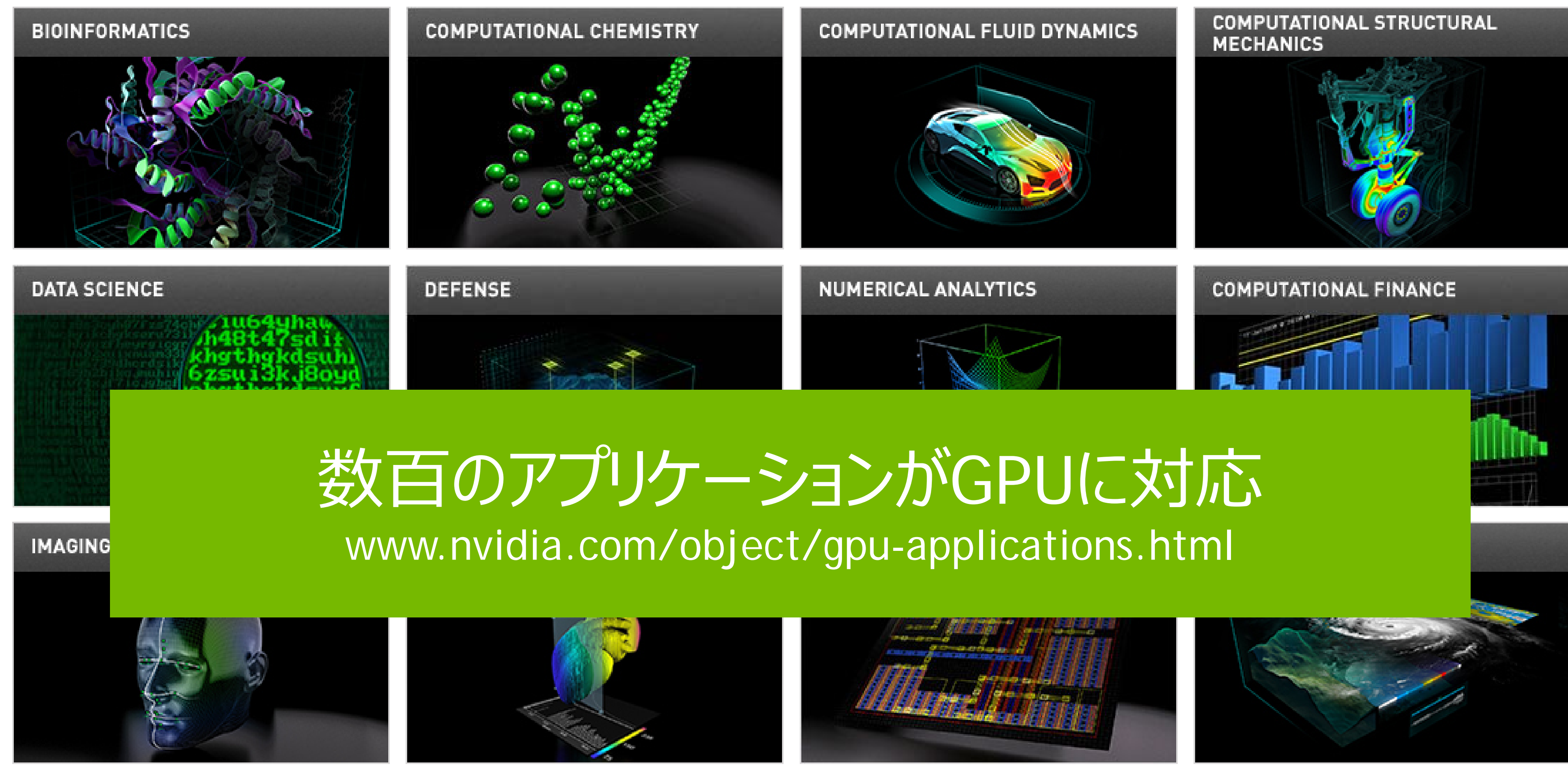
大量のCUDAコア
並列性の抽出が鍵



Tesla K40，15 SM/chip



GPUアプリケーション



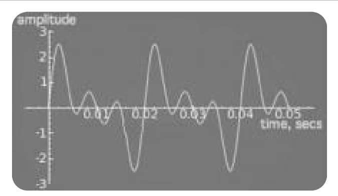
アプリをGPU対応する方法

Application

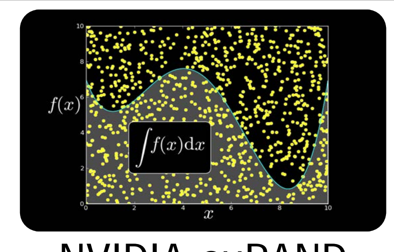
Library	OpenACC	CUDA
GPU対応ライブラリにチェンジ 簡単に開始	既存コードにディレクティブを挿入 簡単に加速	主要処理をCUDAで記述 高い自由度

GPU対応のライブラリ (一部)

NVIDIA cuBLAS

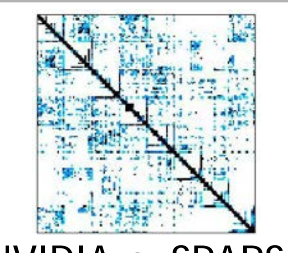


NVIDIA cuFFT



NVIDIA cuRAND

NVIDIA cuDNN



NVIDIA cuSPARSE

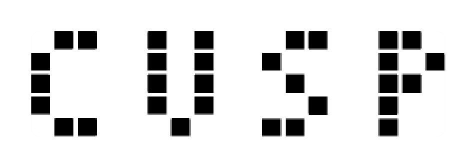
NVIDIA AmgX

GPU Accelerated
Linear Algebra

Vector Signal
Image Processing

IMSL Library

Matrix Algebra on
GPU and Multicore



Sparse Linear
Algebra

C++ STL Features
for CUDA

CUDA計算ライブラリ

高性能な計算ライブラリを提供

- cuDNN　ディープラーニング向けライブラリ
- cuBLAS　BLASライブラリ
- cuFFT　Fast Fourier Transformsライブラリ
- cuRAND　乱数生成ライブラリ
- cuSPARSE　疎行列ライブラリ
- cuSOLVER　Lapackライブラリの一部
- Thrust　並列アルゴリズム C++ STL

CUDAツールキットに標準搭載 (一部、デベロッパー登録必要)

developer.nvidia.com/cuda-downloads

アプリをGPU対応する方法

Application

Library	OpenACC	CUDA
GPU対応ライブラリにチェンジ 簡単に開始	既存コードにディレクティブを挿入 簡単に加速	主要処理をCUDAで記述 高い自由度

SAXPY (Y=A*X+Y)

CPU

```
void saxpy(int n, float a,
           float *x, float *y)
{
  for (int i = 0; i < n; ++i)

}

...
saxpy(N, 3.0, x, y);
...
```

CUDA

```
__global__ void saxpy(int n, float a,
           float *x, float *y)
{
  int i = threadIdx.x + blodkDim.x * blockIdx;
  if (i < n)

}

...

cudaMemcpy(d_x, x, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_y, y, size, cudaMemcpyHostToDevice);
saxpy<<< N/128, 128 >>>(N, 3.0, d_x, d_y);
cudaMemcpy(y, d_y, size, cudaMemcpyDeviceToHost);
...
```

アプリをGPU対応する方法

Application

Library	OpenACC	CUDA
GPU対応ライブラリにチェンジ 簡単に開始	既存コードにディレクティブを挿入 簡単に加速	主要処理をCUDAで記述 高い自由度

SAXPY (Y=A*X+Y)

OpenMP

```
void saxpy(int n,
           float a,
           float *x,
           float *restrict y)
{
#pragma omp parallel for
  for (int i = 0; i < n; ++i)
    y[i] += a*x[i];
}

...
saxpy(N, 3.0, x, y);
...
```

OpenACC

```
void saxpy(int n,
           float a,
           float *x,
           float *restrict y)
{
#pragma acc parallel copy(y[:n]) copyin(x[:n])
  for (int i = 0; i < n; ++i)
    y[i] += a*x[i];
}

...
saxpy(N, 3.0, x, y);
...
```

CUDAプログラミング

- **プログラミングモデル**
- アーキテクチャ
- 性能Tips

GPUコンピューティング

CPU

CPU Memory

PCI

GPU

GPU Memory

- 高スループット指向のプロセッサ
- 分離されたメモリ空間

GPUプログラム

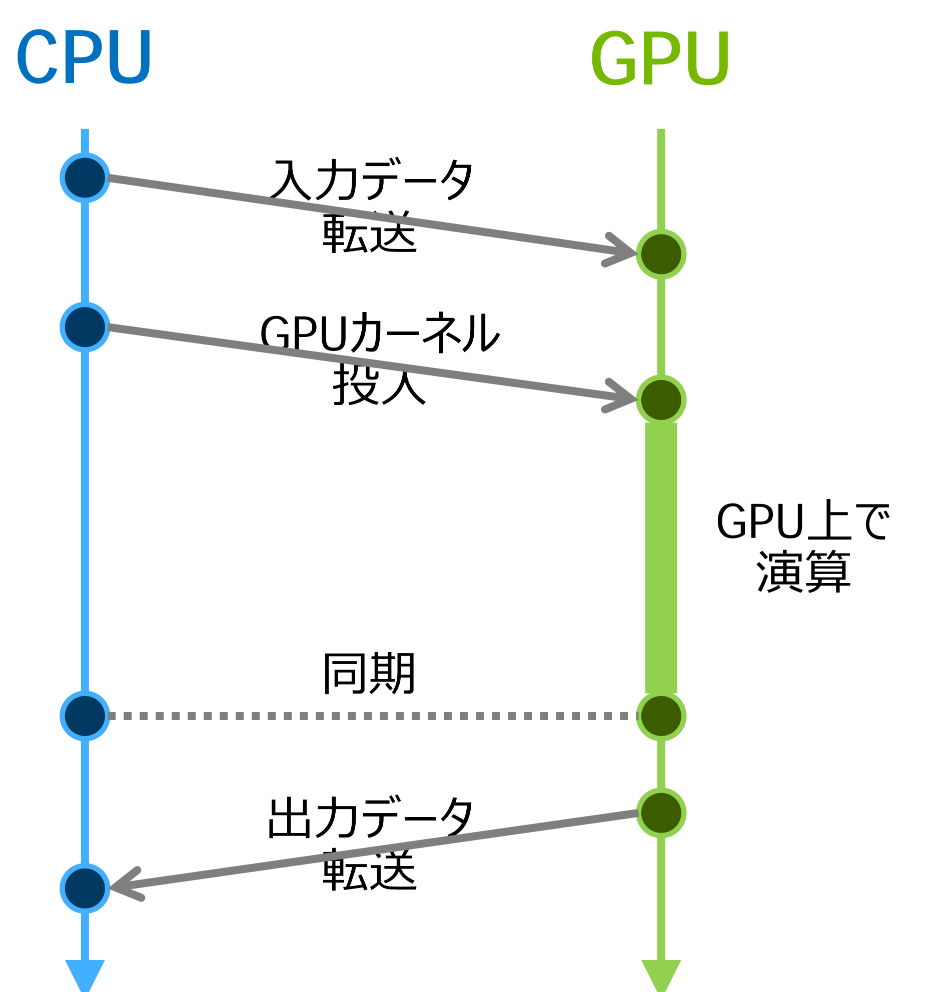
CPU

```
void saxpy(int n, float a,
           float *x, float *y)
{
  for (int i = 0; i < n; ++i)
    y[i] += a*x[i];
}

...
saxpy(N, 3.0, x, y);
...
```

GPU

```
__global__ void saxpy(int n, float a,
                      float *x, float *y)
{
  int i = threadIdx.x + blodkDim.x * blockIdx;
  if (i < n)
    y[i] += a*x[i];
}

...
size_t size = sizeof(float) * N;
cudaMemcpy(d_x, x, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_y, y, size, cudaMemcpyHostToDevice);
saxpy<<< N/128, 128 >>>(N, 3.0, d_x, d_y);
cudaDeviceSynchronize();
cudaMemcpy(y, d_y, size, cudaMemcpyDeviceToHost);
...
```

GPU実行の基本的な流れ



- GPUは、CPUからの制御で動作

- 入力データ: CPUからGPUに転送 (H2D)
- GPUカーネル: CPUから投入
- 出力データ: GPUからCPUに転送 (D2H)

GPUプログラム

CPU

```
void saxpy(int n, float a,
           float *x, float *y)
{
  for (int i = 0; i < n; ++i)
    y[i] += a*x[i];
}

...
saxpy(N, 3.0, x, y);
...
```

GPU

```
__global__ void saxpy(int n, float a,
                      float *x, float *y)
{
  int i = threadIdx.x + blodkDim.x * blockIdx;
  if (i < n)
    y[i] += a*x[i];
}

...
size_t size = sizeof(float) * N;
cudaMemcpy(d_x, x, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_y, y, size, cudaMemcpyHostToDevice);
saxpy<<< N/128, 128 >>>(N, 3.0, d_x, d_y);
cudaDeviceSynchronize();
cudaMemcpy(y, d_y, size, cudaMemcpyDeviceToHost);
...
```

入力データ転送

カーネル起動

同期

出力データ転送

GPUプログラム (Unified Memory)

CPU

```
void saxpy(int n, float a,
           float *x, float *y)
{
  for (int i = 0; i < n; ++i)
    y[i] += a*x[i];
}

...
saxpy(N, 3.0, x,
...
```

GPU

```
__global__ void saxpy(int n, float a,
                      float *x, float *y)
{
  int i = threadIdx.x + blodkDim.x * blockIdx;
  if (i < n)
    y[i] += a*x[i];
}

...
saxpy<<< N/128, 128 >>>(N, 3.0, x, y);
cudaDeviceSynchronize();
...
```

カーネル起動

同期

GPUカーネル

CPU

```
void saxpy(int n, float a,
           float *x, float *y)
{

  for (int i = 0; i < n; ++i)
    y[i] += a*x[i];
}

...
saxpy(N, 3.0, x, y);
...
```

GPU

```
__global__ void saxpy(int n, float a,
           float *x, float *y)
{
  int i = threadIdx.x + blodkDim.x * blockIdx.x;
  if (i < n)
    y[i] += a*x[i];
}

...
saxpy<<< N/128, 128 >>>(N, 3.0, d_x, d_y);
...
```

Global スレッドID

- GPUカーネル: 1つのGPUスレッドの処理内容を記述
  - 基本: 1つのGPUスレッドが、1つの配列要素を担当

Execution Configuration (ブロック数とブロックサイズ)

スレッドID

ブロックID

```
__global__ void saxpy(int n, float a,
                      float *x, float *y)
{
  int i = threadIdx.x + blodkDim.x * blockIdx.x;
  if (i < n)
    y[i] += a*x[i];
}

...
saxpy<<< N/128, 128 >>>(N, 3.0, d_x, d_y);
...
```

ブロックサイズ

ブロック数

ブロックサイズ

ブロック数 x ブロックサイズ = 配列要素数

スレッド階層
(スレッド、ブロック、グリッド)

y[i] = a*x[i] + y[i]

x[]
0 127 128 255 256 383 384

y[]
0 127 128 255 256 383 384

スレッド
(local)

0 127 ブロック0
0 127 ブロック1
0 127 ブロック2
0

グリッド

- ブロックサイズ(スレッド数/ブロック)は、カーネル毎に設定可能
  - 推奨: 128 or 256 スレッド

Execution Configuration (ブロック数とブロックサイズ)

```
__global__ void saxpy(int n, float a,
                      float *x, float *y)
{
  int i = threadIdx.x + blodkDim.x * blockIdx.x;
  if (i < n)
    y[i] += a*x[i];
}

...
saxpy<<< N/256, 256 >>>(N, 3.0, d_x, d_y);
...
```

ブロック数

ブロックサイズ

ブロック数 x ブロックサイズ = 配列要素数

2D配列のGPUカーネル例

```
__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N])
{
    int i = threadIdx.x + blodkDim.x * blockIdx.x;
    int j = threadIdx.y + blodkDim.y * blockIdy.y;
    if ( i < N && j < N )
        C[i][i] = A[i][j] + B[i][j];
}

...
dim3 sizeBlock( 64,  4 );
dim3 numBlocks( N/sizeBlock.x, N/sizeBlock.y );
MatAdd<<< numBlocks, sizeBlock >>>(A, B, C);
...
```

GlobalスレッドID (x)

GlobalスレッドID (y)

ブロックサイズ (x,y)

ブロック数 (x,y)

- ブロックサイズ(ブロック形状)は、1D～3Dで表現可能

ブロック・マッピング、スレッド・マッピング

```
dim3 sizeBlock(16,16)
```

(0,0)

(0,0)	(1,0)	(2,0)
(0,1)	(1,1)	(2,1)
(0,2)	(1,2)	(2,2)

(15,15)

```
dim3 sizeBlock(32,8)
```

(0,0)

(0,0)	(1,0)
(0,1)	(1,1)
(0,2)	(1,2)
(0,3)	(1,3)
(0,4)	(1,4)

(31,7)

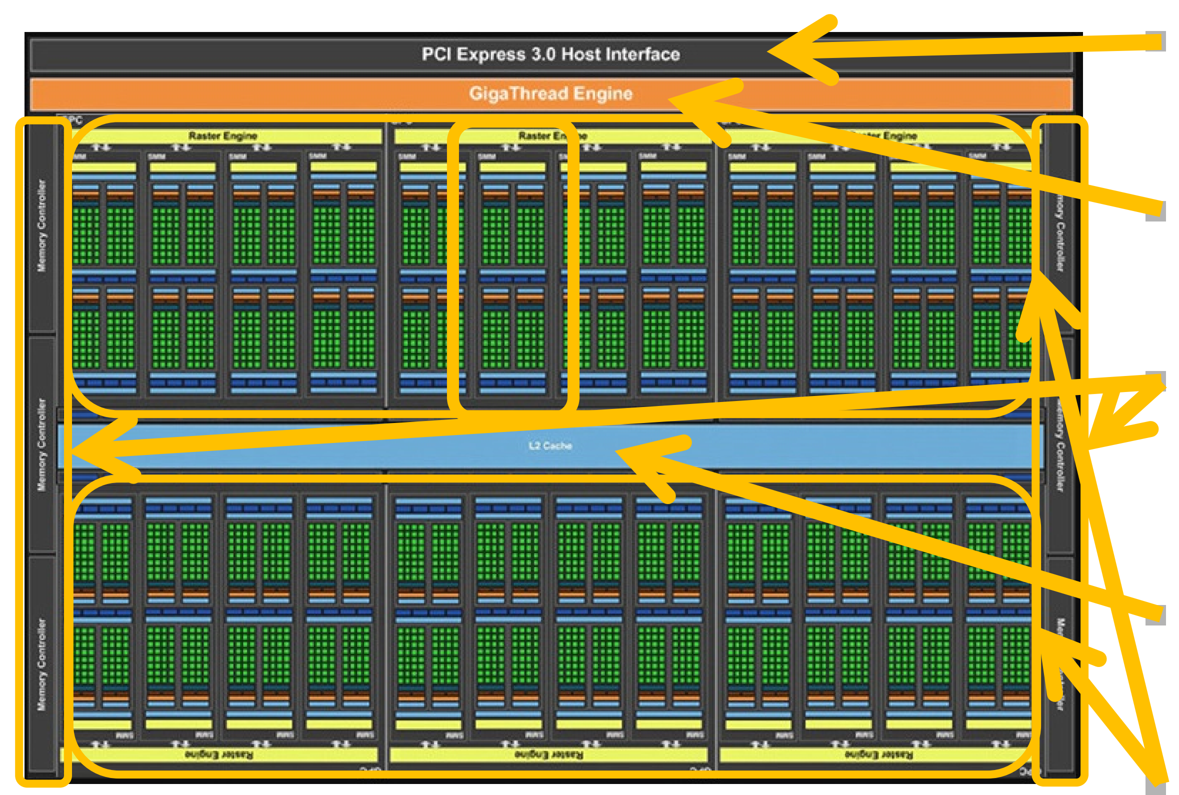
ブロックID(blockIdx)

スレッドID(threadIdx)

CUDAプログラミング

- プログラミングモデル
- **アーキテクチャ**
- 性能Tips

GPUアーキテクチャ概要



Maxwell GM200

PCI I/F

- ホスト接続インタフェース

Giga Thread Engine

- SMに処理を割り振るスケジューラ

DRAM I/F (GDDR5, 384-bit)

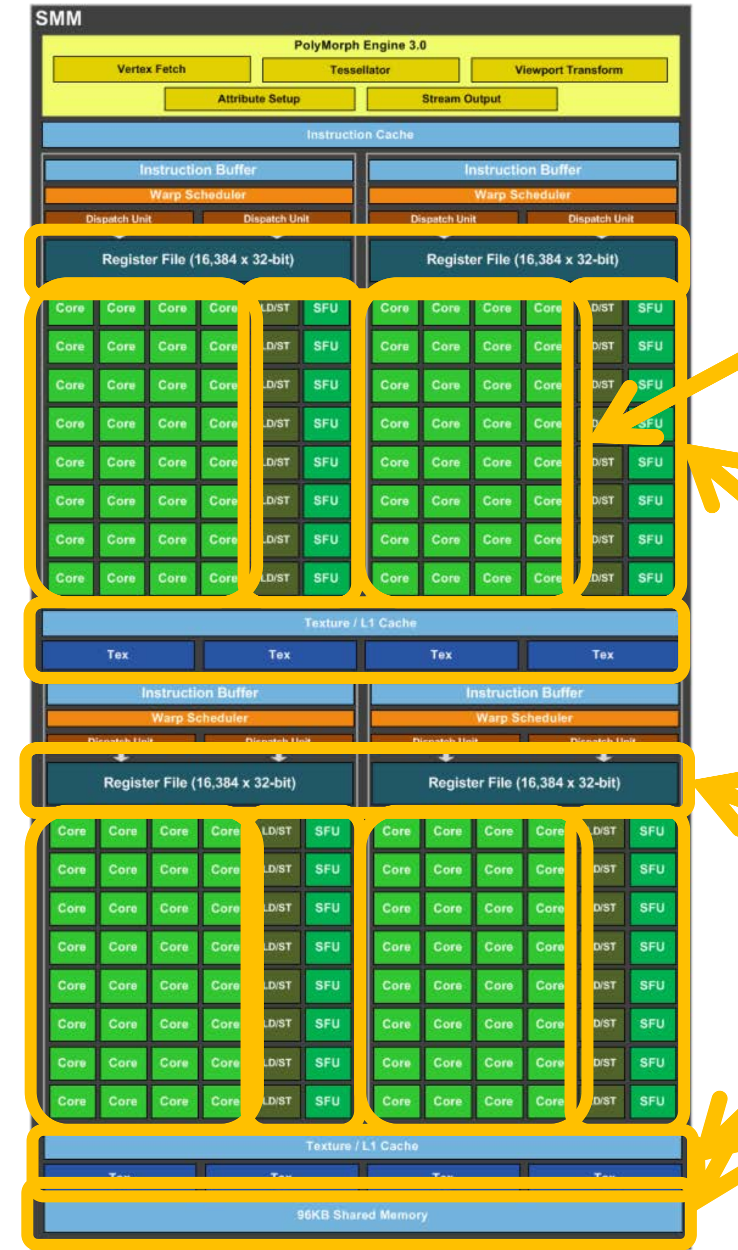
- 全SM、PCI I/Fからアクセス可能なメモリ (デバイスメモリ, フレームバッファ)

L2 cache (3MB)

- 全SMからアクセス可能なR/Wキャッシュ

SM (Streaming Multiprocessor)

- 「並列」プロセッサ、Maxwell:最多24



Maxwell
GM200

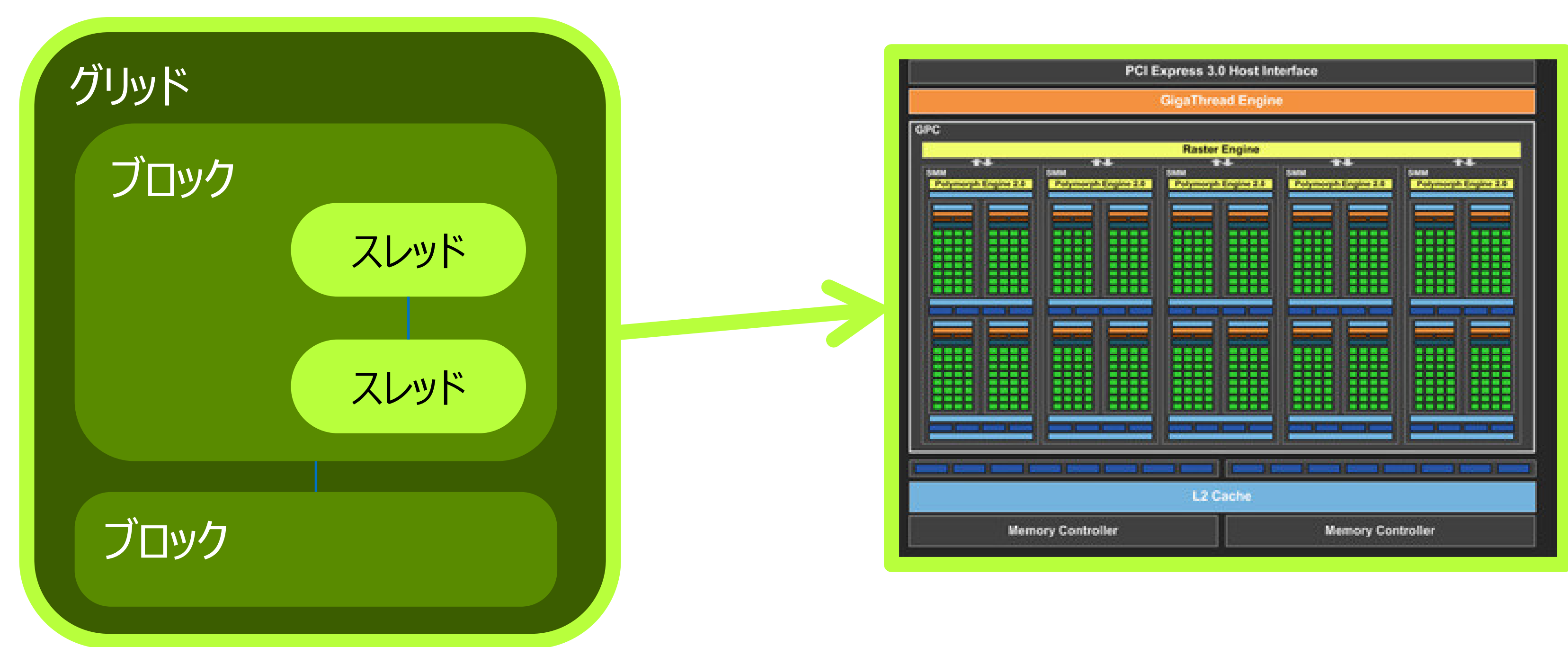
SM
(Stream Multi-Processor)

- CUDAコア
  - GPUスレッドはこの上で動作
  - Maxwell: 128個
- Other units
  - LD/ST, SFU, etc
- レジスタ(32bit): 64K個
- 共有メモリ: 96KB
- Tex/L1キャッシュ

 NVIDIA.

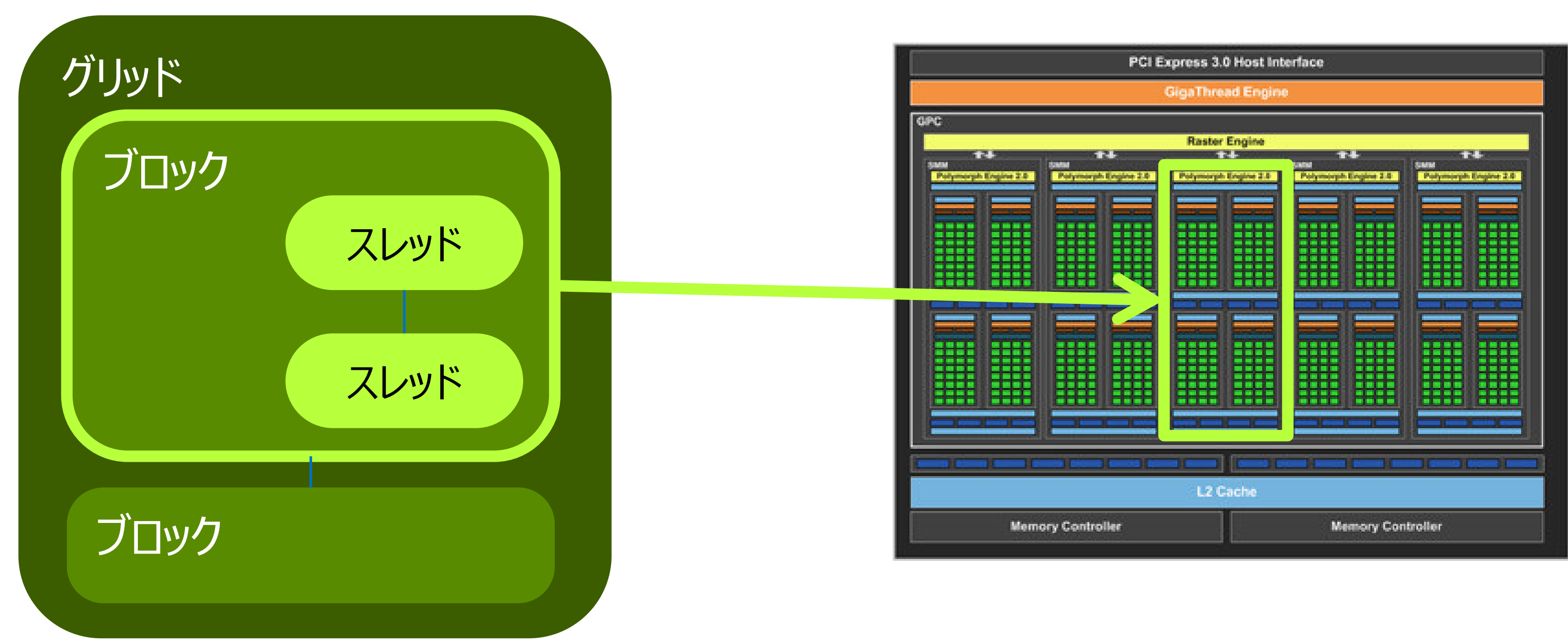
GPUカーネル実行の流れ

- CPUが、GPUに、グリッドを投入
  - 具体的な投入先は、Giga Thread Engine



GPUカーネル実行の流れ

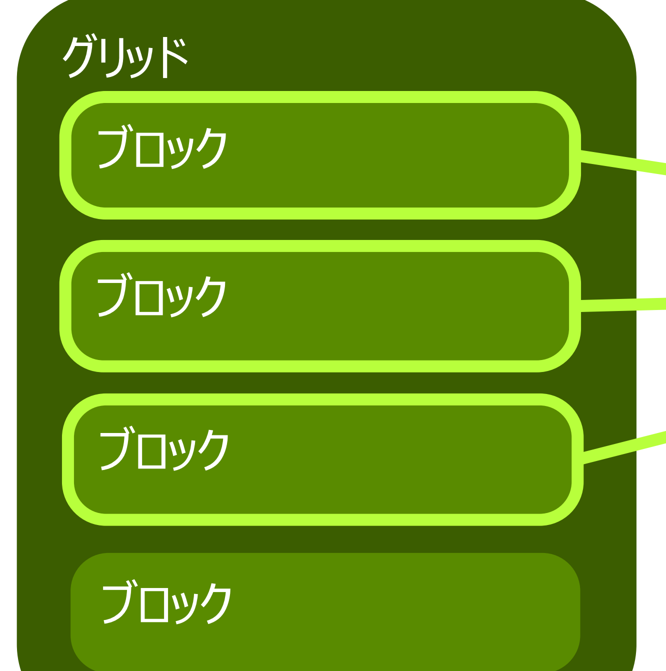
- Giga Thread Engine(GTE)が、SMに、ブロックを投入
  - GTEは、ブロックスケジューラ
  - グリッドをブロックに分解して、ブロックを、空いているSMに割当てる

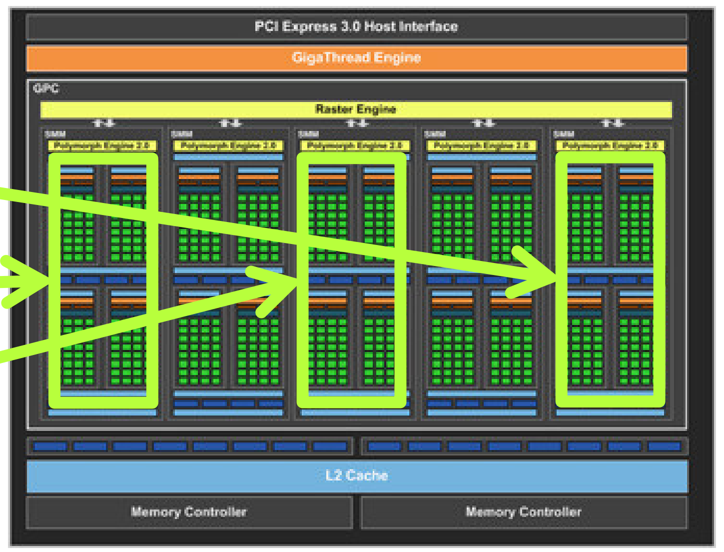


NVIDIA

ブロックをSMに割り当て

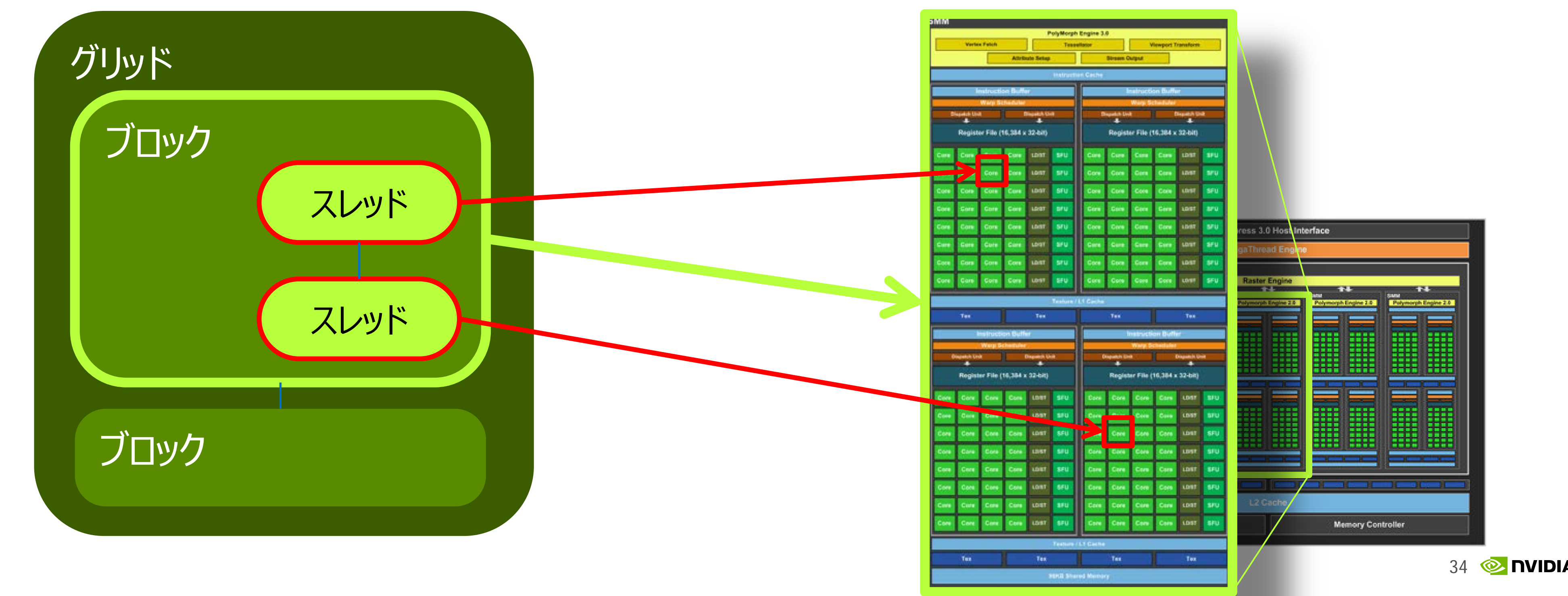
- 各ブロックは、互いに独立に実行
  - ブロック間では同期しない、実行順序の保証なし
- 1つのブロックは複数SMにまたがらない
  - 1つのSMに、複数ブロックが割当てられることはある





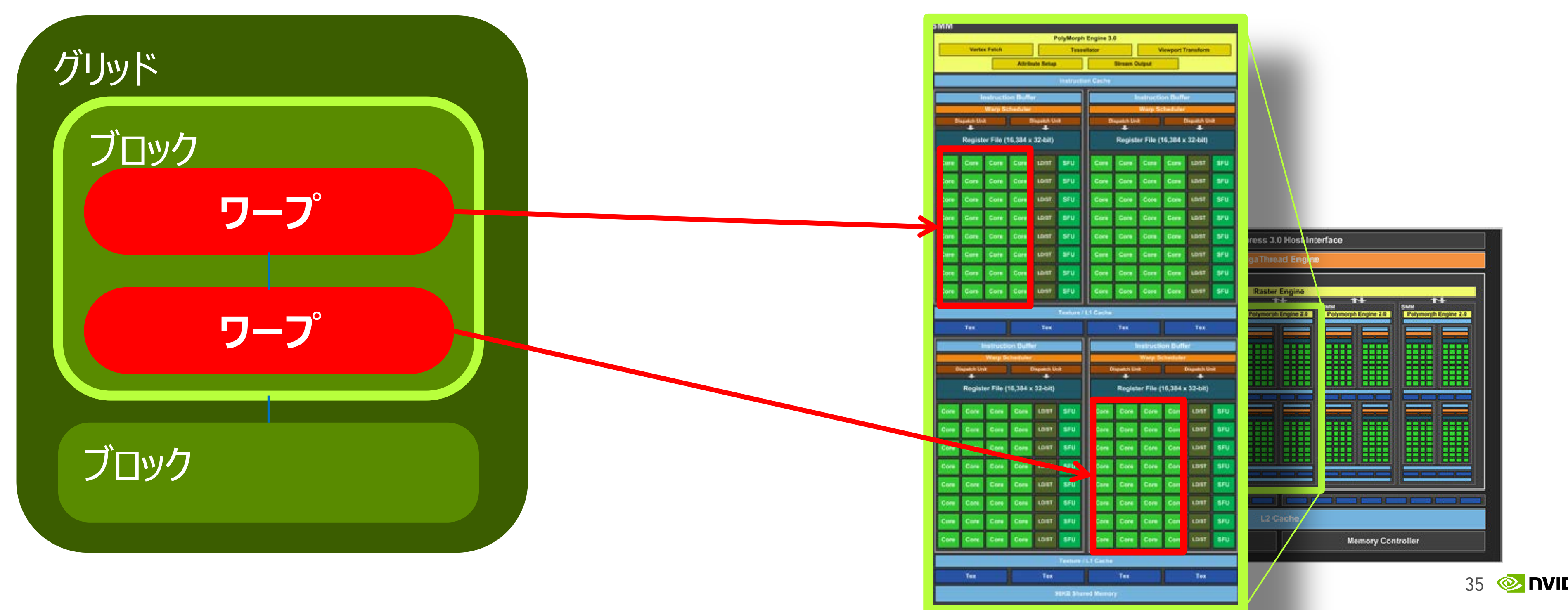
GPUカーネル実行の流れ

- ~~SM内のスケジューラが、スレッドをCUDAコアに投入~~



GPUカーネル実行の流れ

- SM内のスケジューラが、**ワープ**をCUDAコアに投入
  - ワープ: 32スレッドの塊
  - ブロックをワープに分割、実行可能なワープを、空CUDAコアに割当てる

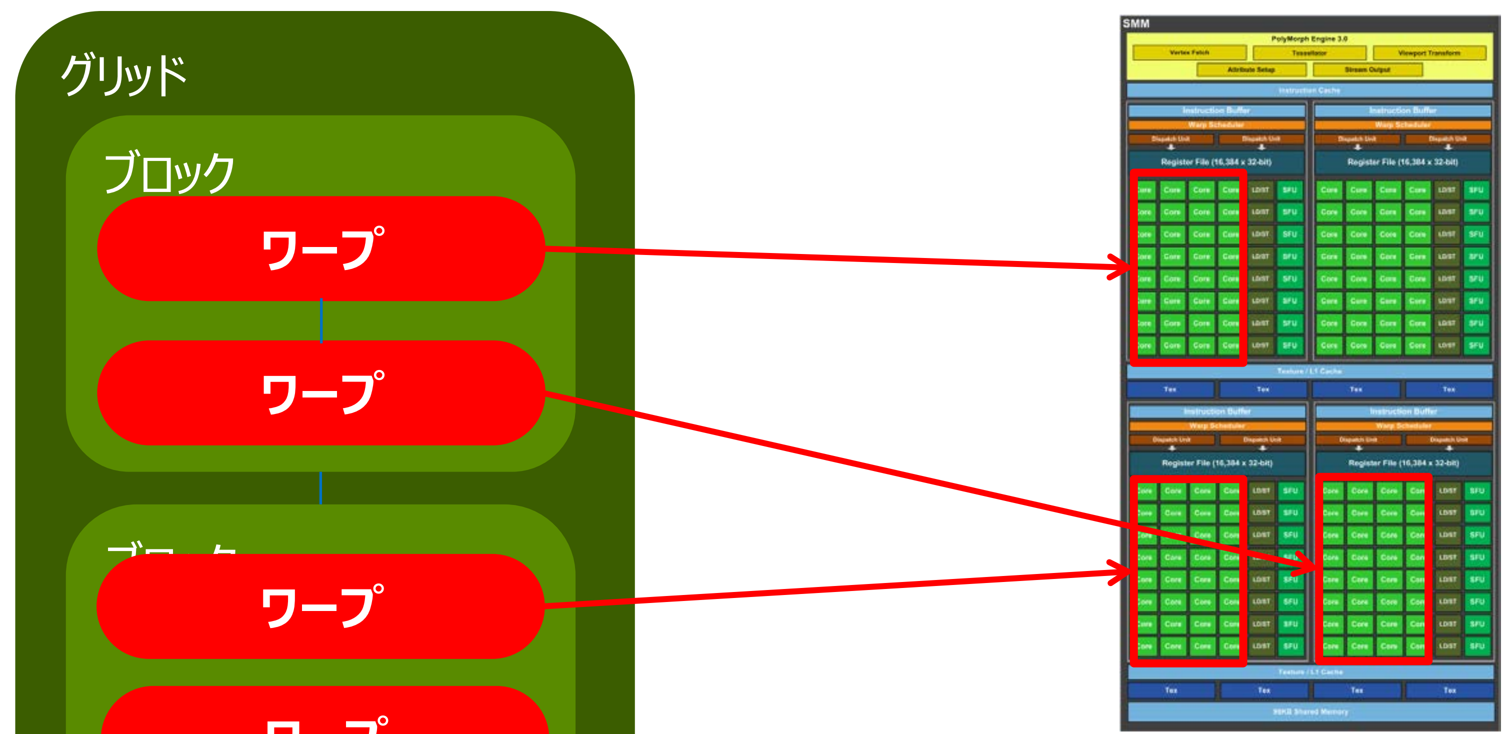


NVIDIA

ワープのCUDAコアへの割り当て

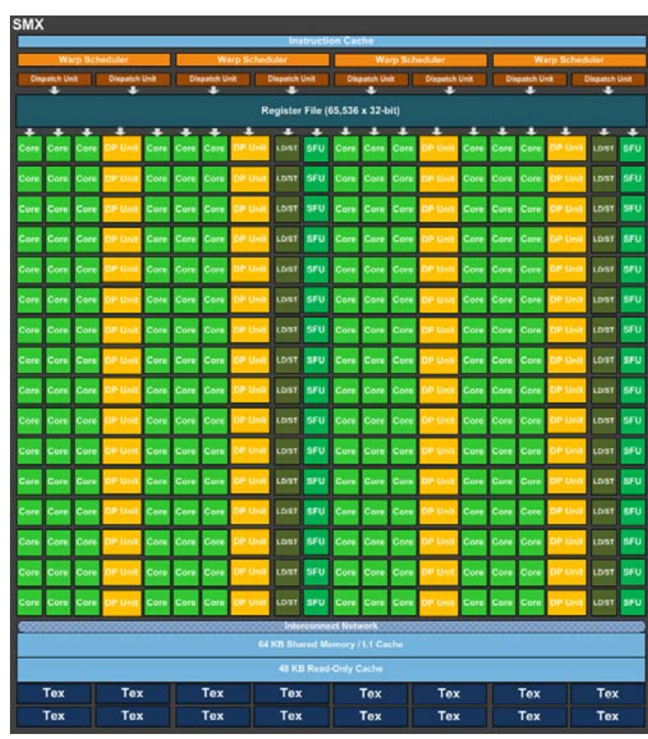
- ワープ内の32スレッドは、同じ命令を同期して実行
- 各ワープは、互いに独立して実行
  - 同じブロック内のワープは、明示的に同期可能(__syncthreads())

SIMT
(Single Instruction Multiple Threads)



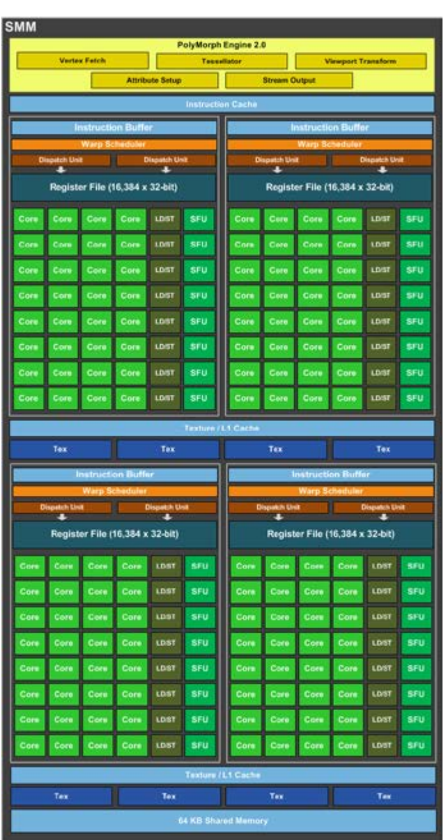
NVIDIA.

GPUアーキの変化を問題としないプログラミングモデル



Kepler, CC 3.5

192 cores /SM



Maxwell, CC 5.0

128 cores /SM

Pascal, CC 6.0

64 cores /SM

CUDAプログラミング

- プログラミングモデル
- アーキテクチャ
- **性能Tips**

リソース使用率 (Occupancy)

SMの利用効率を上げる

≒ SMに割当て可能なスレッド数を、
上限に近づける

- レジスタ使用量(/スレッド)
  - できる限り減らす
    - DP(64bit)は、2レジスタ消費
    - レジスタ割り当て単位は8個
  - レジスタ使用量と、割当て可能なスレッド数の関係
    - 32レジスタ: 2048(100%), 64レジスタ: 1024(50%)
    - 128レジスタ:512(25%), 256レジスタ: 256(12.5%)

CUDAコア数: 128
最大スレッド数: 2048
最大ブロック数: 32
共有メモリ: 96KB
レジスタ数(32-bit): 64K個

リソース量/SM (GM200)

リソース使用率 (Occupancy)

SMの利用効率を上げる

≒ SMに割当て可能なスレッド数を、
上限に近づける

- スレッド数(/ブロック)
  - 64以上にする
  - 64未満だと最大ブロック数がネックになる
- 共有メモリ使用量(/ブロック)
  - できる限り減らす
  - 共有メモリ使用量と、割当て可能なブロック数の関係
    - 48KB:2ブロック， 12KB:8ブロック， 3KB:32ブロック

CUDAコア数: 128
最大スレッド数: 2048
最大ブロック数: 32
共有メモリ: 96KB
レジスタ数(32-bit): 64K個

リソース量/SM (GM200)

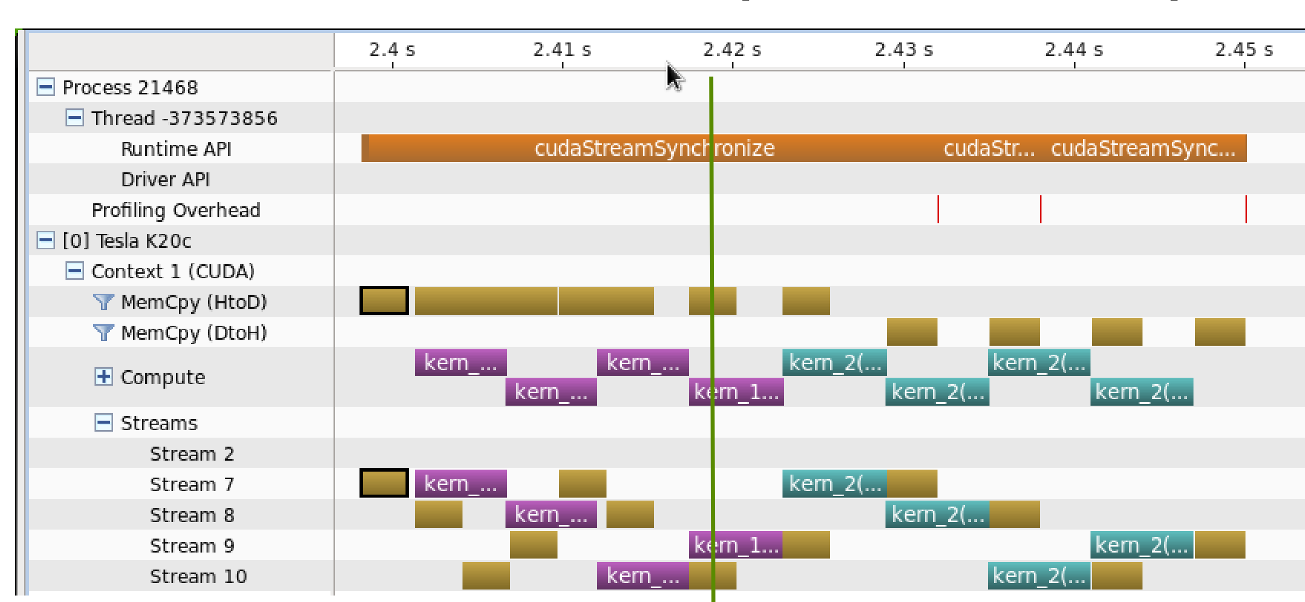
リソース使用率 (Occupancy)

空き時間を埋める

(*) 操作: GPUカーネル, データ転送

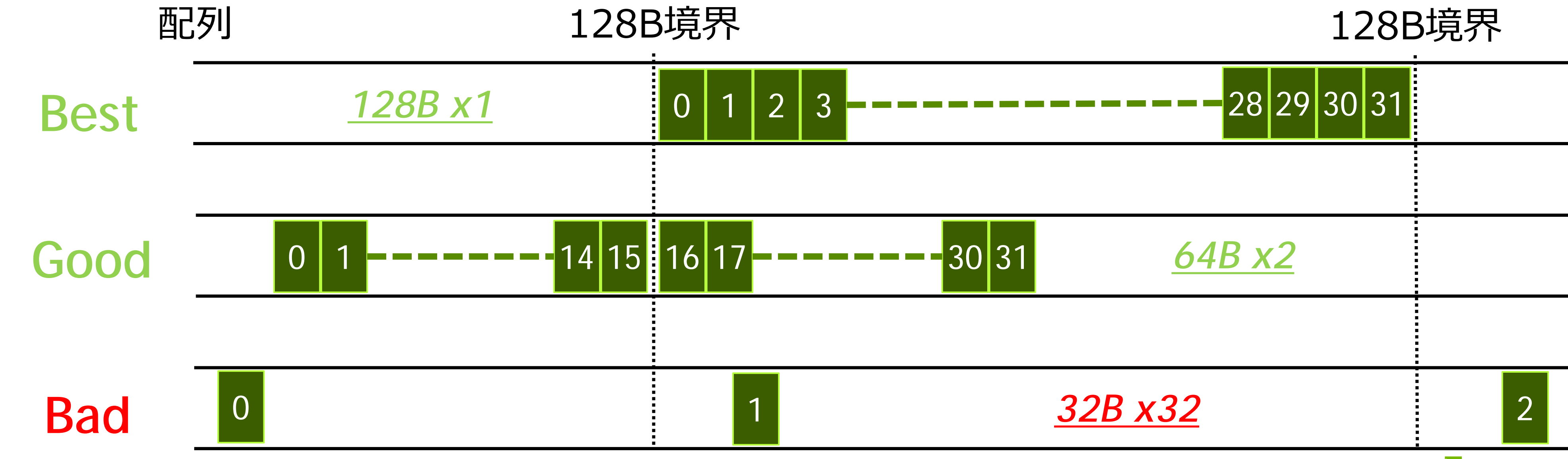
- CUDAストリーム (≒キュー)
  - 同じCUDAストリームに投入した操作: 投入順に実行
  - 別のCUDAストリームに投入した操作: 非同期に実行 (オーバラップ実行)

[CUDAストリームの効果例]
GPUカーネルとデータ転送が
オーバラップして
同時に実行されている



デバイスメモリへのアクセスは、まとめて

- コアレス・アクセス
  - 32スレッド(ワープ)のロード/ストアをまとめて、メモリトランザクションを発行
    - トランザクションサイズ: 32B, 64B or 128B
  - トランザクション数は、少ないほど良い



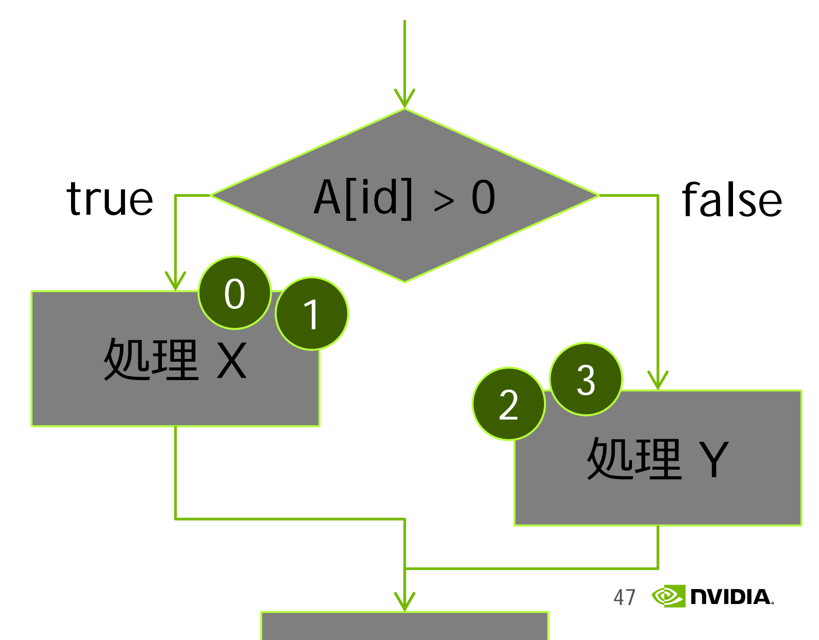
NVIDIA.

分岐を減らす

- ワープ内のスレッドが別パスを選択すると遅くなる
  - ワープ内のスレッドは、命令を共有 (SIMT)
  - ワープ内のスレッドが選んだ全パスの命令を実行
  - あるパスの命令を実行中、そのパスにいないスレッドはinactive状態

Path divergence

- Path divergenceを減らす
  - できる限り、同ワープ内のスレッドは同じパスを選択させる



NVIDIA.

OPENACCプログラミング

- 概要紹介
- プログラムのOpenACC化
- OpenACC化事例

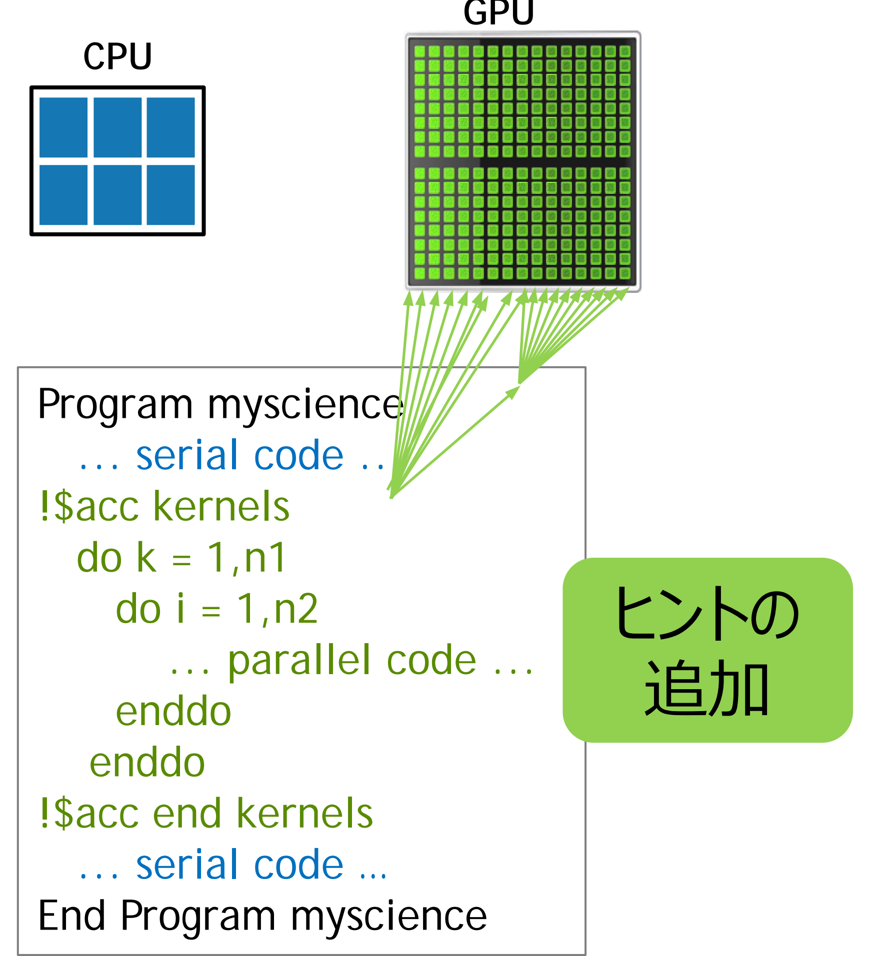
NVIDIA.

アプリをGPU対応する方法

Application

Library	OpenACC	CUDA
GPU対応ライブラリにチェンジ 簡単に開始	既存コードにディレクティブを挿入 簡単に加速	主要処理をCUDAで記述 高い自由度

OPENACC



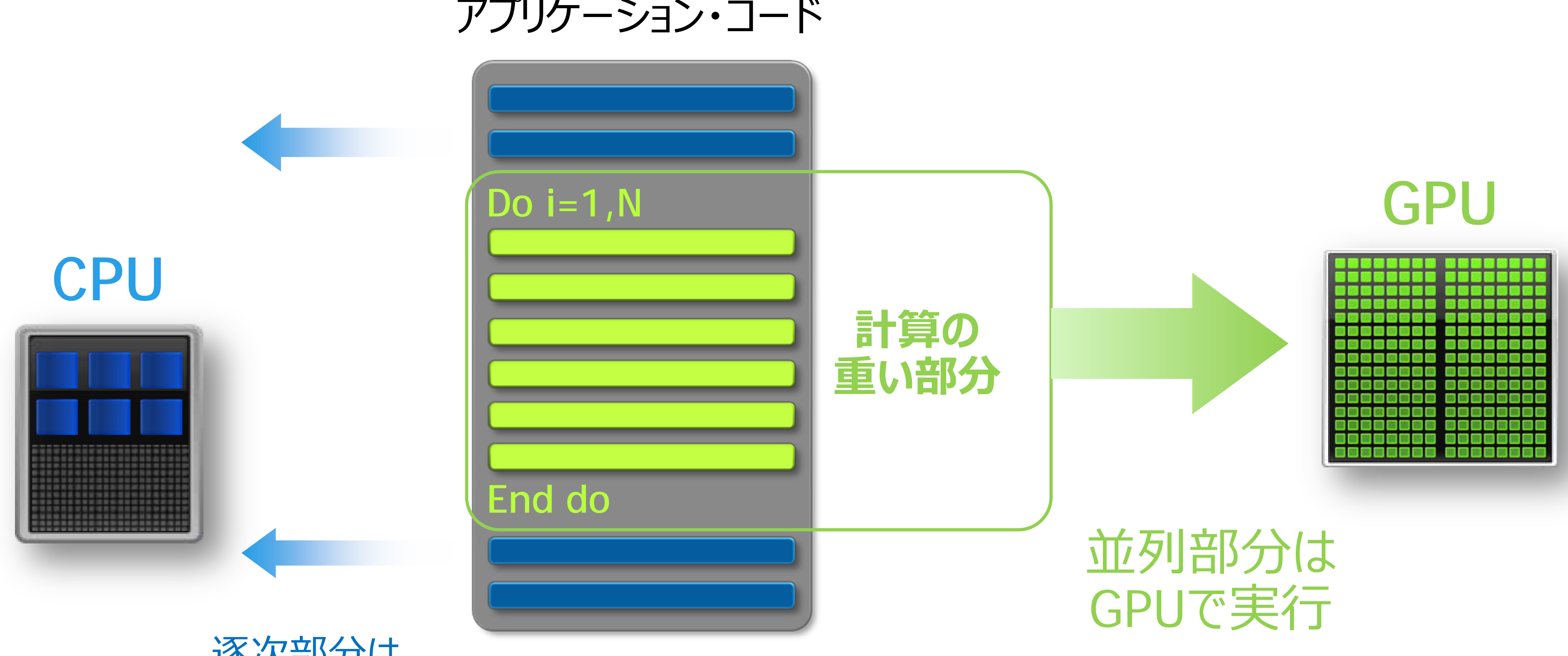
既存のC/Fortranコード

簡単: 既存のコードに
コンパイラへのヒントを追加

強力: そこそこの労力で、コンパイラが
コードを自動で並列化

オープン: 複数コンパイラベンダが、
複数アクセラレータをサポート
NVIDIA, AMD, Intel(予定)

アプリケーション実行



NVIDIA.

SAXPY (Y=A*X+Y)

CPU

```
void saxpy(int n, float a,
           float *x, float *y)
{
  for (int i = 0; i < n; ++i)
    y[i] += a*x[i];
}

...
saxpy(N, 3.0, x, y);
...
```

CUDA

```
__global__ void saxpy(int n, float a,
           float *x, float *y)
{
  int i = threadIdx.x + blodkDim.x * blockIdx;
  if (i < n)
    y[i] += a*x[i];
}

...

cudaMemcpy(d_x, x, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_y, y, size, cudaMemcpyHostToDevice);
saxpy<<< N/128, 128 >>>(N, 3.0, d_x, d_y);
cudaMemcpy(y, d_y, size, cudaMemcpyDeviceToHost);
...
```

SAXPY (y=a*x+y)

OpenMP

```
void saxpy(int n,
           float a,
           float *x,
           float *restrict y)
{
#pragma omp parallel for
  for (int i = 0; i < n; ++i)
    y[i] += a*x[i];
}

...
saxpy(N, 3.0, x, y);
...
```

OpenACC

```
void saxpy(int n,
           float a,
           float *x,
           float *restrict y)
{
#pragma acc parallel copy(y[:n]) copyin(x[:n])
  for (int i = 0; i < n; ++i)
    y[i] += a*x[i];
}

...
saxpy(N, 3.0, x, y);
...
```

- omp → acc
- データの移動

SAXPY (y=a*x+y, FORTRAN)

OpenMP

```
subroutine saxpy(n, a, X, Y)
  real :: a, X(:), Y(:)
  integer :: n, i

  !$omp parallel do
  do i=1,n
    Y(i) = a*X(i)+Y(i)
  enddo
  !$omp end parallel do
end subroutine saxpy


...
call saxpy(N, 3.0, x, y)
...
```

OpenACC

```
subroutine saxpy(n, a, X, Y)
  real :: a, Y(:), Y(:)
  integer :: n, i

  !$acc parallel copy(Y(:)) copyin(X(:))
  do i=1,n
    Y(i) = a*X(i)+Y(i)
  enddo
  !$acc end parallel
end subroutine saxpy


...
call saxpy(N, 3.0, x, y)
...
```

- FORTRANも同様

簡単にコンパイル

OpenMP / OpenACC

```
void saxpy(int n, float a,
           float *x,
           float *restrict y)
```

```
$ pgcc -Minfo -acc saxpy.c
saxpy:
     16, Generating present_or_copy(y[:n])
         Generating present_or_copyin(x[:n])
         Generating Tesla code
     19, Loop is parallelizable
         Accelerator kernel generated
         19, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
```

```
...
```

簡単に実行

OpenMP / OpenACC

```
void saxpy(int n, float a,
           float *x,
           float *restrict y)
```

```
$ nvprof ./a.out
==10302== NVPROF is profiling process 10302, command: ./a.out
==10302== Profiling application: ./a.out
==10302== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 62.95%  3.0358ms         2  1.5179ms  1.5172ms  1.5186ms  [CUDA memcpy HtoD]
 31.48%  1.5181ms         1  1.5181ms  1.5181ms  1.5181ms  [CUDA memcpy DtoH]
  5.56%  268.31us         1  268.31us  268.31us  268.31us  saxpy_19_gpu
```

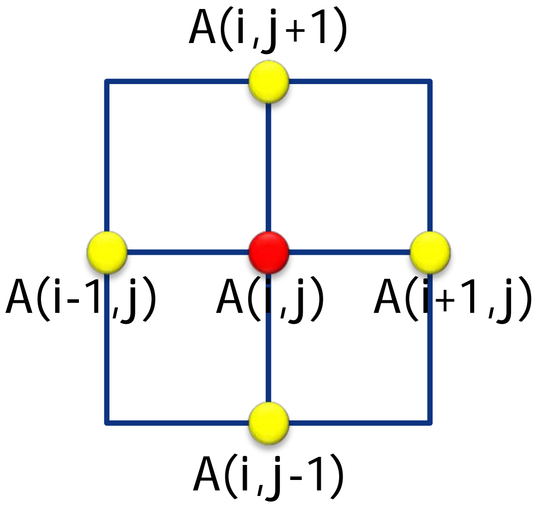
```
...
```

OPENACCプログラミング

- 概要紹介
- プログラムのOpenACC化
- OpenACC化事例

事例: ヤコビ反復法

```
while ( error > tol ) {
  error = 0.0;

  for (int j = 1; j < N-1; j++) {
    for (int i = 1; i < M-1; i++) {
      Anew[j][i] = (A[j][i+1] + A[j][i-1] +
                    A[j-1][i] + A[j+1][i]) * 0.25;
      error = max(error, abs(Anew[j][i] - A[j][i]));
    }
  }

  for (int j = 1; j < N-1; j++) {
    for (int i = 1; i < M-1; i++) {
      A[j][i] = Anew[j][i];
    }
  }
}
```



NVIDIA.

並列領域 (kernels construct)

```
while ( error > tol ) {
  error = 0.0;

  #pragma acc kernels
  for (int j = 1; j < N-1; j++) {
    for (int i = 1; i < M-1; i++) {
      Anew[j][i] = (A[j][i+1] + A[j][i-1] +
                    A[j-1][i] + A[j+1][i]) * 0.25;
      error = max(error, abs(Anew[j][i] - A[j][i]);
    }
  }

  #pragma acc kernels
  for (int j = 1; j < N-1; j++) {
    for (int i = 1; i < M-1; i++) {
      A[j][i] = Anew[j][i];
    }
  }
}
```

- 並列領域の指定
  - Parallels と Kernels
- Parallels
  - OpenMPと親和性
  - 開発者主体
- Kernels
  - 複数kernelの生成
  - コンパイラ主体

[PGI tips] コンパイラメッセージ

```
$ pgcc -acc -Minfo=accel jacobi.c
jacobi:
     44, Generating copyout(Anew[1:4094][1:4094])
         Generating copyin(A[:][:])
         Generating Tesla code
     45, Loop is parallelizable
     46, Loop is parallelizable
         Accelerator kernel generated
         45, #pragma acc loop gang /* blockIdx.y */
         46, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
         49, Max reduction generated for error
```

並列領域 (kernels construct)

```
while ( error > tol ) {
  error = 0.0;

  #pragma acc kernels
  for (int j = 1; j < N-1; j++) {
    for (int i = 1; i < M-1; i++) {
      Anew[j][i] = (A[j][i+1] + A[j][i-1] +
```

```
}
```

- 並列領域の指定
  - Parallels と Kernels

```
$ pgcc -Minfo=acc -acc jacobi.c
jacobi:
     59, Generating present_or_copyout(Anew[1:4094][1:4094])
         Generating present_or_copyin(A[:][:])
         Generating Tesla code
     61, Loop is parallelizable
     63, Loop is parallelizable
         Accelerator kernel generated
         61, #pragma acc loop gang /* blockIdx.y */
         63, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
             Max reduction generated for error
```

データ転送(data clause)

```
while ( error > tol ) {
  error = 0.0;

  #pragma acc kernels
  for (int j = 1; j < N-1; j++) {
    for (int i = 1; i < M-1; i++) {
      Anew[j][i] = (A[j][i+1] + A[j][i-1] +
```

```
$ pgcc -Minfo=acc -acc jacobi.c
jacobi:
     59, Generating present_or_copyout(Anew[1:4094][1:4094])
         Generating present_or_copyin(A[:][:])
         Generating Tesla code
     61, Loop is parallelizable
     63, Loop is parallelizable
         Accelerator kernel generated
         61, #pragma acc loop gang /* blockIdx.y */
         63, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
             Max reduction generated for error
```

- 並列領域の指定
  - Parallels と Kernels

データ転送 (DATA CLAUSE)

```
while ( error > tol ) {
  error = 0.0;

  #pragma acc kernels \
      pcopyout(Anew[1:4094][1:4094]) pcopyin(A[:][:])
  for (int j = 1; j < N-1; j++) {
    for (int i = 1; i < M-1; i++) {
      Anew[j][i] = (A[j][i+1] + A[j][i-1] +
                    A[j-1][i] + A[j+1][i]) * 0.25;
      error = max(error, abs(Anew[j][i] - A[j][i]));
    }
  }

  #pragma acc kernels \
     pcopyout(A[1:4094][1:4094]) pcopyin(Anew[1:4094][1:4094])
  for (int j = 1; j < N-1; j++) {
    for (int i = 1; i < M-1; i++) {
      A[j][i] = Anew[j][i];
    }
  }
}
```

- copyin (Host→GPU)
- copyout (Host←GPU)
- copy
- create
- present

- pcopyin
- pcopyout
- pcopy
- pcreate

配列形状の指定

- 配列は、全要素でなく、一部だけ指定して転送することも可能
  - 注意: C/C++ と Fortranでは指定方法が異なる

- C/C++: array[start : size]

```
float Anew[4096][4096]
pcopyout( Anew[1:4094][1:4094]) pcopyin( A[:][:]) )
```

- Fortran: array(start : end)

```
real Anew(4096,4096)
pcopyout( Anew(2:4095, 2:4095) ) pcopyin( A(:,:) )
```

データ転送 (data clause)

```
while ( error > tol ) {
  error = 0.0;

  #pragma acc kernels \
      pcopy(Anew[:][:]) pcopyin(A[:][:])
  for (int j = 1; j < N-1; j++) {
    for (int i = 1; i < M-1; i++) {
      Anew[j][i] = (A[j][i+1] + A[j][i-1] +
                    A[j-1][i] + A[j+1][i]) * 0.25;
      error = max(error, abs(Anew[j][i] - A[j][i]);
    }
  }

  #pragma acc kernels \
      pcopy(A[:][:]) pcopyin(Anew[:][:])
  for (int j = 1; j < N-1; j++) {
    for (int i = 1; i < M-1; i++) {
      A[j][i] = Anew[j][i];
    }
  }
}
```

- copyin (Host→GPU)
- copyout (Host←GPU)
- copy
- create
- present

- pcopyin
- pcopyout
- pcopy
- pcreate

[PGI tips] PGI_ACC_TIME

統計データ

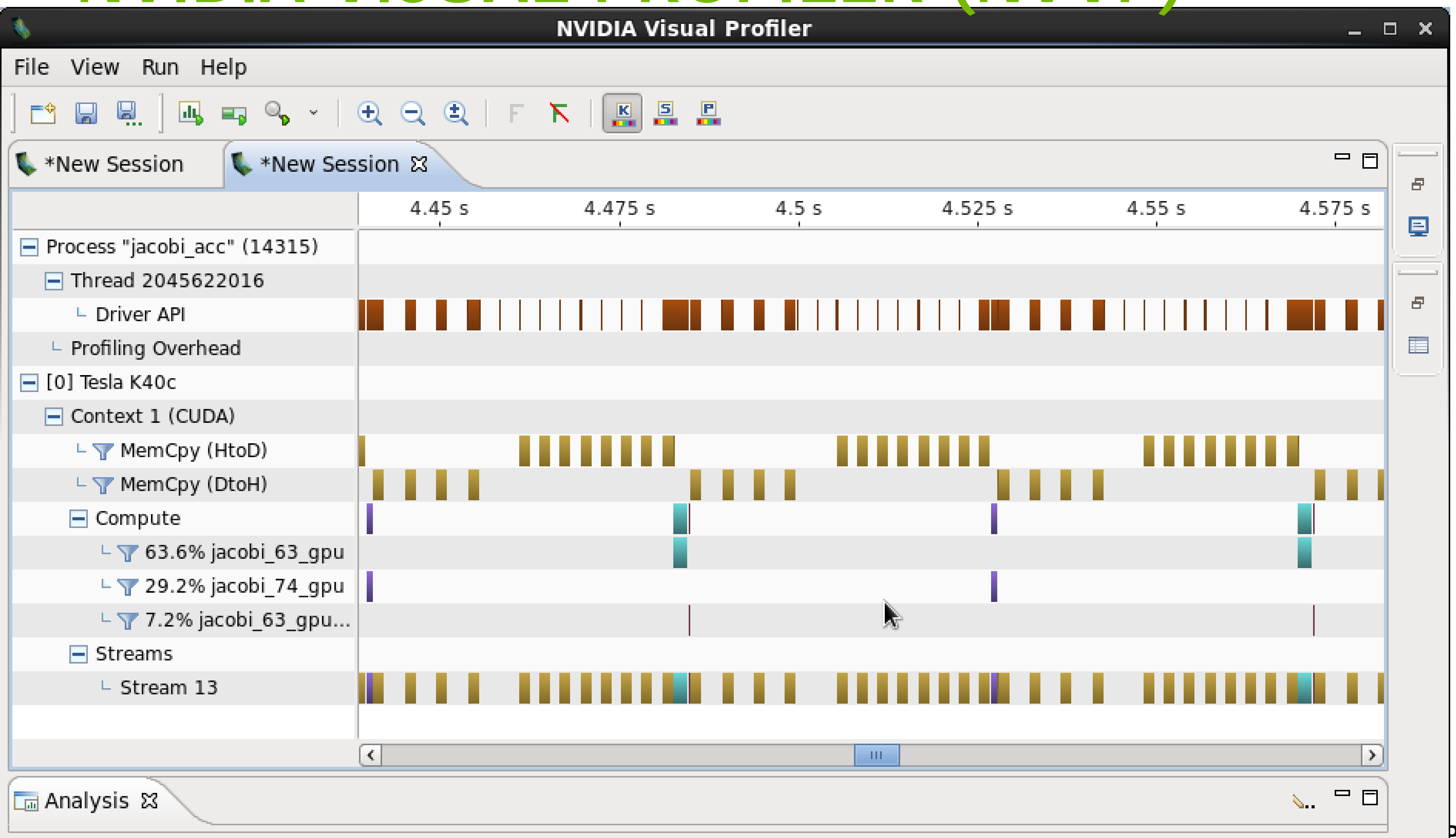
```
$ PGI_ACC_TIME=1 ./a.out
Accelerator Kernel Timing data
/home/anaruse/src/OpenACC/jacobi/C/task1-solution/jacobi.c
  jacobi  NVIDIA  devicenum=0
    time(us): 649,886
    44: data region reached 200 times
        44: data copyin transfers: 800
             device time(us): total=14,048 max=41 min=15 avg=17
        53: data copyout transfers: 800
             device time(us): total=11,731 max=43 min=6 avg=14
    44: compute region reached 200 times
        46: kernel launched 200 times
            grid: [32x4094]  block: [128]
             device time(us): total=382,798 max=1,918 min=1,911 avg=1,913
            elapsed time(us): total=391,408 max=1,972 min=1,953 avg=1,957
        46: reduction kernel launched 200 times
            grid: [1]  block: [256]
             device time(us): total=48,235 max=242 min=241 avg=241
            elapsed time(us): total=53,510 max=280 min=266 avg=267
    53: data region reached 200 times
```

[PGI tips] PGI_ACC_NOTIFY

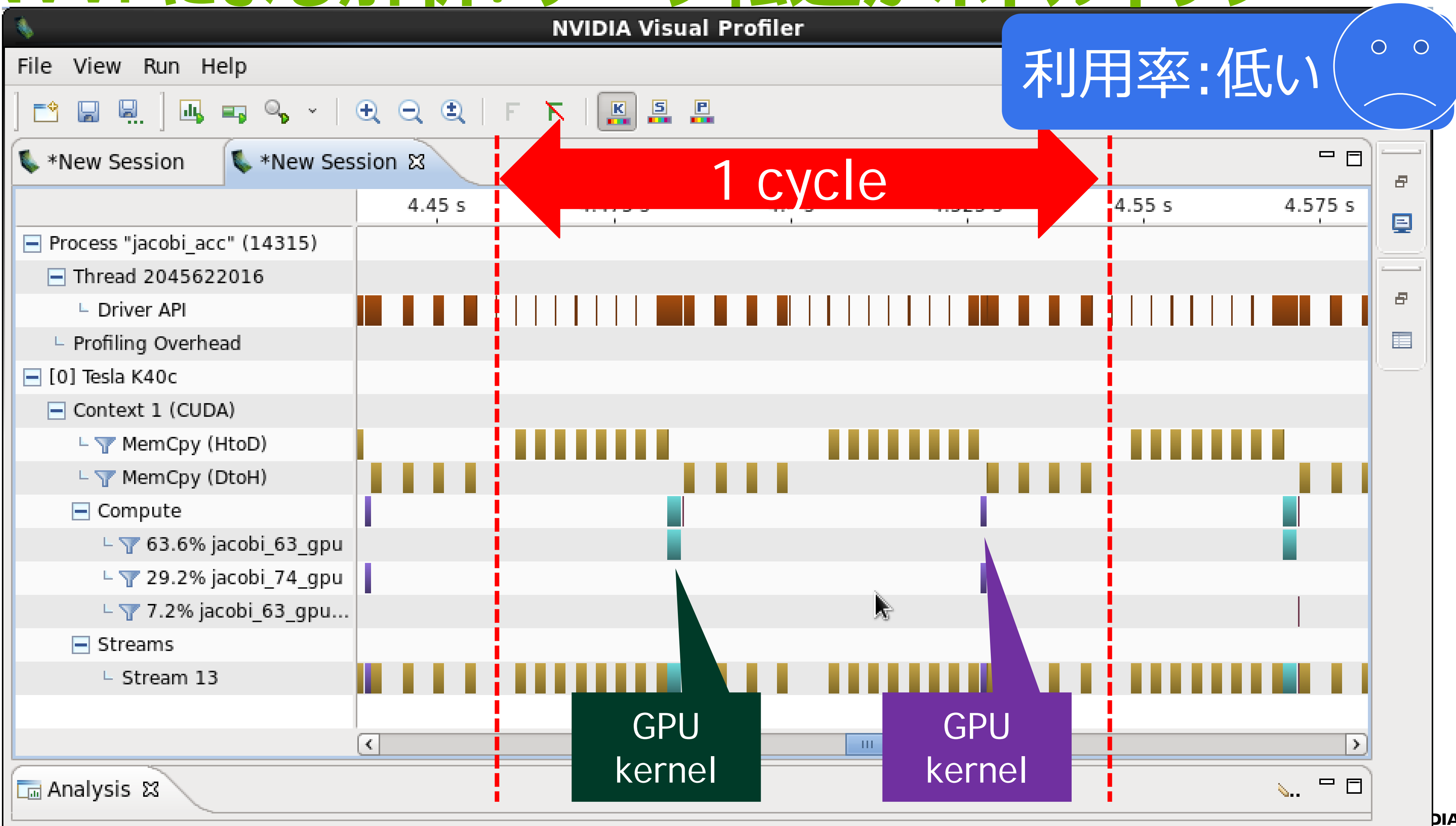
トレースデータ

```
$ PGI_ACC_NOTIFY=3 ./a.out
...
upload CUDA data  file=/home/anaruse/src/OpenACC/jacobi/C/task1-
solution/jacobi.c function=jacobi line=44 device=0 variable=A bytes=16777216
...
launch CUDA kernel  file=/home/anaruse/src/OpenACC/jacobi/C/task1-
solution/jacobi.c function=jacobi line=46 device=0 num_gangs=131008
num_workers=1 vector_length=128 grid=32x4094 block=128 shared memory=1024
...
download CUDA data  file=/home/anaruse/src/OpenACC/jacobi/C/task1-
solution/jacobi.c function=jacobi line=53 device=0 variable=Anew bytes=16736272
...
```

NVIDIA VISUAL PROFILER (NVVP)



NVVPによる解析: データ転送がボトルネック



過剰なデータ転送

```
while ( error > tol ) {
  error = 0.0;

  #pragma acc kernels \
      pcopy(Anew[:][:]) pcopyin(A[:][:])
  for (int j = 1; j < N-1; j++) {
    for (int i = 1; i < M-1; i++) {
      Anew[j][i] = (A[j][i+1] + A[j][i-1] +
                   A[j-1][i] + A[j+1][i]) * 0.25;
      error = max(error, abs(Anew[j][i] - A[j][i]);
    }
  }

  #pragma acc kernels \
      pcopy(A[:][:]) pcopyin(Anew[:][:])
  for (int j = 1; j < N-1; j++) {
    for (int i = 1; i < M-1; i++) {
      A[j][i] = Anew[j][i];
    }
  }
}
```

過剰なデータ転送

Host

```
while ( error > tol ) {
  error = 0.0;

  #pragma acc kernels \
      pcopy(Anew[:][:]) \
      pcopyin(A[:][:])
  {

  }

  #pragma acc kernels \
      pcopy(A[:][:]) \
      pcopyin(Anew[:][:])
  {

  }
}
```

copyin

copyout

copyin

copyout

GPU

```
#pragma acc loop reduction(max:error)
for (int j = 1; j < N-1; j++) {
  for (int i = 1; i < M-1; i++) {
    Anew[j][i] = (A[j][i+1] + A[j][i-1] +
                  A[j-1][i] + A[j+1][i]) * 0.25;
    error = max(error, abs(Anew[j][i] - A[j][i]);
  }
}
```

```
for (int j = 1; j < N-1; j++) {
  for (int i = 1; i < M-1; i++) {
    A[j][i] = Anew[j][i];
  }
}
```

NVIDIA

データ領域 (data construct)

```
#pragma acc data pcopy(A, Anew)
while ( error > tol ) {
  error = 0.0;

  #pragma acc kernels pcopy(Anew[:][:]) pcopyin(A[:][:])
  for (int j = 1; j < N-1; j++) {
    for (int i = 1; i < M-1; i++) {
      Anew[j][i] = (A[j][i+1] + A[j][i-1] +
                    A[j-1][i] + A[j+1][i]) * 0.25;
      error = max(error, abs(Anew[j][i] - A[j][i]);
    }
  }

  #pragma acc kernels pcopy(A[:][:]) pcopyin(Anew[:][:])
  for (int j = 1; j < N-1; j++) {
    for (int i = 1; i < M-1; i++) {
      A[j][i] = Anew[j][i];
    }
  }
}
```

- copyin (CPU→GPU)
- copyout (CPU←GPU)
- copy
- create
- present

- pcopyin
- pcopyout
- pcopy
- pcreate

データ領域 (data construct)

```
#pragma acc data pcopy(A) create(Anew)
while ( error > tol ) {
  error = 0.0;

  #pragma acc kernels pcopy(Anew[:][:]) pcopyin(A[:][:])
  for (int j = 1; j < N-1; j++) {
    for (int i = 1; i < M-1; i++) {
      Anew[j][i] = (A[j][i+1] + A[j][i-1] +
                    A[j-1][i] + A[j+1][i]) * 0.25;
      error = max(error, abs(Anew[j][i] - A[j][i]);
    }
  }

  #pragma acc kernels pcopy(A[:][:]) pcopyin(Anew[:][:])
  for (int j = 1; j < N-1; j++) {
    for (int i = 1; i < M-1; i++) {
      A[j][i] = Anew[j][i];
    }
  }
}
```

- copyin (CPU→GPU)
- copyout (CPU←GPU)
- copy
- create
- present

- pcopyin
- pcopyout
- pcopy
- pcreate

NVIDIA

適正なデータ転送

Host

```
#pragma acc data \
    pcopy(A) create(Anew)
while ( error > tol ) {
  error = 0.0;

  #pragma acc kernels \
      pcopy(Anew[:][:]) \
      pcopyin(A[:][:])
  {

  }

  #pragma acc kernels \
      pcopy(A[:][:]) \
      pcopyin(Anew[:][:])
  {

  }
}
```

copyin

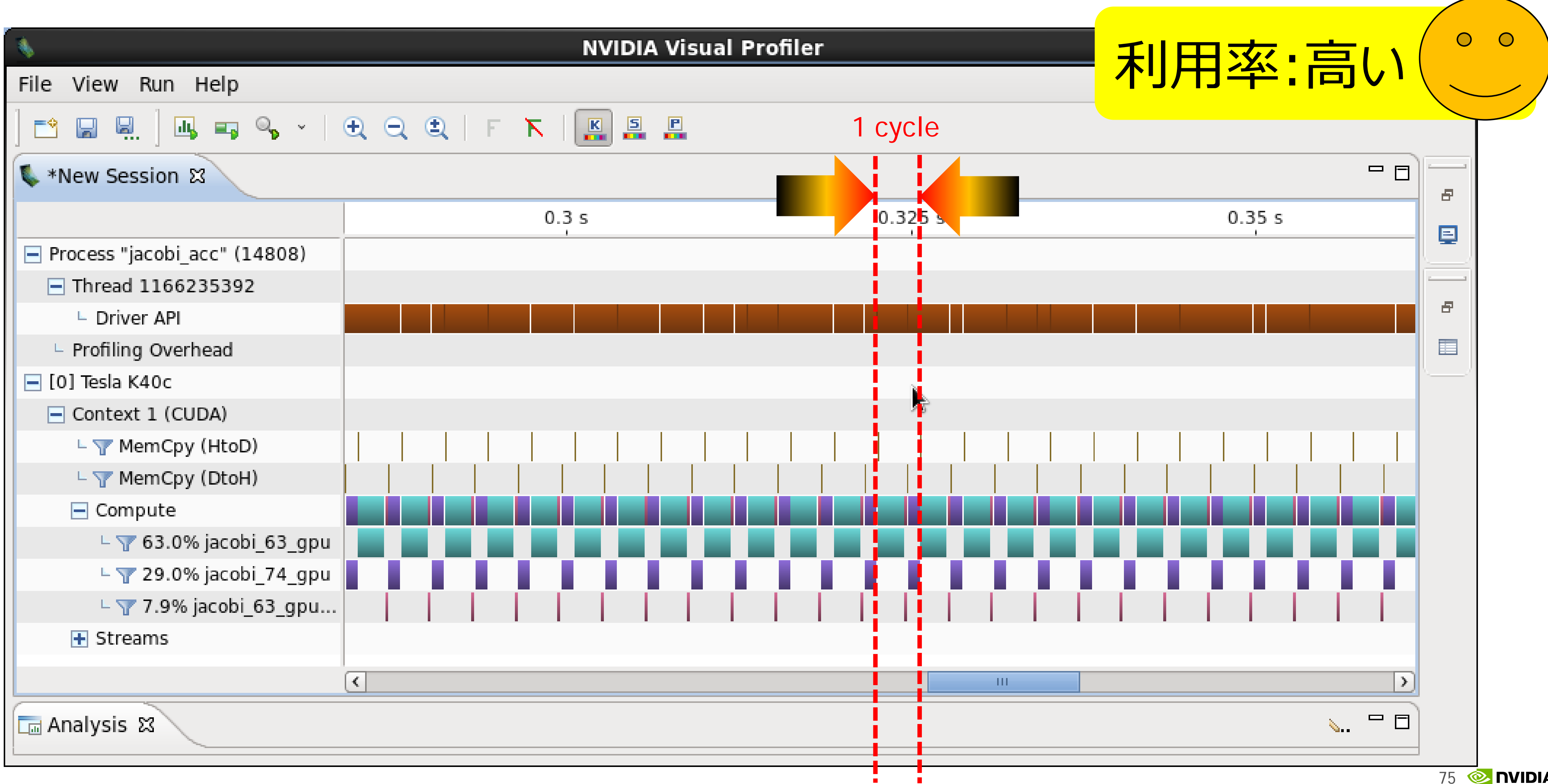
copyout

GPU

```
for (int j = 1; j < N-1; j++) {
  for (int i = 1; i < M-1; i++) {
    Anew[j][i] = (A[j][i+1] + A[j][i-1] +
                  A[j-1][i] + A[j+1][i]) * 0.25;
    error = max(error, abs(Anew[j][i] - A[j][i]);
  }
}

for (int j = 1; j < N-1; j++) {
  for (int i = 1; i < M-1; i++) {
    A[j][i] = Anew[j][i];
  }
}
```

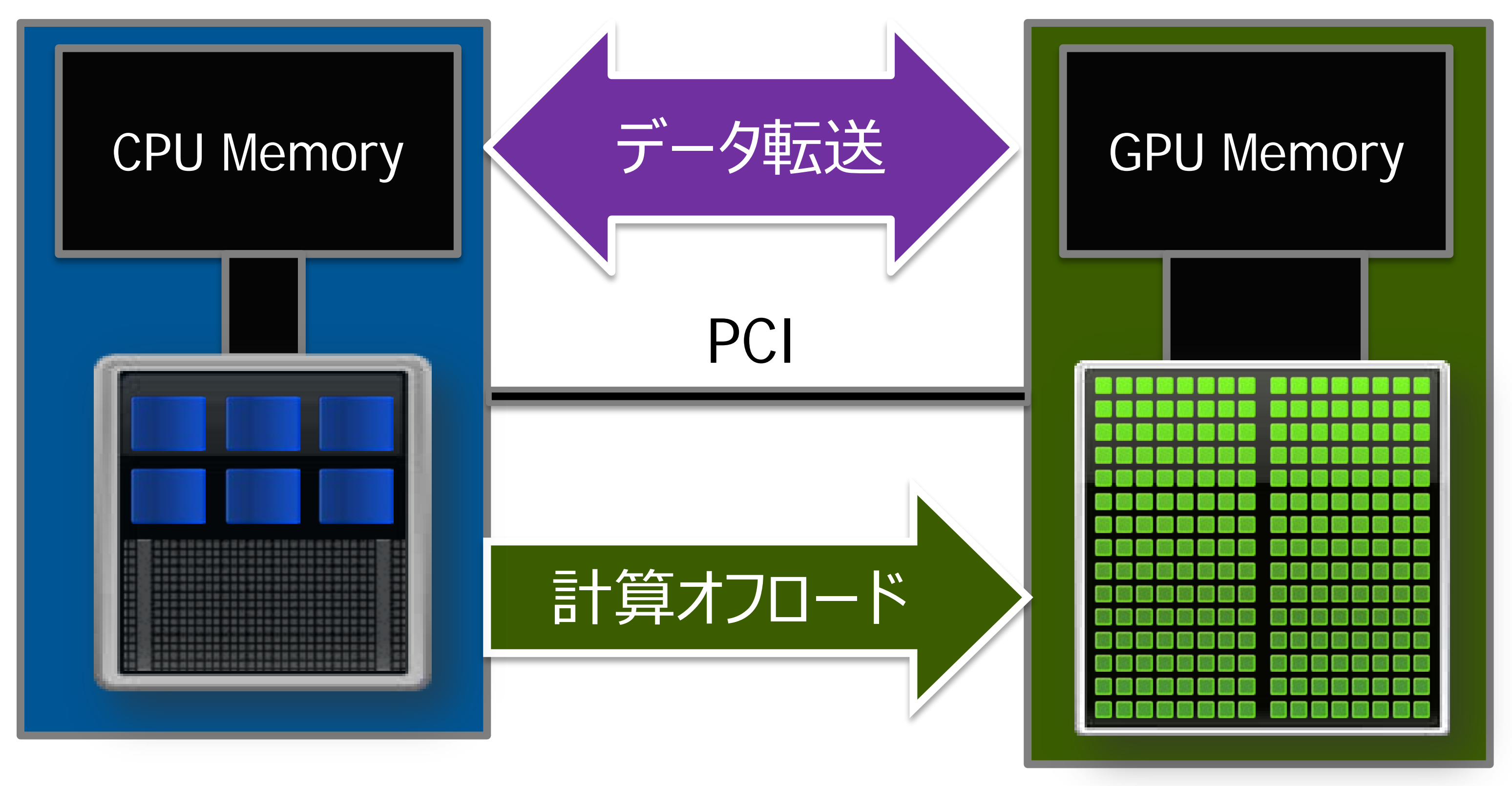
NVIDIA

データ転送が減少 (NVVP)



NVIDIA

2つの処理



計算オフロード、データ転送、両方を考慮する必要がある

その他のデータ管理方法

```
float *array;

Init( ) {
    ...
    array = (float*)malloc( … );
    input_array( array );
    #pragma enter data copyin(array)
    ...
}

Fin( ) {
    ...
    #pragma exit data copyout(array)
    output_array( array );
    free( array );
    ...
}
```

- Enter data
  - Copyin
  - Create
  - Pcopyin
  - Pcreate
- Exit data
  - Copyout
  - Delete

その他のデータ管理方法

```
#pragma acc data pcopy(A,B)
for (k=0; k<LOOP; k++) {
    #pragma acc kernels present(A,B)
    for (i=0; i<N; i++) {
        A[i] = subA(i,A,B);
    }

    #pragma acc update self(A[0:1])
    output[k] = A[0];

    A[N-1] = input[k];
    #pragma acc update device(A[N-1:1])

    #pragma acc kernels present(A,B)
    for (i=0; i<N; i++) {
        B[i] = subB(i,A,B);
    }
}
```

- Update self
  CPU ← GPU
- Update device
  CPU → GPU

リダクション(縮約計算)

```
while ( error > tol ) {
  error = 0.0;

  #pragma acc kernels
  for (int j = 1; j < N-1; j++) {
    for (int i = 1; i < M-1; i++) {
      Anew[j][i] = (A[j][i+1] + A[j][i-1] +
                    A[j-1][i] + A[j+1][i]) * 0.25;
      error = max(error, abs(Anew[j][i] - A[j][i]);
    }
  }

  ...
}
```

リダクション(縮約計算)

```
while ( error > tol ) {
  error = 0.0;

  #pragma acc kernels
  for (int j = 1; j < N-1; j++) {
    for (int i = 1; i < M-1; i++) {
      Anew[j][i] = (A[j][i+1] + A[j][i-1] +
```

```
$ pgcc -Minfo=acc -acc jacobi.c
jacobi:
     59, Generating present_or_copyout(Anew[1:4094][1:4094])
         Generating present_or_copyin(A[:][:])
         Generating Tesla code
     61, Loop is parallelizable
     63, Loop is parallelizable
         Accelerator kernel generated
         61, #pragma acc loop gang /* blockIdx.y */
         63, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
             Max reduction generated for error
```

リダクション (REDUCTION CLAUSE)

```
while ( error > tol ) {
  error = 0.0;

  #pragma acc kernels
  #pragma acc loop reduction(max:error)
  for (int j = 1; j < N-1; j++) {
    for (int i = 1; i < M-1; i++) {
      Anew[j][i] = (A[j][i+1] + A[j][i-1] +
                    A[j-1][i] + A[j+1][i]) * 0.25;
      error = max(error, abs(Anew[j][i] - A[j][i]);
    }
  }

  ...
}
```

- 演算種類 (C/C++)

+	和
*	積
max	最大
min	最小
|	ビット和
&	ビット積
^	XOR
||	論理和
&&	論理積

並列方法の指示

```
#pragma acc data pcopy(A) create(Anew)
while ( error > tol ) {
  error = 0.0;

  #pragma acc kernels pcopy(Anew[:][:]) pcopyin(A[:][:])
  #pragma acc loop reduction(max:error)
  for (int j = 1; j < N-1; j++) {
    for (int i = 1; i < M-1; i++) {
      Anew[j][i] = (A[j][i+1] + A[j][i-1] +
                    A[j-1][i] + A[j+1][i]) * 0.25;
      error = max(error, abs(Anew[j][i] - A[j][i]);
    }
  }

  ...

}
```

並列方法の指示

```
#pragma acc data pcopy(A) create(Anew)
while ( error > tol ) {
  error = 0.0;

  #pragma acc kernels pcopy(Anew[:][:]) pcopyin(A[:][:])
  #pragma acc loop reduction(max:error)
```

```
$ pgcc -Minfo=acc -acc jacobi.c
jacobi:
     59, Generating present_or_copyout(Anew[1:4094][1:4094])
         Generating present_or_copyin(A[:][:])
         Generating Tesla code
     61, Loop is parallelizable
     63, Loop is parallelizable
         Accelerator kernel generated
         61, #pragma acc loop gang /* blockIdx.y */
         63, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
             Max reduction generated for error
```

並列方法の指示 (loop construct)

```
#pragma acc data pcopy(A) create(Anew)
while ( error > tol ) {
  error = 0.0;

  #pragma acc kernels pcopy(Anew[:][:]) pcopyin(A[:][:])
  #pragma acc loop gang vector(1) reduction(max:error)
  for (int j = 1; j < N-1; j++) {
    #pragma acc loop gang vector(128)
    for (int i = 1; i < M-1; i++) {
      Anew[j][i] = (A[j][i+1] + A[j][i-1] +
                    A[j-1][i] + A[j+1][i]) * 0.25;
      error = max(error, abs(Anew[j][i] - A[j][i]);
    }
  }

  ...

}
```

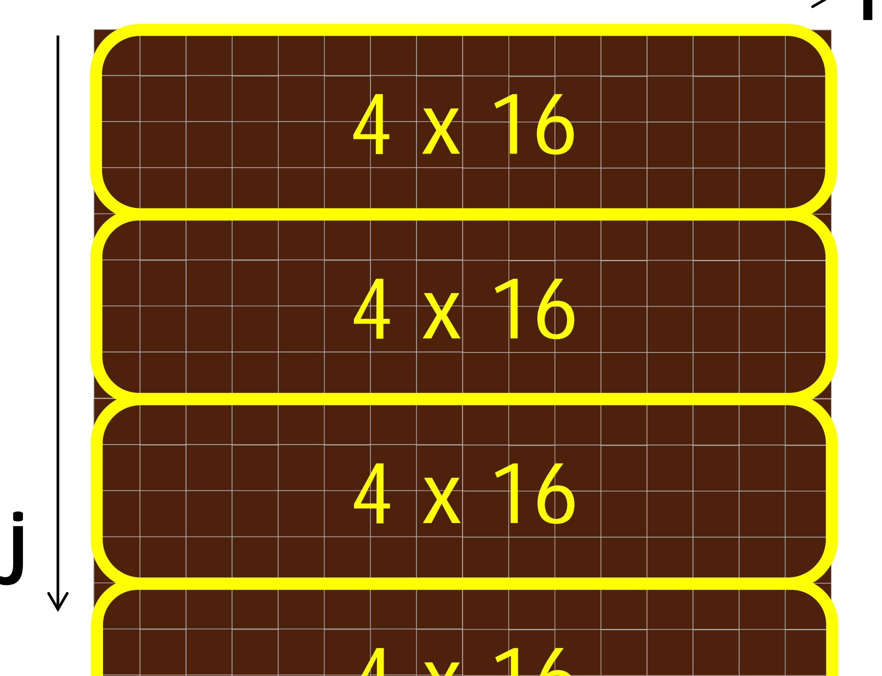
- Gang
- Worker
- Vector ... SIMD幅

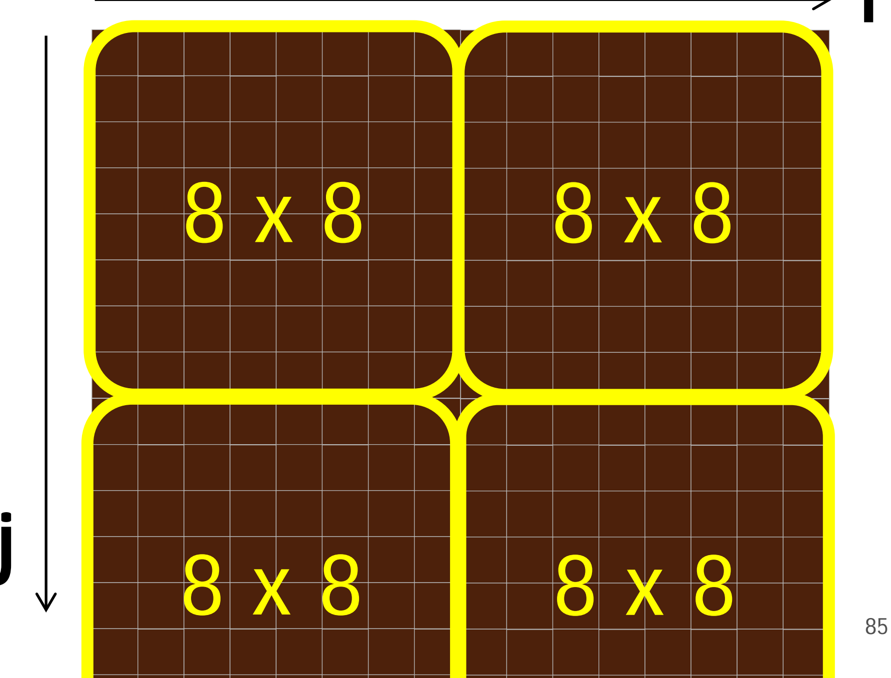
- Collapse
- Independent
- Seq
- Cache
- Tile

実行条件設定 (gang, vector)

```
#pragma acc loop gang vector(4)
for (j = 0; j < 16; j++) {
    #pragma acc loop gang vector(16)
    for (i = 0; i < 16; i++) {
        ...
```

```
#pragma acc loop gang vector(8)
for (j = 1; j < 16; j++) {
    #pragma acc loop gang vector(8)
    for (i = 0; i < 16; i++) {
        ...
```





ループを融合 (collapse)

```
#pragma acc data pcopy(A) create(Anew)
while ( error > tol ) {
  error = 0.0;

  #pragma acc kernels pcopy(Anew[:][:]) pcopyin(A[:][:])
  #pragma acc loop reduction(max:error) \
      collapse(2) gang vector(128)
  for (int j = 1; j < N-1; j++) {
    for (int i = 1; i < M-1; i++) {
      Anew[j][i] = (A[j][i+1] + A[j][i-1] +
                    A[j-1][i] + A[j+1][i]) * 0.25;
      error = max(error, abs(Anew[j][i] - A[j][i]);
    }
  }
  ...
}
```

- Gang
- Worker
- Vector ... SIMD幅

- Collapse
- Independent
- Seq
- Cache
- Tile
- ...

ループを融合 (collapse)

```
#pragma acc data pcopy(A) create(Anew)
while ( error > tol ) {
  error = 0.0;

  #pragma acc kernels pcopy(Anew[:][:]) pcopyin(A[:][:])
  #pragma acc loop reduction(max:error) gang vector(128)
  for (int ji = 0; ji < (N-2)*(M-2); ji++) {
      j = (ji / (M-2)) + 1;
      i = (ji % (M-2)) + 1;
      Anew[j][i] = (A[j][i+1] + A[j][i-1] +
                    A[j-1][i] + A[j+1][i]) * 0.25;
      error = max(error, abs(Anew[j][i] - A[j][i]);
  }
  ...
}
```

- Gang
- Worker
- Vector … SIMD幅

- Collapse
- Independent
- Seq
- Cache
- Tile
- …

並列実行可能(independent)

```
#pragma acc data pcopy(A) create(Anew)
while ( error > tol ) {
  error = 0.0;

  #pragma acc kernels pcopy(Anew[:][:]) pcopyin(A[:][:])
  #pragma acc loop reduction(max:error) independent
  for (int jj = 1; jj < NN-1; jj++) {
    int j = list_j[jj];
    for (int i = 1; i < M-1; i++) {
      Anew[j][i] = (A[j][i+1] + A[j][i-1] +
                    A[j-1][i] + A[j+1][i]) * 0.25;
      error = max(error, abs(Anew[j][i] - A[j][i]);
    }
  }
  ...
}
```

- Gang
- Worker
- Vector … SIMD幅

- Collapse
- Independent
- Seq
- Cache
- Tile
- …

逐次に実行 (seq)

```
#pragma acc kernels pcopy(Anew[:][:]) pcopyin(A[:][:])
#pragma acc loop seq
for (int k = 3; k < NK-3; k++) {
  #pragma acc loop
  for (int j = 0; j < NJ; j++) {
    #pragma acc loop
    for (int i = 0; i < NI; i++) {
      Anew[k][j][i] = func(
          A[k-1][j][i], A[k-2][j][i], A[k-3][j][i],
          A[k+1][j][i], A[k+2][j][i], A[k+3][j][i], ...
      );
    }
  }
}
```

- Gang
- Worker
- Vector … SIMD幅

- Collapse
- Independent
- Seq
- Cache
- Tile
- …

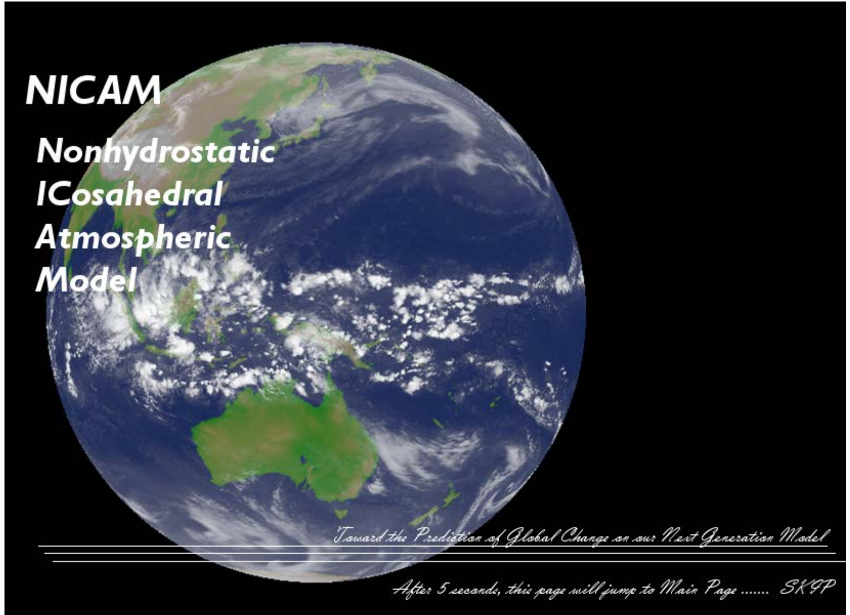
OPENACCプログラミング

- 概要紹介
- プログラムのOpenACC化
- OpenACC化事例

NVIDIA

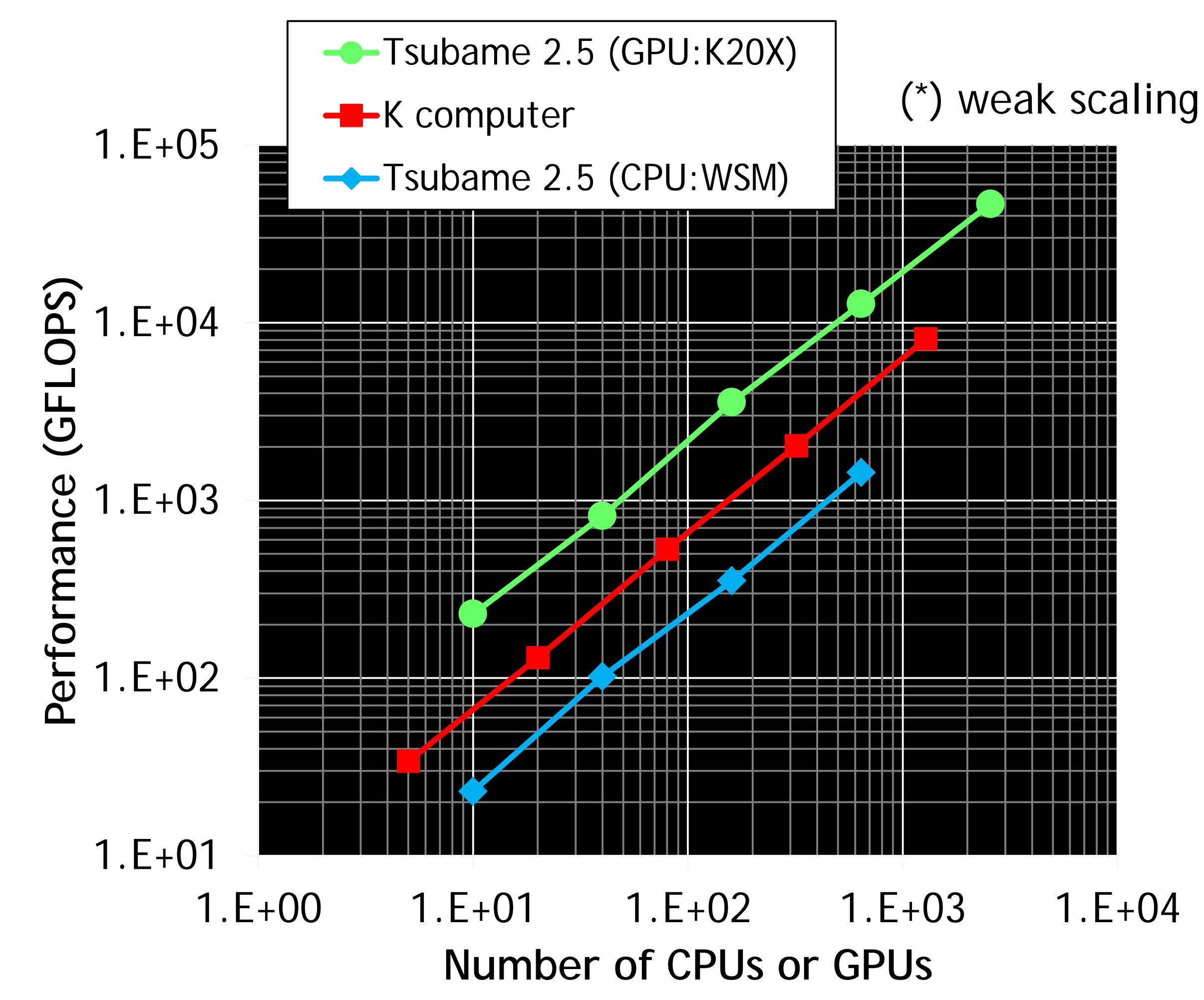
NICAM

- 気象・気候モデル by 理研AICS/東大
  - 膨大なコード (数十万行)
  - ホットスポットがない (~~パレートの法則~~)
- 特性の異なる2種類の処理
  - 力学系 … メモリバンド幅ネック
  - 物理系 … 演算ネック



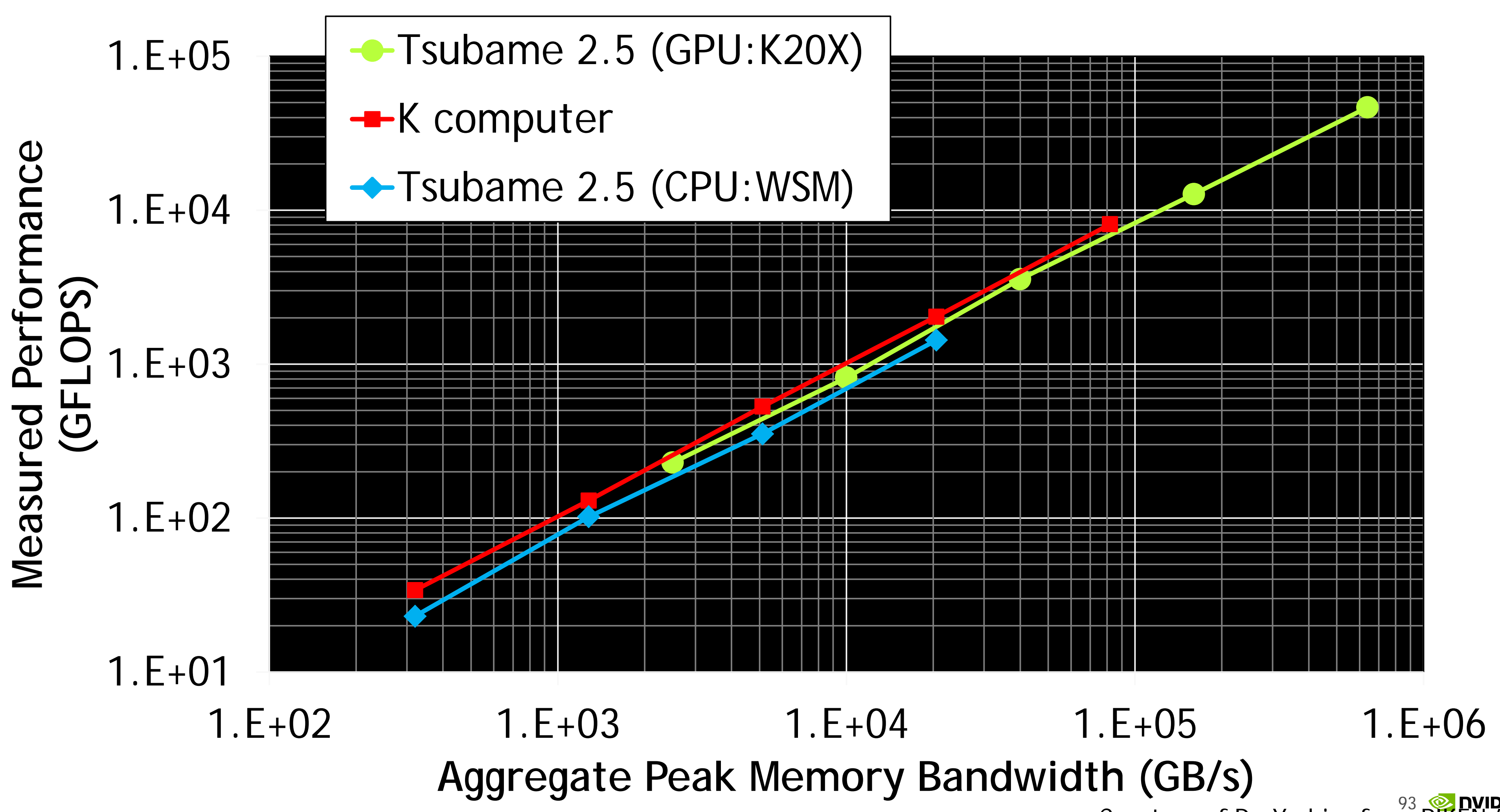
NICAM: 力学系(NICAM-DC)

- OpenACCによるGPU化
  - 主要サブルーチンは、全てGPU上で動作(50以上)
  - MPI対応済み
  - 2週間
- 良好なスケーラビリティ
  - Tsubame 2.5, 最大2560 GPUs
  - Scaling factor: 0.8



Courtesy of Dr. Yashiro from RIKEN AICS

NICAM: 力学系(NICAM-DC)

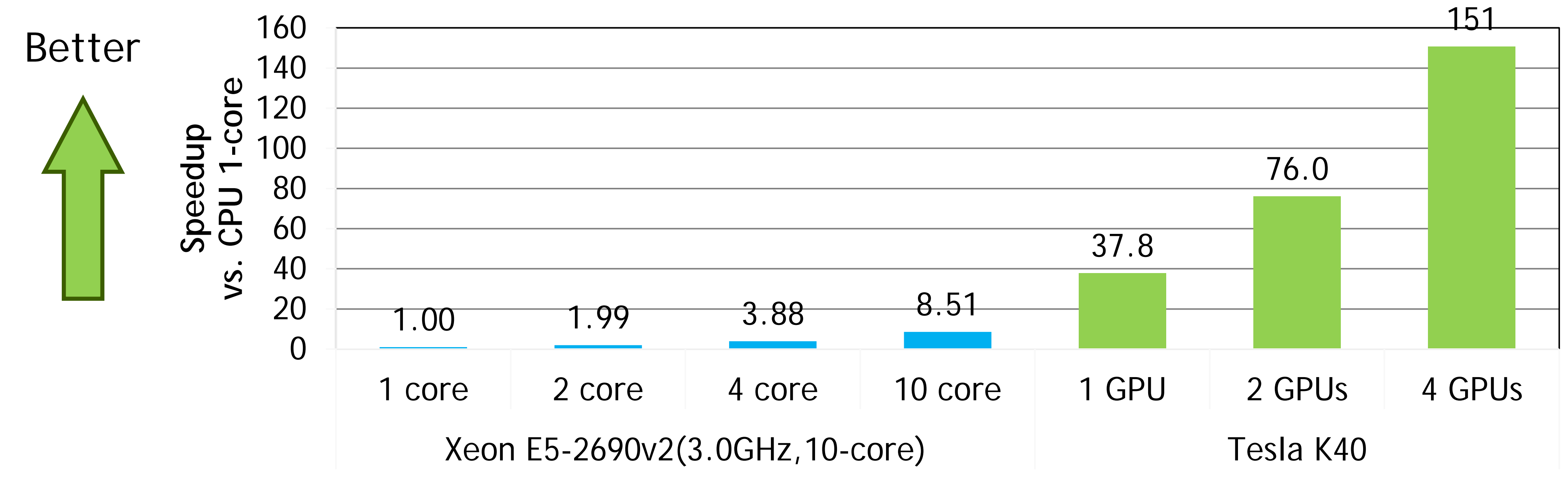


Courtesy of Dr. Yashiro from RIKEN AICS

NVIDIA

NICAM: 物理系(SCALE-LES)

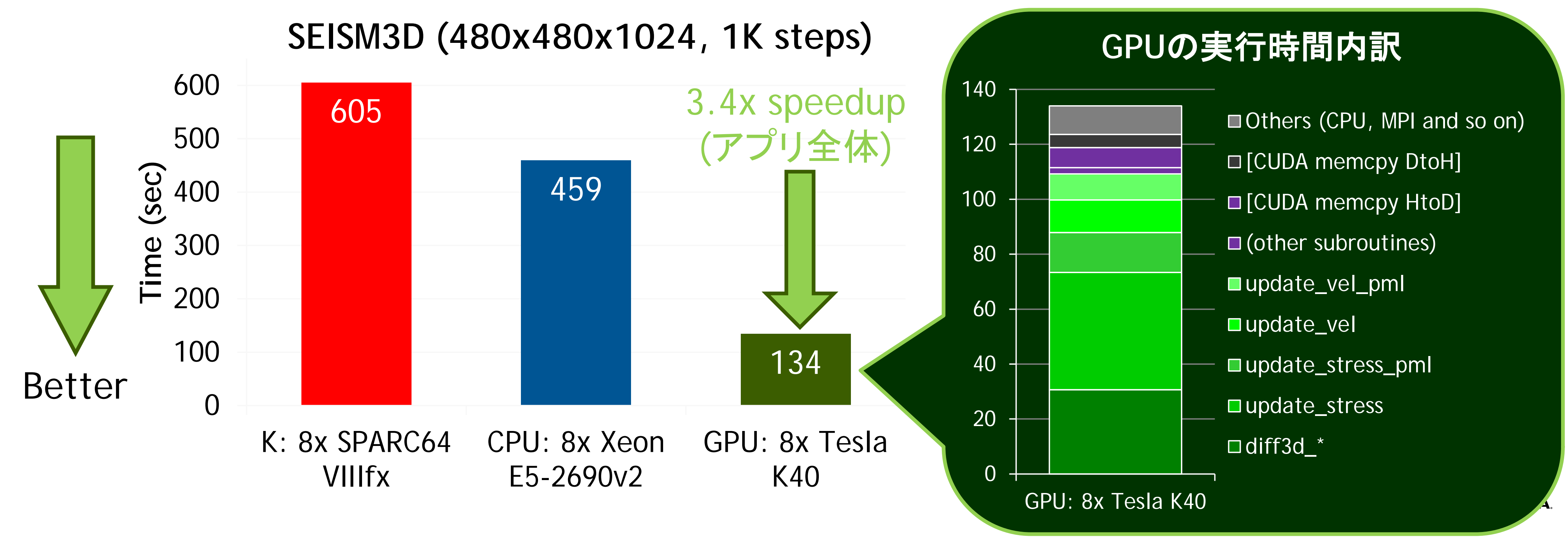
- Atmospheric radiation transfer
  - 物理系の中で、最も重い計算
  - OpenACCによるGPU対応、完了



(*) PCIデータ転送時間込み, グリッドサイズ:1256x32x32

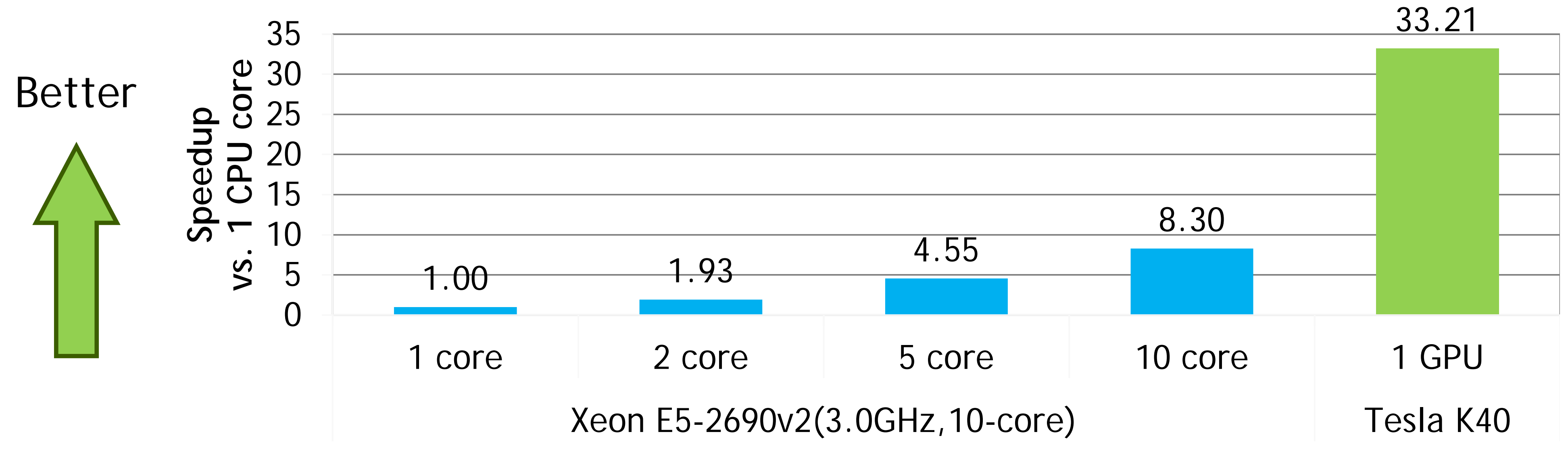
SEISM3D

- 地震シミュレーション　by 古村教授(東大地震研)
- 主要サブルーチンのGPU対応が完了
  - メモリバンド幅ネック、3次元モデル(2次元分割)、隣接プロセス間通信



FFR/BCM (仮称)

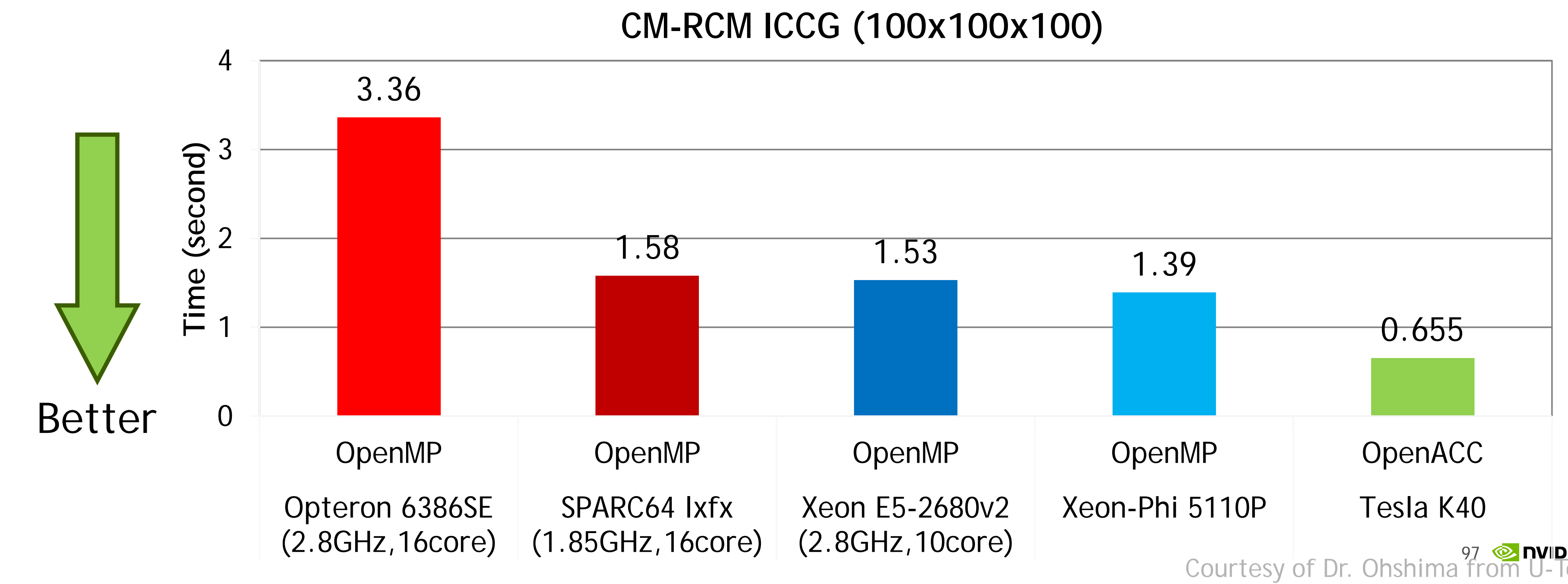
- 次世代CFDコード by 坪倉准教授(理研AICS/北大)
- MUSCL_bench:
  - MUSCLスキームに基づくFlux計算 (とても複雑な計算)
  - CFD計算の主要部分 (60-70%)
  - OpenACCによるGPU対応、完了



(*) PCIデータ転送時間込み、サイズ:80x32x32x32

CM-RCM IC-CG (PRELIMINARY)

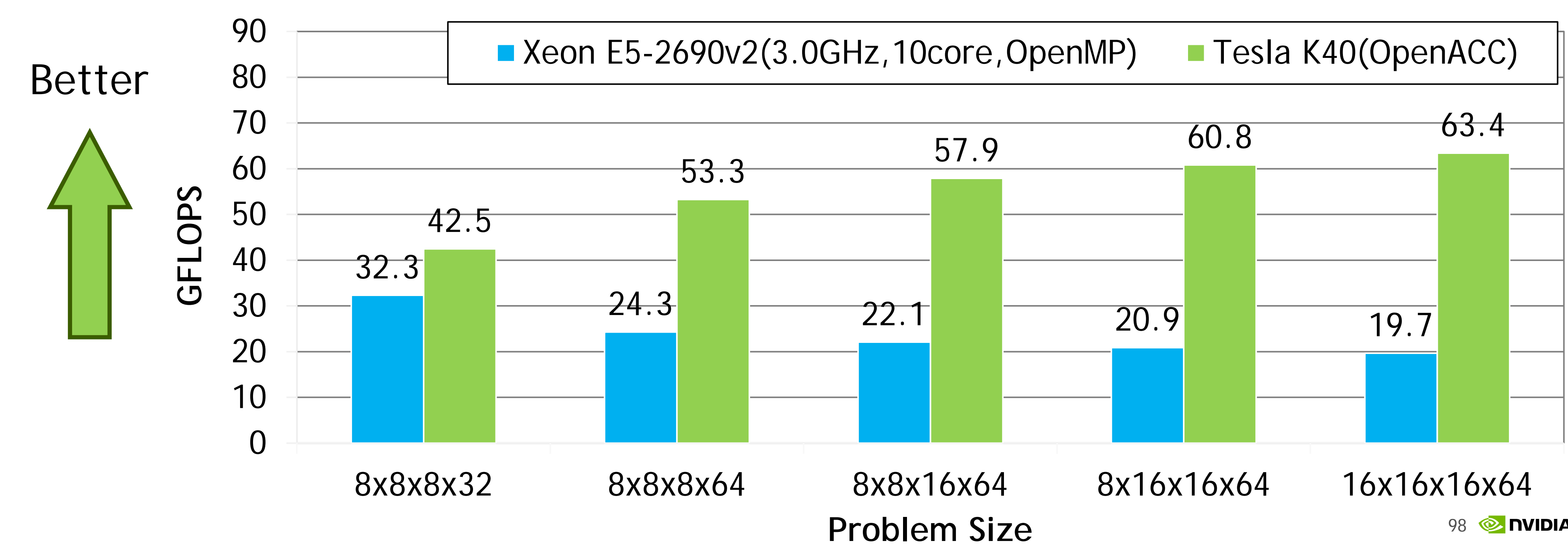
- IC-CG法のベンチマークコード by 中島教授(東大)
  - CM-RCM法(Cyclic Multi-coloring Reverse Cuthill-Mckee)を使用
- メインループ内のサブルーチンを全てOpenACCでGPU化



NVIDIA
Courtesy of Dr. Ohshima from U-Tokyo

CCS-QCD

- QCDコード by 石川准教授(広島大)
- BiCGStab計算を全てOpenACCでGPU化
  - データレイアウトを変更: AoS → SoA



NVIDIA

NVIDIA