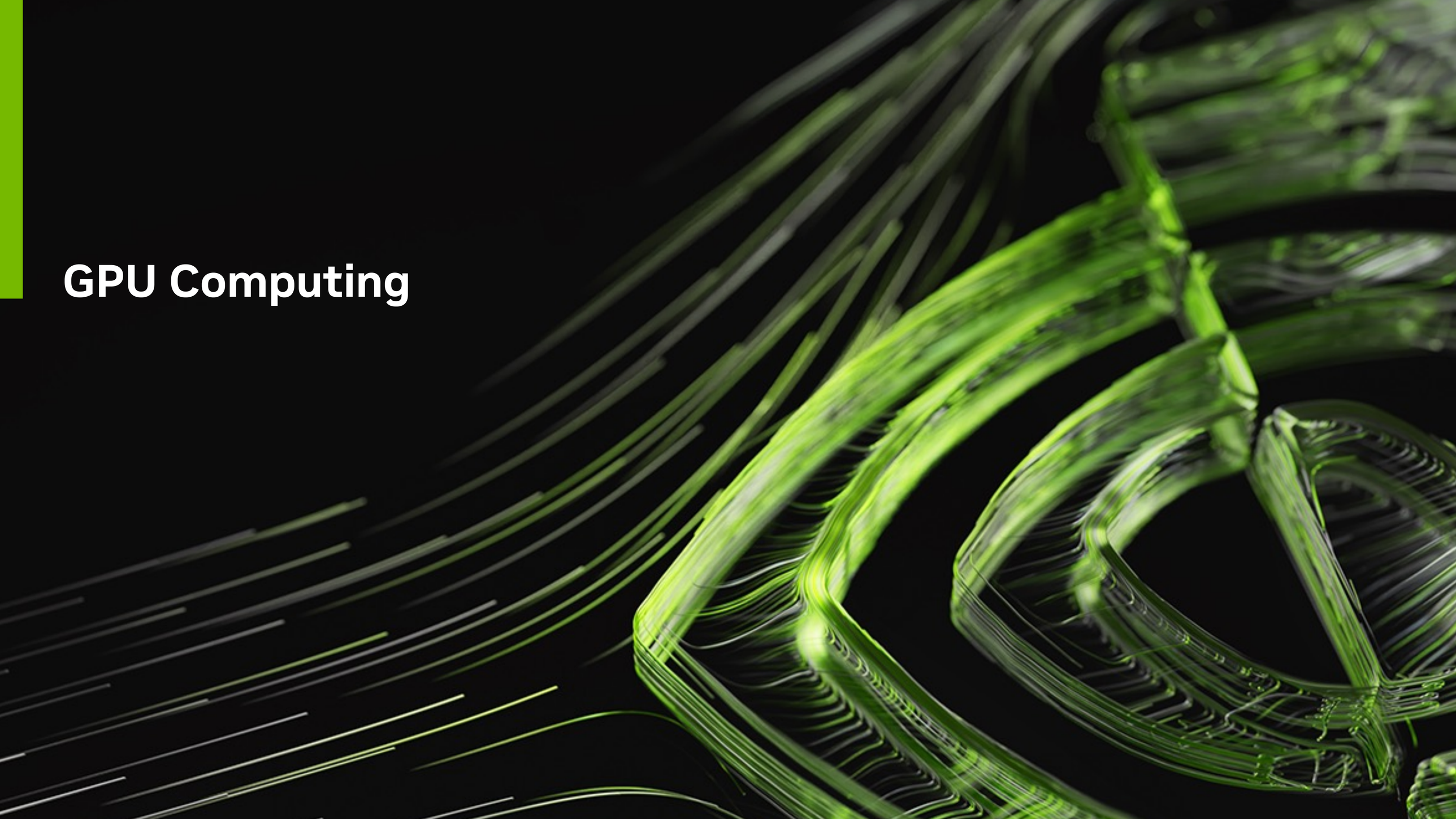
CUDA や OpenACC による GPU Computing

Naruhiko Tan, 2023/4/20

Agenda

- GPU Computing
- OpenACC
- CUDA
- GPU on TSUBAME3.0 and the latest GPU

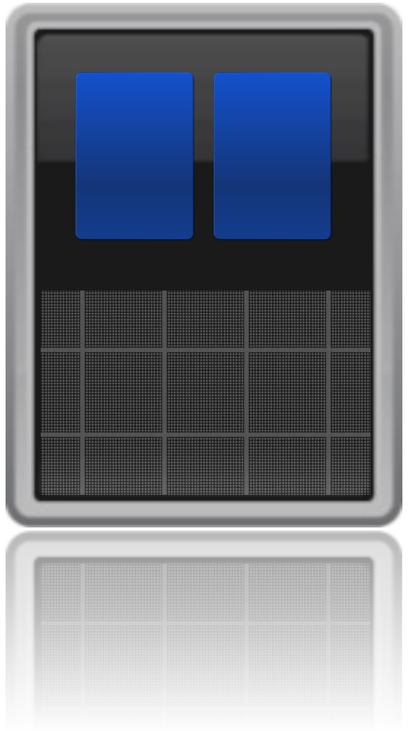
GPU Computing



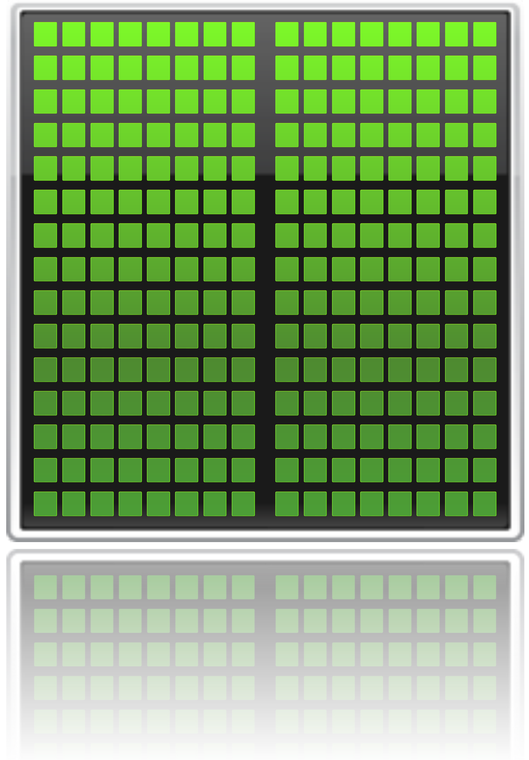
GPU コンピューティング

Low latency + High throughput

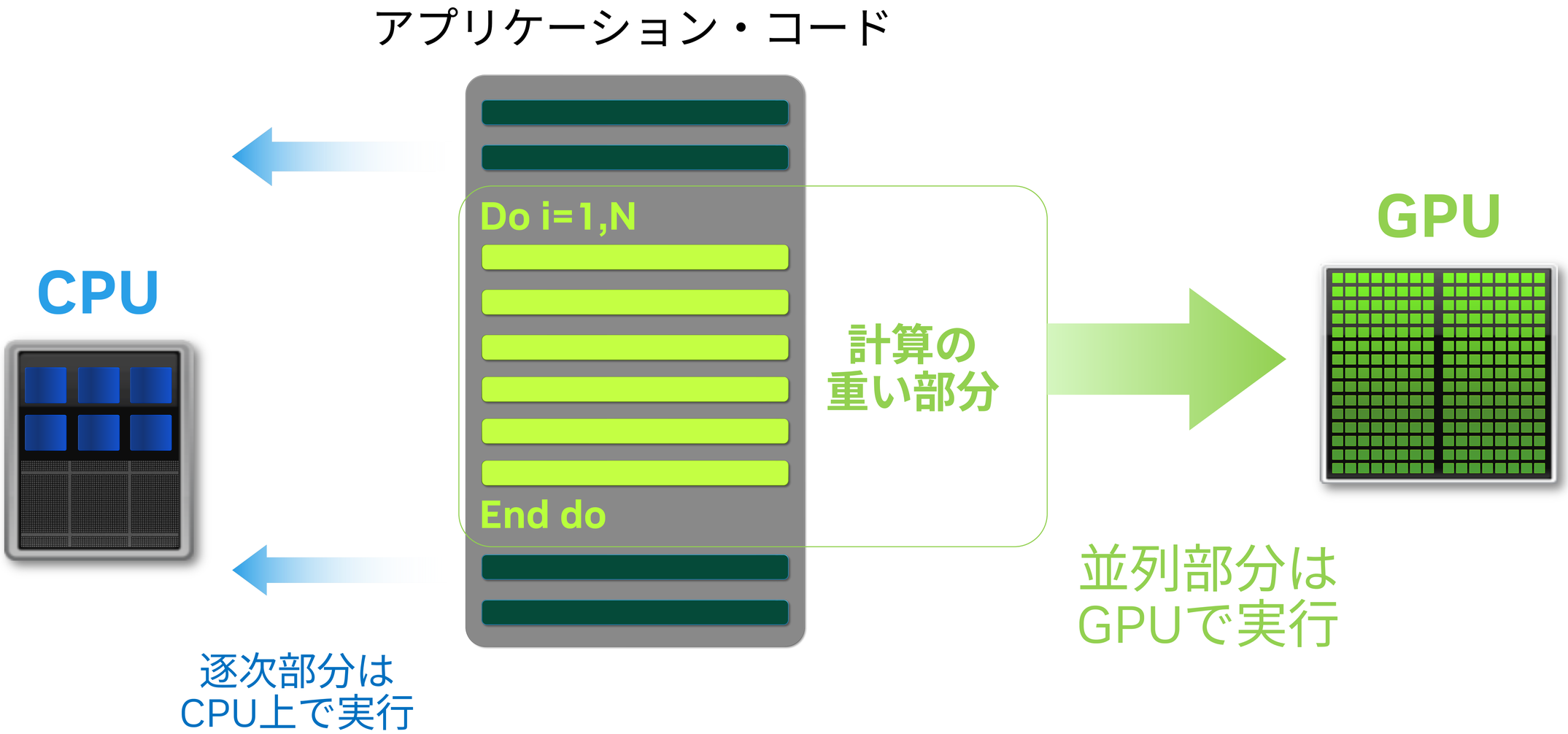
CPU



GPU



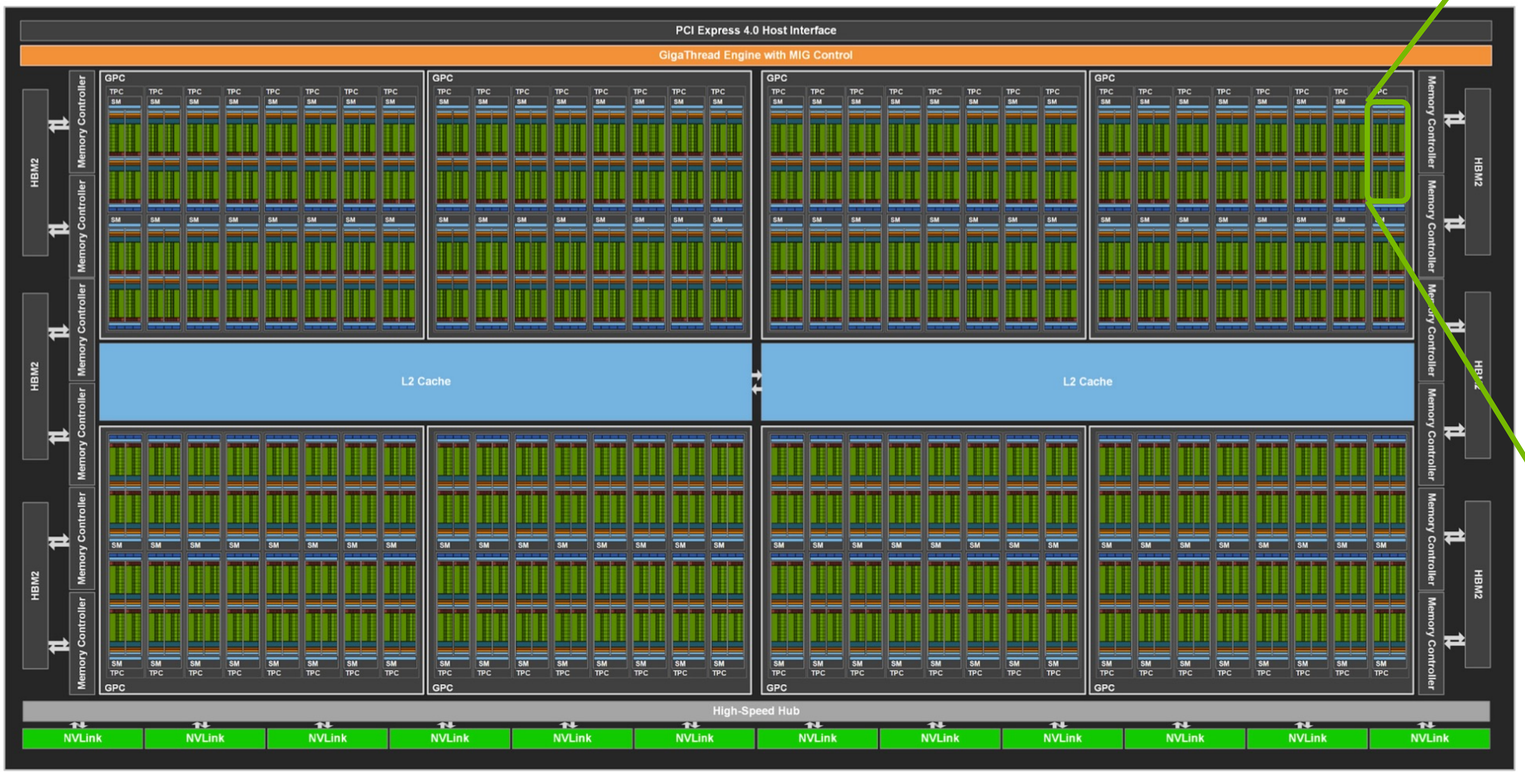
アプリケーション実行


アプリケーション・コード
Do i=1,N
End do
CPU
GPU
計算の
重い部分
並列部分は
GPUで実行
逐次部分は
CPU上で実行

GPU の構造

NVIDIA A100

大量の CUDA コア
並列性の抽出が鍵



108 Streaming Multiprocessor (SM) / chip

64 CUDA core / SM

GPU Applications

https://www.nvidia.com/en-us/gpu-accelerated-applications/

NVIDIA. Products Solutions Industries For You Shop Drivers Support

Accelerated Apps Catalog

Use the search bar below to find out if your favorite app is one of the hundreds being accelerated by NVIDIA GPUs. Can't find a specific app? Come back and search again as more are added every month.

Filter Categories Filter Industries Sort AZ AI Accelerated Share Search apps

Prev Show 15 per page Page 1 Next

Shenzhen Rayvision Technology Co Ltd

3D CAT.live

Real-time rendering cloud service for 3D applications. The massive GPU computing power in the cloud is used to process heavy image rendering calculations and stream output to the terminal device synchronously, thereby realizing light weight

3D Slicer

3D Slicer

3D Slicer is an open-source software platform for medical image informatics, image processing, and three-dimensional visualization. Slicer brings free, powerful cross-platform processing tools to physicians, researchers, and the general public.

QT Imaging Inc

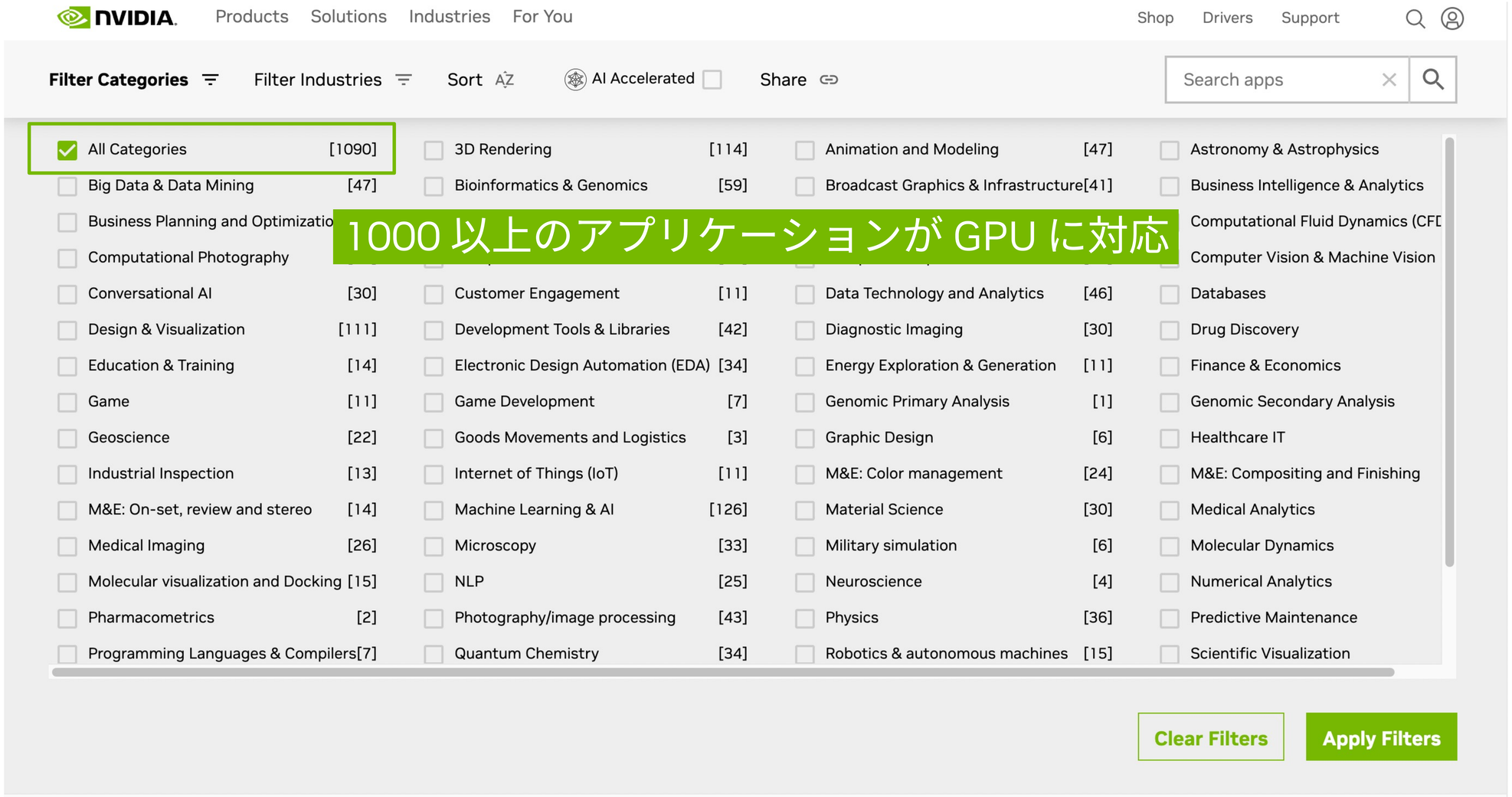
3D Ultrasound Tomography/Volography

3D ultrasound breast, pediatric and whole body tomography.

- Quantitative high resolution 3D US images of tissue properties of breast with GPU

GPU Applications

https://www.nvidia.com/en-us/gpu-accelerated-applications/



1000 以上のアプリケーションが GPU に対応

Leading Applications

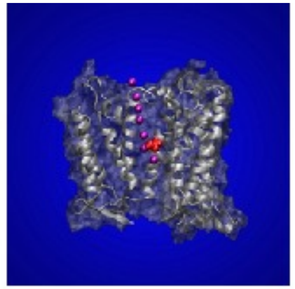
VASP
b-initio
ackage
imulation
ienna

GROMACS FAST. FLEXIBLE. FREE.

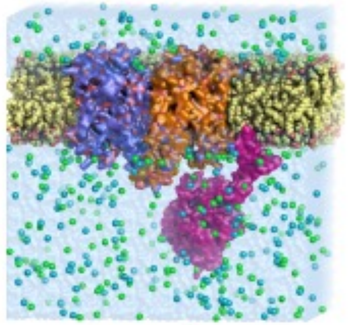
CONSORTIUM FOR SMALL SCALE MODELING
COSMO

NAMD
Scalable Molecular Dynamics

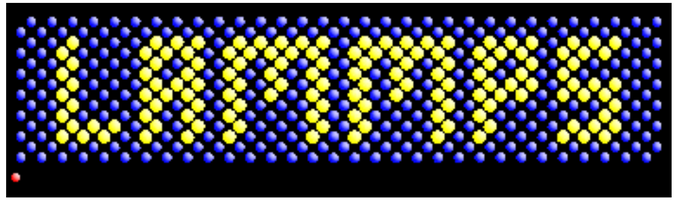
Gaussian

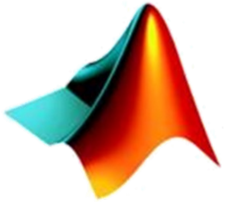


Desmond



Amber





MATLAB

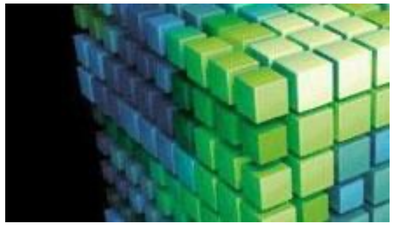
HOOMD-blue

アプリを GPU 対応する方法

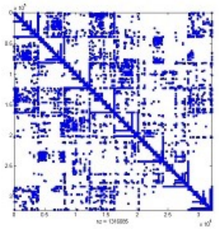
Application

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

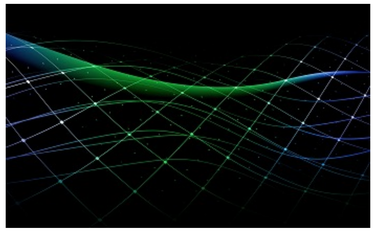
NVIDIA Libraries



cuBLAS

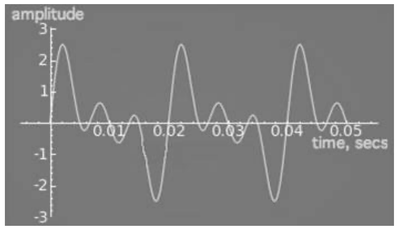


cuSPARSE

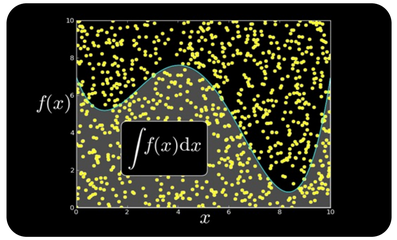


cuSOLVER

AmgX



cuFFT



cuRAND

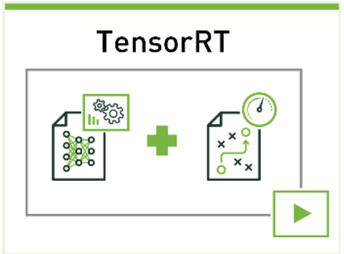
nvGRAPH

Thrust

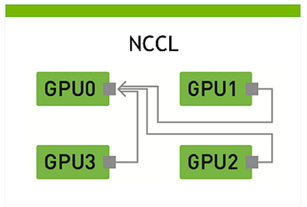
Performance Primitives

cuDNN

cuDNN



TensorRT



NCCL

Partner Libraries

OpenCV Computer Vision	FFmpeg Audio and Video	ArrayFire Matrix, Signal and Image	MAGMA Linear Algebra
IMSL RogueWave Software Math, Signal and Image	GUNROCK GPU GRAPH ANALYTICS FRAMEWORK AT UC DAVIS Graph	SuiteSparse CHOLMOD Sparse direct solvers	CULA tools Linear Algebra
[HiPLAR] Linear Algebra	gpp Geometry Performance Primitives Computational Geometry	API: C++ FORTRAN; Plug-ins: MATLAB/Octave, Deal.II, OpenFOAM, Hermes, Elmer, ...; PARALUTION; Multi-core CPU OpenMP; GPU CUDA/OpenCL; Xeon Phi OpenMP Sparse Iterative Methods	Sundog™ Software Real-time visual simulation

アプリを GPU 対応する方法

Application

Library

GPU 対応ライブラリに
チェンジ
簡単に開始

OpenACC

既存コードに
ディレクティブを挿入
簡単に加速

CUDA

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);
...
```

アプリを GPU 対応する方法

Application

Library

GPU 対応ライブラリに
チェンジ
簡単に開始

OpenACC

既存コードに
ディレクティブを挿入
簡単に加速

CUDA

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)
    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);
cudaDeviceSynchronize();
cudaMemcpy(y, d_y, size, cudaMemcpyDeviceToHost);
...
```

OpenACC

アプリを GPU 対応する方法

Application

Library

GPU 対応ライブラリに
チェンジ
簡単に開始

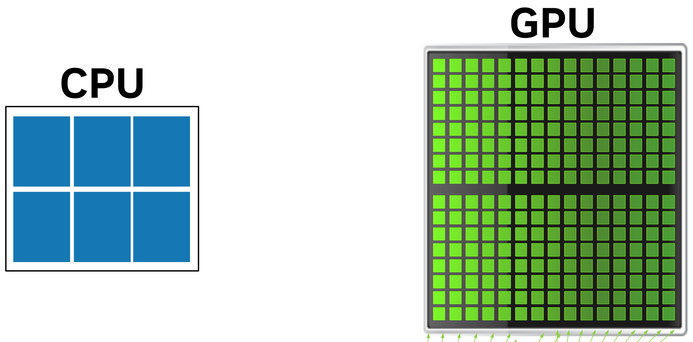
OpenACC

既存コードに
ディレクティブを挿入
簡単に加速

CUDA

主要処理を CUDA で記述
高い自由度

OpenACC




```
Program myscience
   ... serial code ...
!$acc kernels
  do k = 1,n1
    do i = 1,n2
      ... parallel code ...
    enddo
  enddo
!$acc end kernels
   ... serial code …
End Program myscience
```

既存の C/Fortran コード

ヒントの追加

簡単: 既存のコードにコンパイラへのヒントを追加

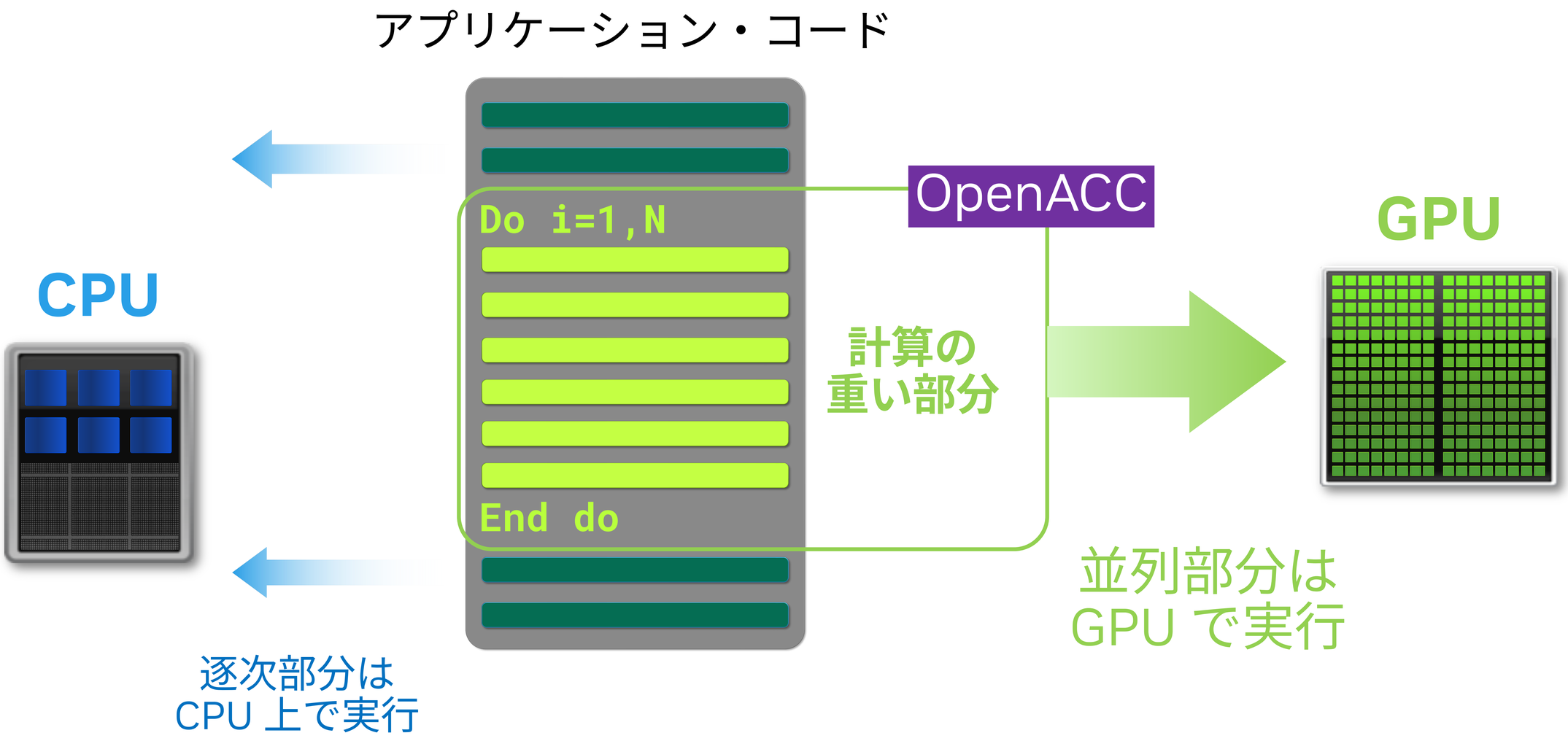
強力: 相応の労力で、コンパイラが自動で並列化

オープン: 複数コンパイラベンダが、様々なプロセッサをサポート

NVIDIA GPU, AMD GPU,

x86 CPU, ARM CPU

GPU Computing



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

OpenACC: Parallel Directive

並列化領域を指示する

```
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);
...
```

OpenACC: Kernels Directive

並列化領域を指示する

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

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

Parallel: ユーザ主導
Kernels: コンパイラ主導

OpenACC: Loop Directive

ループの並列化方法を指示する

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

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

independent: 並列化可能

seq: 逐次実行

collapse: ループ融合

gang/vector: 並列化粒度

...

OpenACC: Data Clause

データの移動方法を指示する

```
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);
...
```

copyin: CPU-> GPU

copyout: CPU <- GPU

copy: 両方

create: メモリ確保

...

OpenACC: Data Directive

データ領域を指示する

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

...
#pragma acc data copy(y[:N]) copyin(x[:N])
{
  saxpy(N, 3.0, x, y);
}
...
```

OpenMP と OpenACC の共存

コンパイル時のオプションで切り替え

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

...

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

OpenACC コードのビルド

NVIDIA HPC SDK

- C: `nvc`, C++: `nvc++`, Fortran: `nvfortran`
- `-acc=gpu` : OpenACC を有効にし、NVIDIA GPU 向けにビルド
  - `-acc=multicore` とすることで、マルチコア CPU 向けにもビルド可能
- `-Minfo=accel` : どのように並列化されたかに関する、コンパイラ メッセージを表示
- `-gpu=...` : GPU コード生成に関する詳細を指定
  - `-gpu=managed` とすることで、CUDA Unified Memory を有効化

```
$ nvc -acc=gpu -Minfo=accel -gpu=… saxpy.c
```

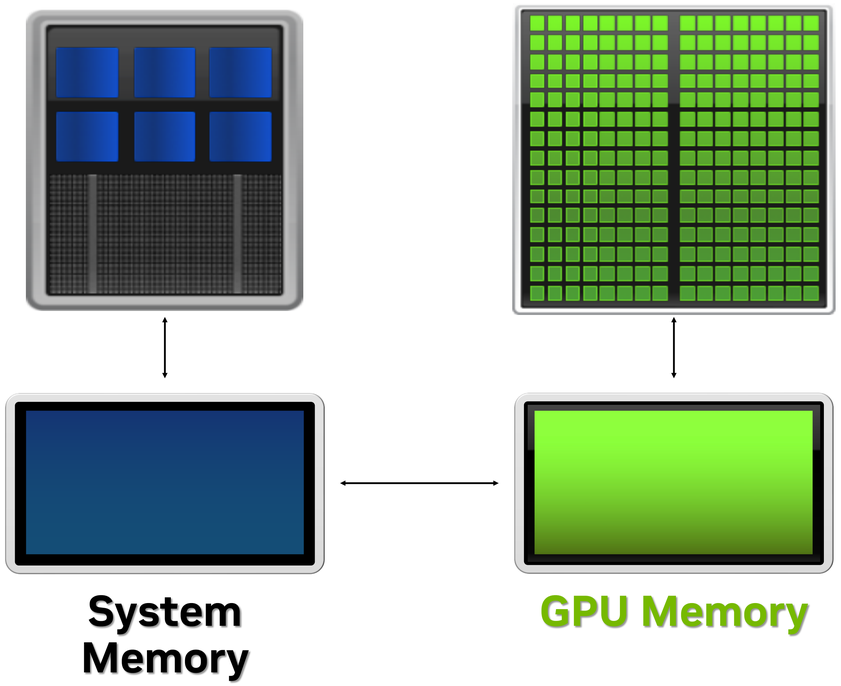
NVIDIA HPC Compilers User's Guide
https://docs.nvidia.com/hpc-sdk/compilers/hpc-compilers-user-guide/index.html

CUDA Unified Memory

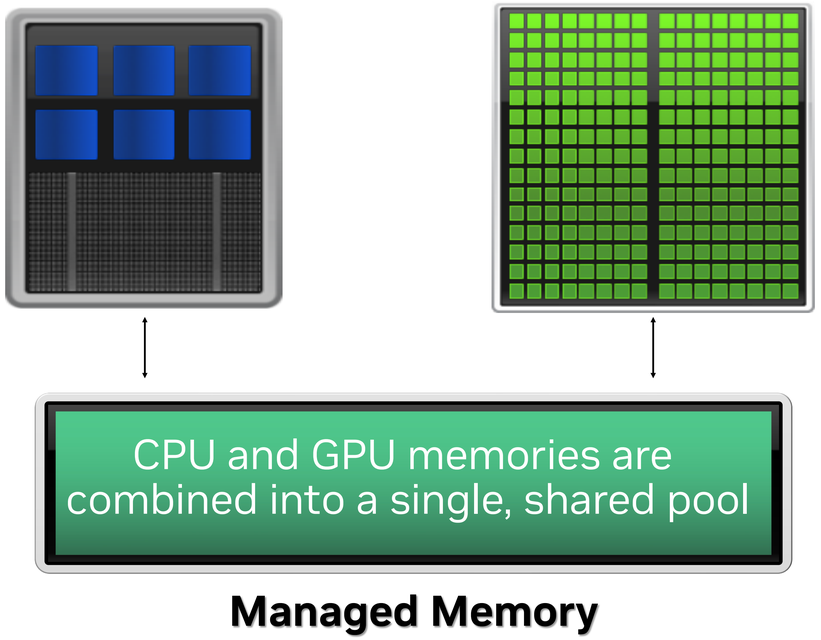
Simplified Developer Effort

Commonly referred to as *"managed memory."*

Without Managed Memory



With Managed Memory



簡単にコンパイル

```
void saxpy(int n, float a,
           float *x, ¥
           float *restrict y)
{
#pragma acc parallel copy(y[:n]) copyin(x[:n])
```

```
$ nvc -acc=gpu -Minfo=accel -gpu=... saxpy.c
saxpy:
      5, Generating copyin(x[:n]) [if not already present]
         Generating copy(y[:n]) [if not already present]
      7, Loop is parallelizable
         Generating NVIDIA GPU code
          7, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
```

```
...

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

簡単に実行

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

```
$ nsys profile --stats=true ./a.out

...
[6/8] Executing 'gpukernsum' stats report

 Time (%)  Total Time (ns)  Instances  Avg (ns)  Med (ns)  Min (ns)  Max (ns)  StdDev (ns)     Name
 --------  ---------------  ---------  --------  --------  --------  --------  -----------  ------------
    100.0            2,528          1   2,528.0   2,528.0     2,528     2,528          0.0  saxpy_7_gpu

[7/8] Executing 'gpumemtimesum' stats report

 Time (%)  Total Time (ns)  Count  Avg (ns)  Med (ns)  Min (ns)  Max (ns)  StdDev (ns)      Operation
 --------  ---------------  -----  --------  --------  --------  --------  -----------  ------------------
     74.8            8,543      2   4,271.5   4,271.5     3,680     4,863        836.5  [CUDA memcpy HtoD]
     25.2            2,880      1   2,880.0   2,880.0     2,880     2,880          0.0  [CUDA memcpy DtoH]
...
```

Nsight Systems Documentation
https://docs.nvidia.com/nsight-systems/UserGuide/index.html

OpenACC が加速する科学技術計算

GAUSSIAN 16

Mike Frisch, Ph.D.
President and CEO
Gaussian, Inc.

"Using OpenACC allowed us to continue development of our fundamental algorithms and software capabilities simultaneously with the GPU-related work. In the end, we could use the same code base for SMP, cluster/ network and GPU parallelism. PGI's compilers were essential to the success of our efforts."

ANSYS FLUENT

Sunil Sathe
Lead Software Developer
ANSYS Fluent

"We've effectively used OpenACC for heterogeneous computing in ANSYS Fluent with impressive performance. We're now applying this work to more of our models and new platforms."

VASP

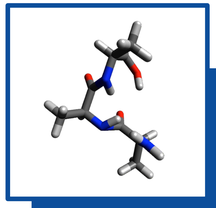
Prof. Georg Kresse
Computational
Materials Physics
University of Vienna

"For VASP, OpenACC is ***the*** way forward for GPU acceleration. Performance is similar and in some cases better than CUDA C, and OpenACC dramatically decreases GPU development and maintenance efforts. We're excited to collaborate with NVIDIA and PGI as an early adopter of CUDA Unified Memory."

OpenACC training materials
https://drive.google.com/drive/folders/1d_eIwIRfScHxfJu6pnR28JrV3cMIwkIL

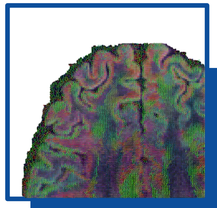
OpenACC が加速する科学技術計算

OPENACC SUCCESSES



LSDalton

Quantum Chemistry
Aarhus University
12X speedup
1 week

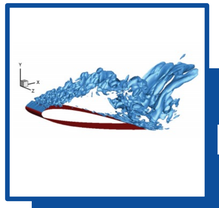


PowerGrid

Medical Imaging
University of Illinois
40 days to
2 hours

COSMO

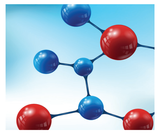
Weather and Climate
MeteoSwiss, CSCS
40X speedup
3X energy efficiency



INCOMP3D

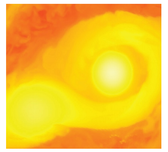
CFD
NC State University

4X speedup



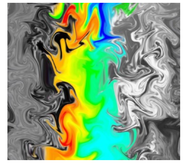
NekCEM

Comp Electromagnetics
Argonne National Lab
2.5X speedup
60% less energy



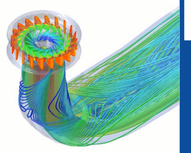
MAESTRO CASTRO

Astrophysics
Stony Brook University
4.4X speedup
4 weeks effort



CloverLeaf

Comp Hydrodynamics
AWE
4X speedup
Single CPU/GPU code



FINE/Turbo

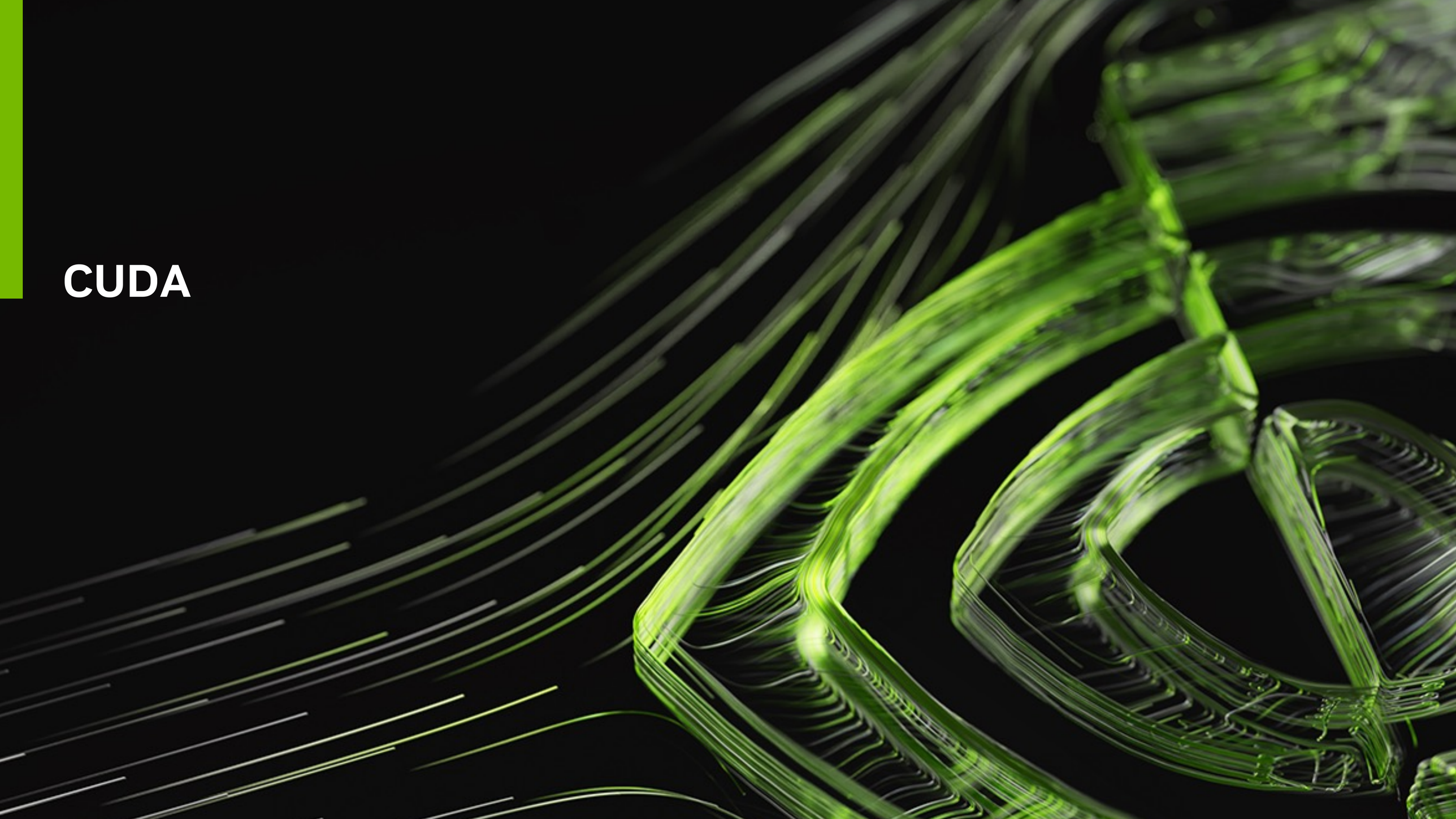
CFD
NUMECA
International
10X faster routines
2X faster app

OpenACC
More Science, Less Programming

OpenACC training materials
https://drive.google.com/drive/folders/1d_eIwIRfScHxfJu6pnR28JrV3cMIwkIL

NVIDIA

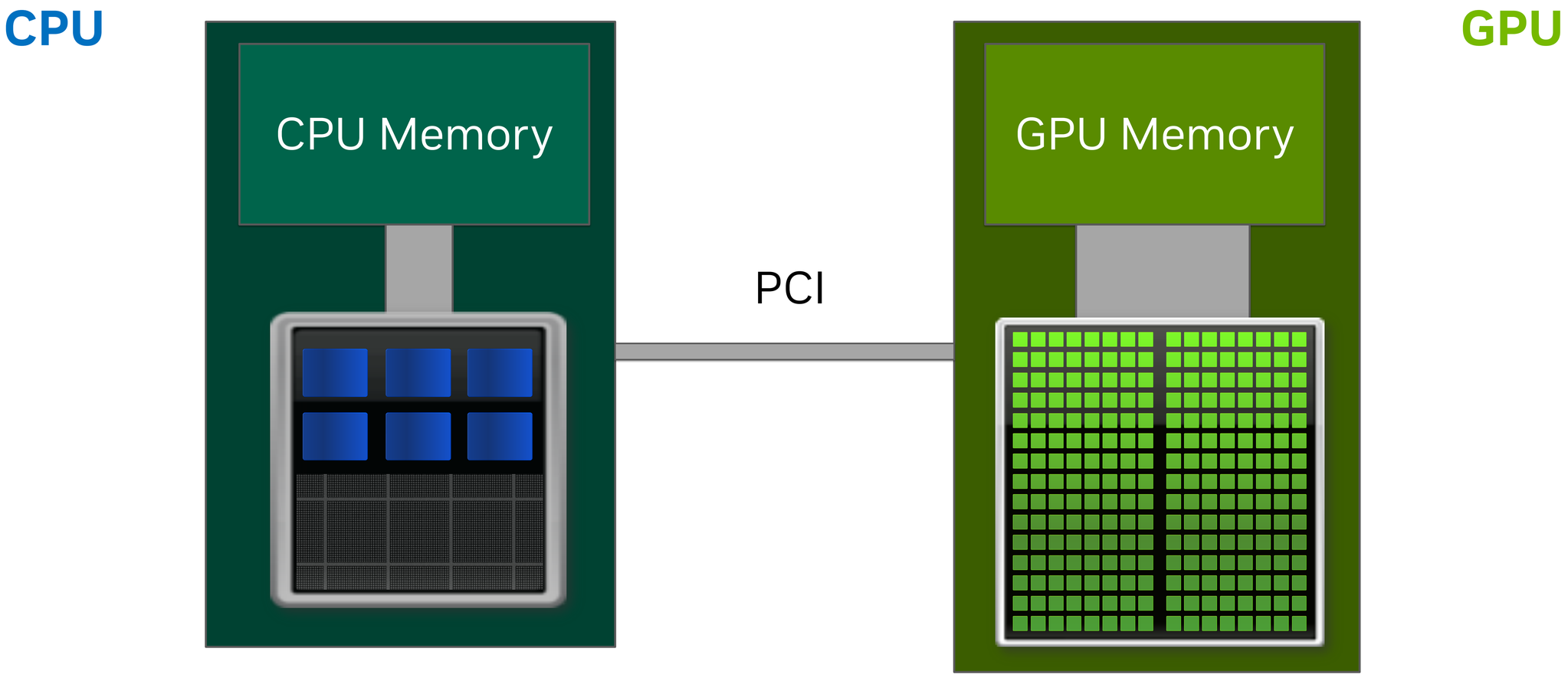
CUDA



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



- 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);
...
```

CUDA

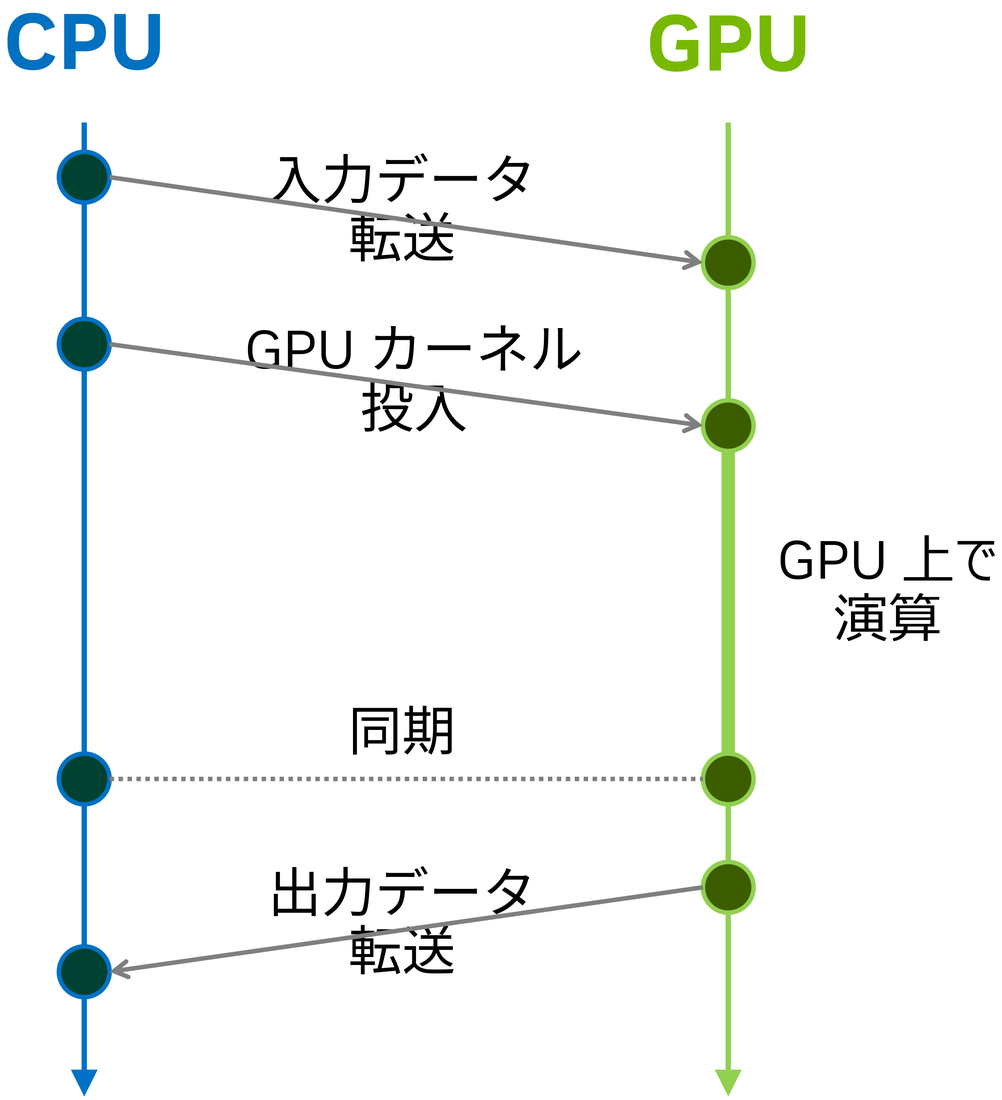
```
__global__ void saxpy(int n, float a,
                      float *x, float *y)
{
  int i = threadIdx.x + blodkDim.x * blockIdx;
  if (i < n)
    y[i] += a*x[i];
}



...

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

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);
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,
...
```

カーネル起動

同期

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];
}

...

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

...
```

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, 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];
}

...
int dev_id = 0;
size_t size = sizeof(float) * N;
cudaMemPrefetchAsync(x, size, dev_id);
cudaMemPrefetchAsync(y, size, dev_id);
saxpy<<< N/128, 128 >>>(N, 3.0, d_x, d_y);
cudaDeviceSynchronize();

...
```

プリフェッチ

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);
...
```

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];
}

...
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;
  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

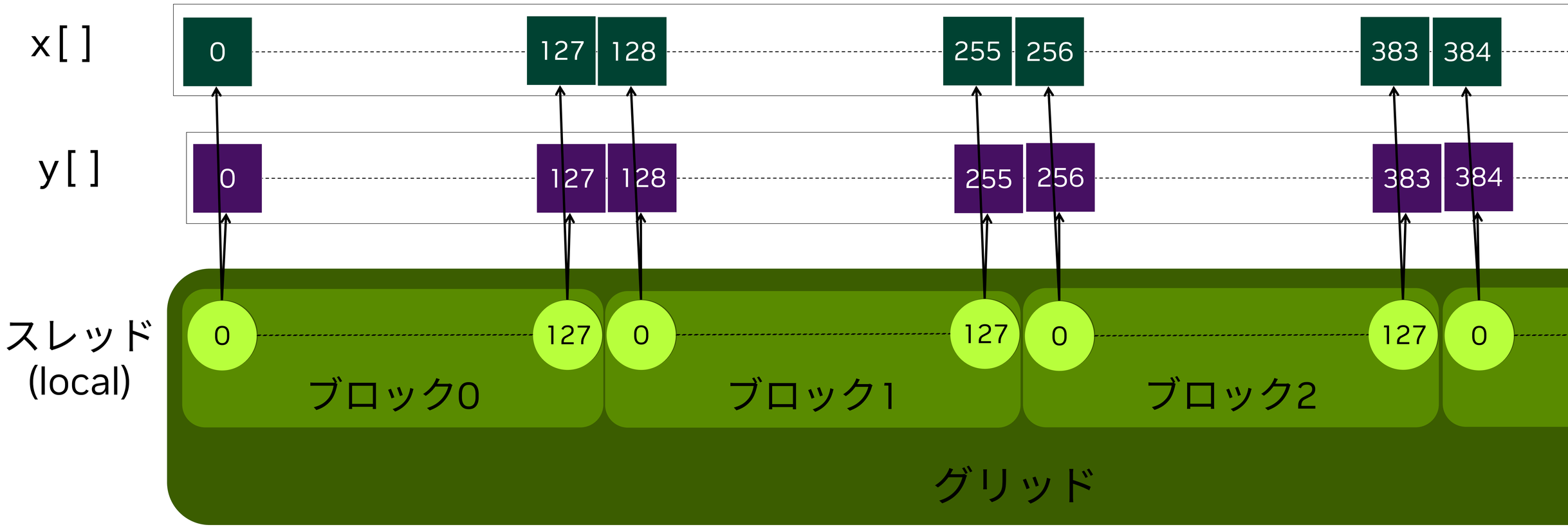
ブロックサイズ

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]
```



- 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;
  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][j] = 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)

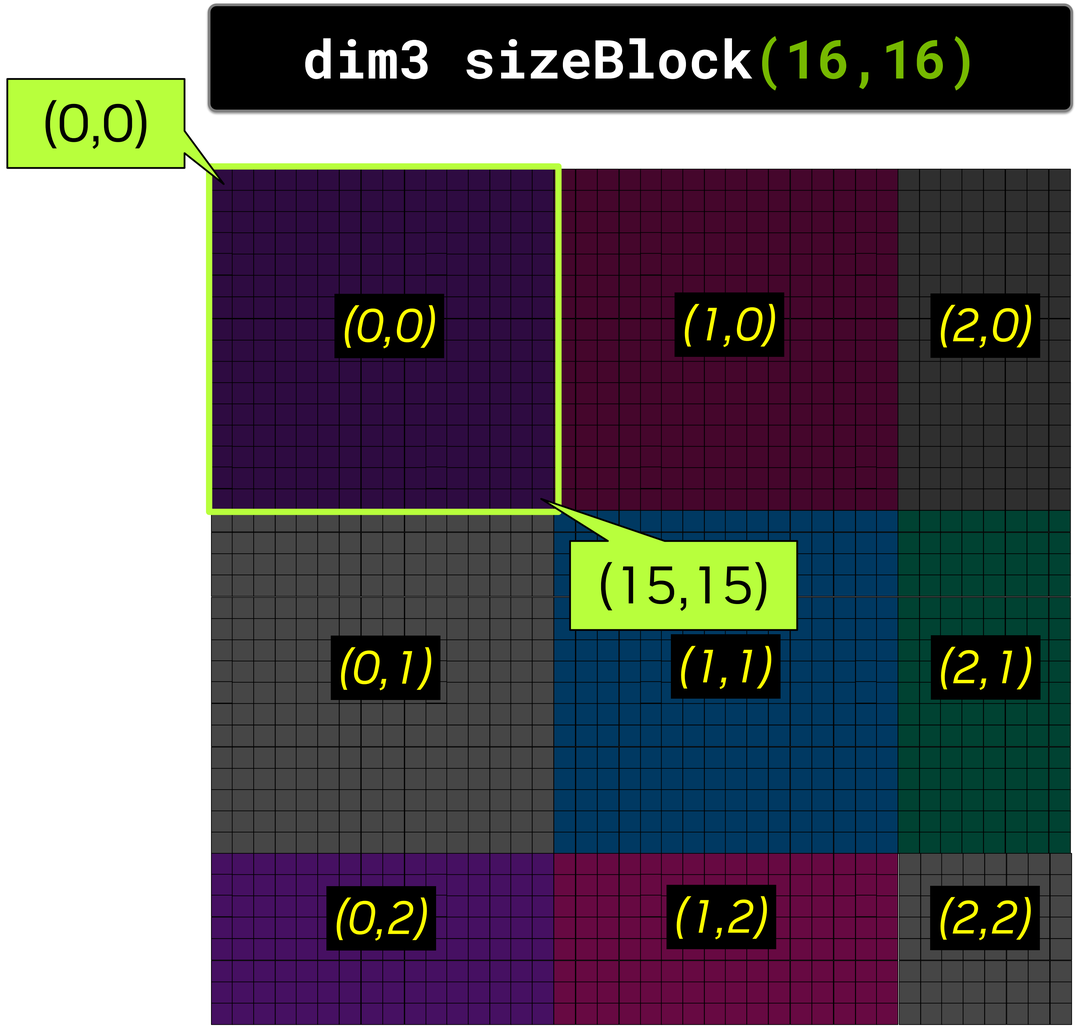
Global スレッド ID (y)

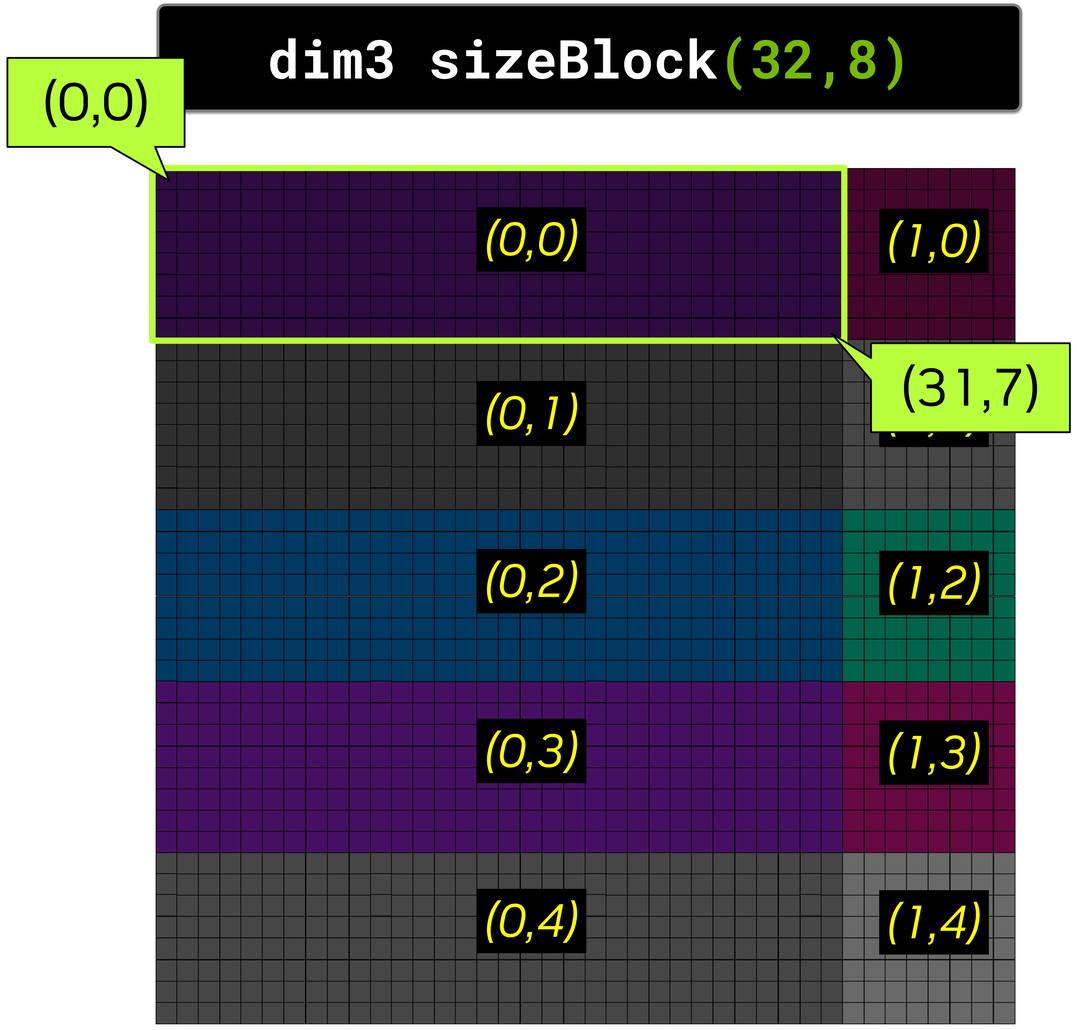
ブロックサイズ (x,y)

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





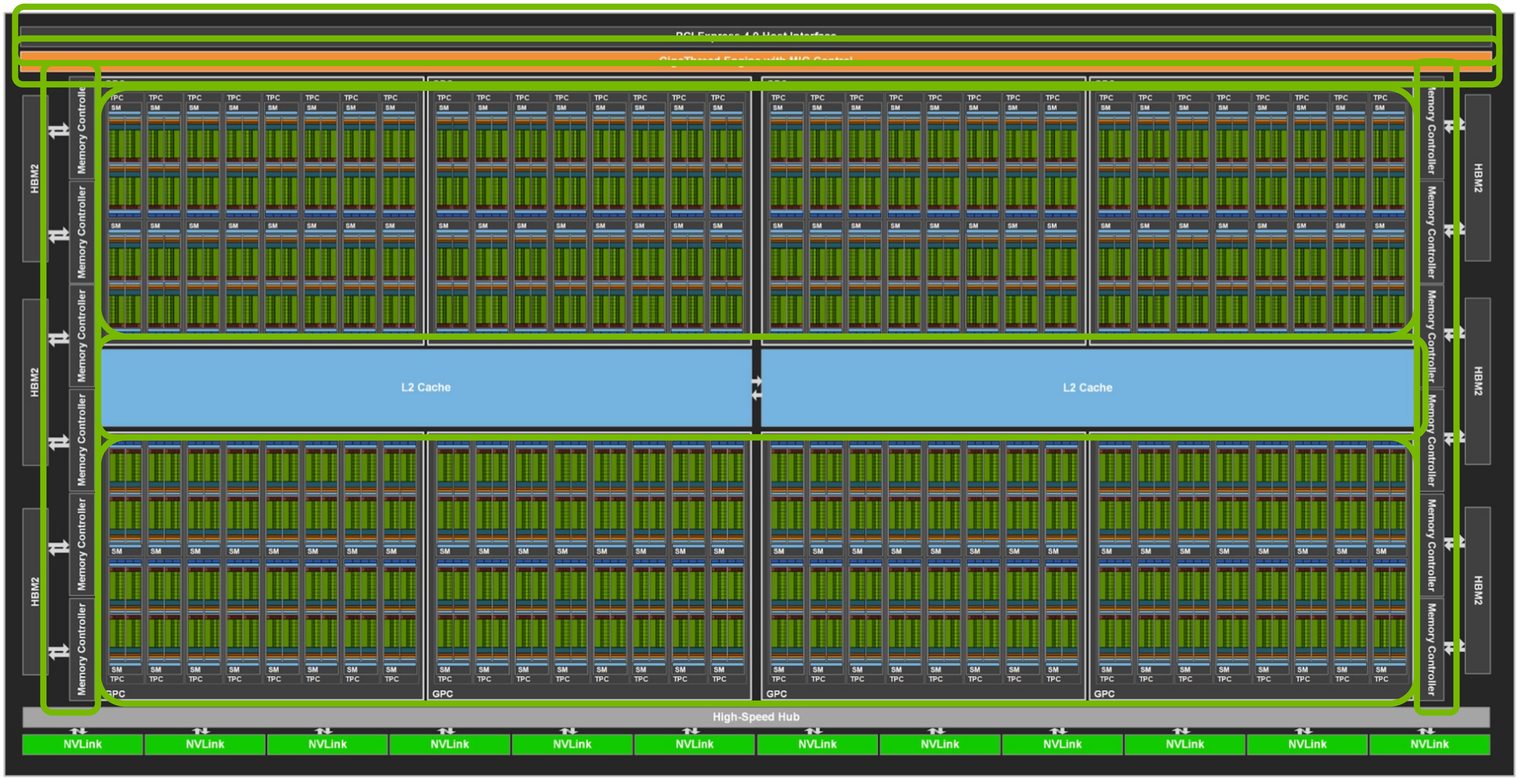
ブロック ID (blockIdx)

スレッド ID (threadIdx)

CUDA プログラミング

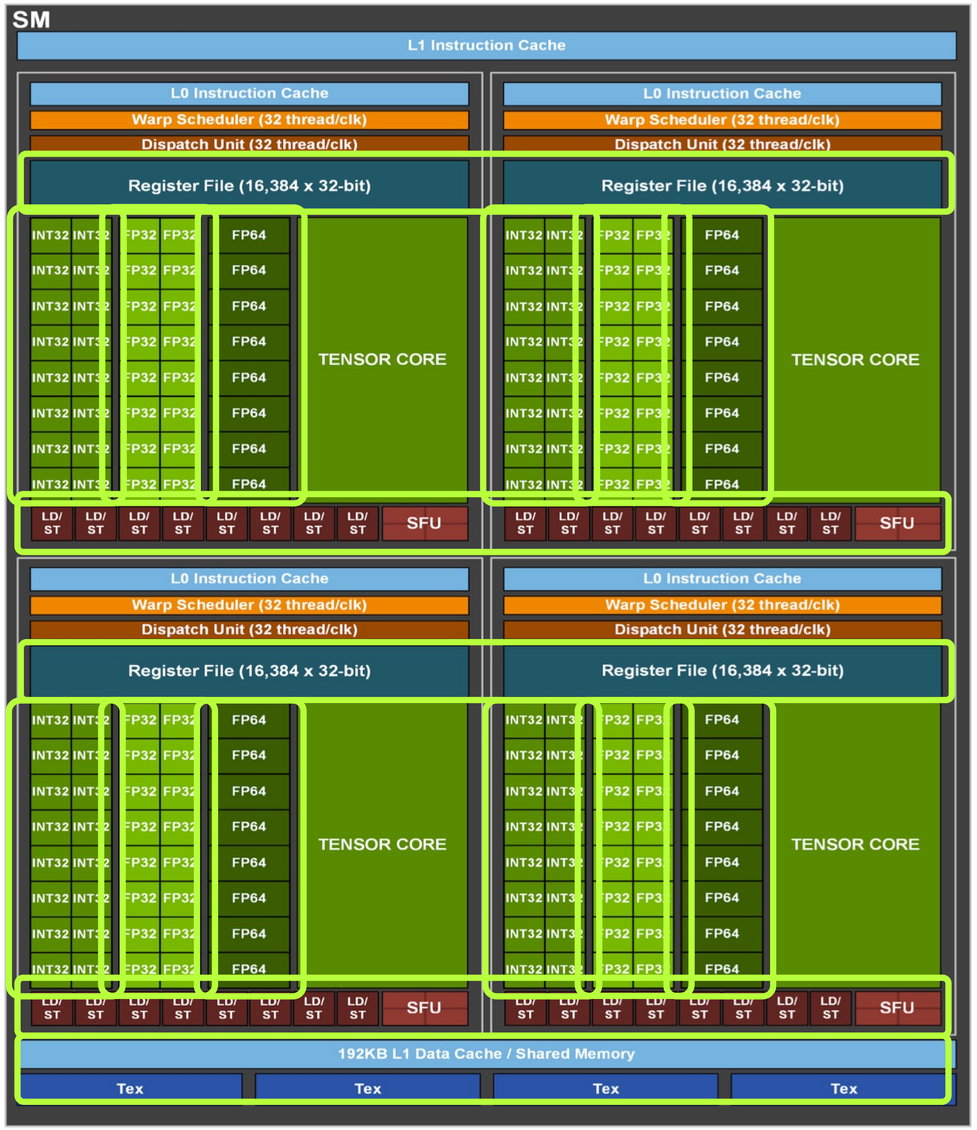
- プログラミングモデル
- アーキテクチャ
- 性能 Tips

GPU アーキテクチャ概要



- 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 (HBM2e)
  - 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 (40 MB)
  - 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、A100 : 108

SM (Streaming Multi-processor)

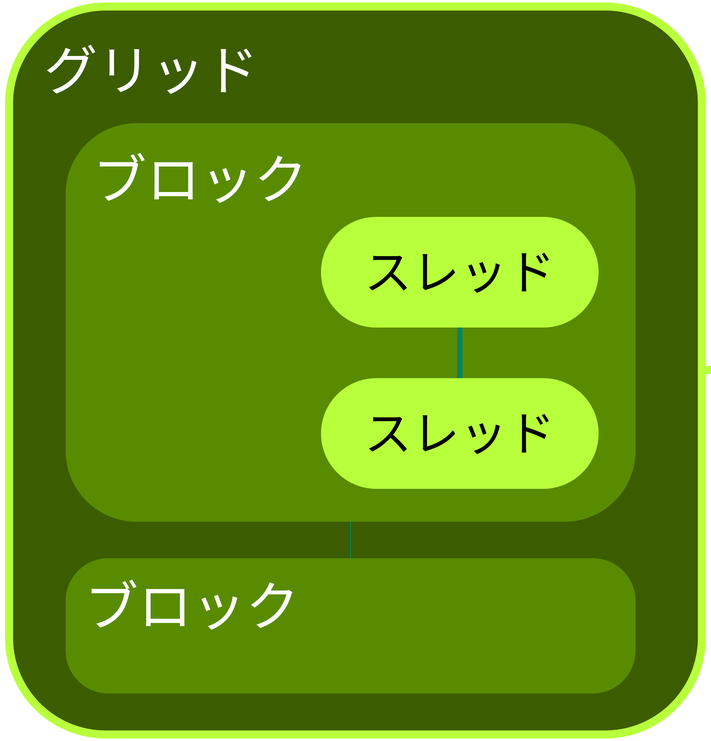


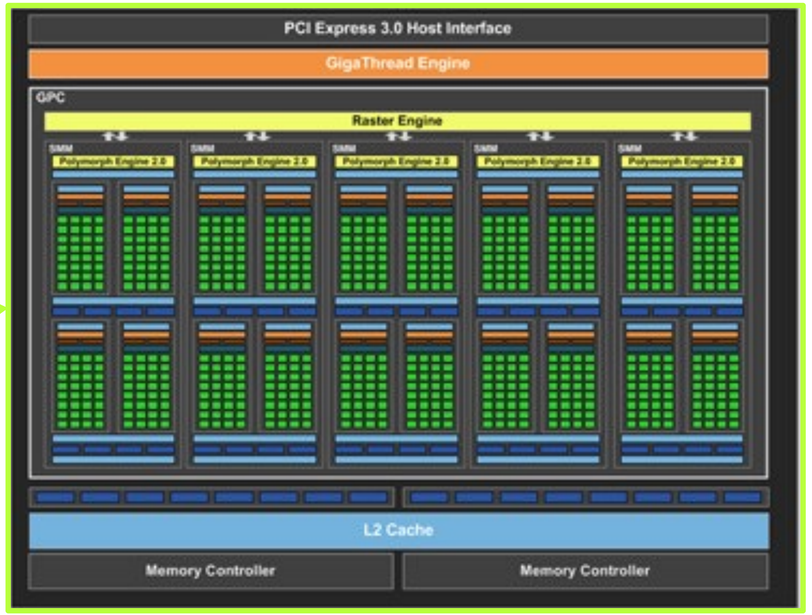
- 演算ユニット
  - INT32: 64 個
  - FP32: 64 個
  - FP64: 32 個
  - TensorCore: 4 個
- Other units
  - LD/ST, SFU, etc

- レジスタ (32 bit): 64K 個
- 共有メモリ/L1 キャッシュ: 192 KB

GPU カーネル実行の流れ

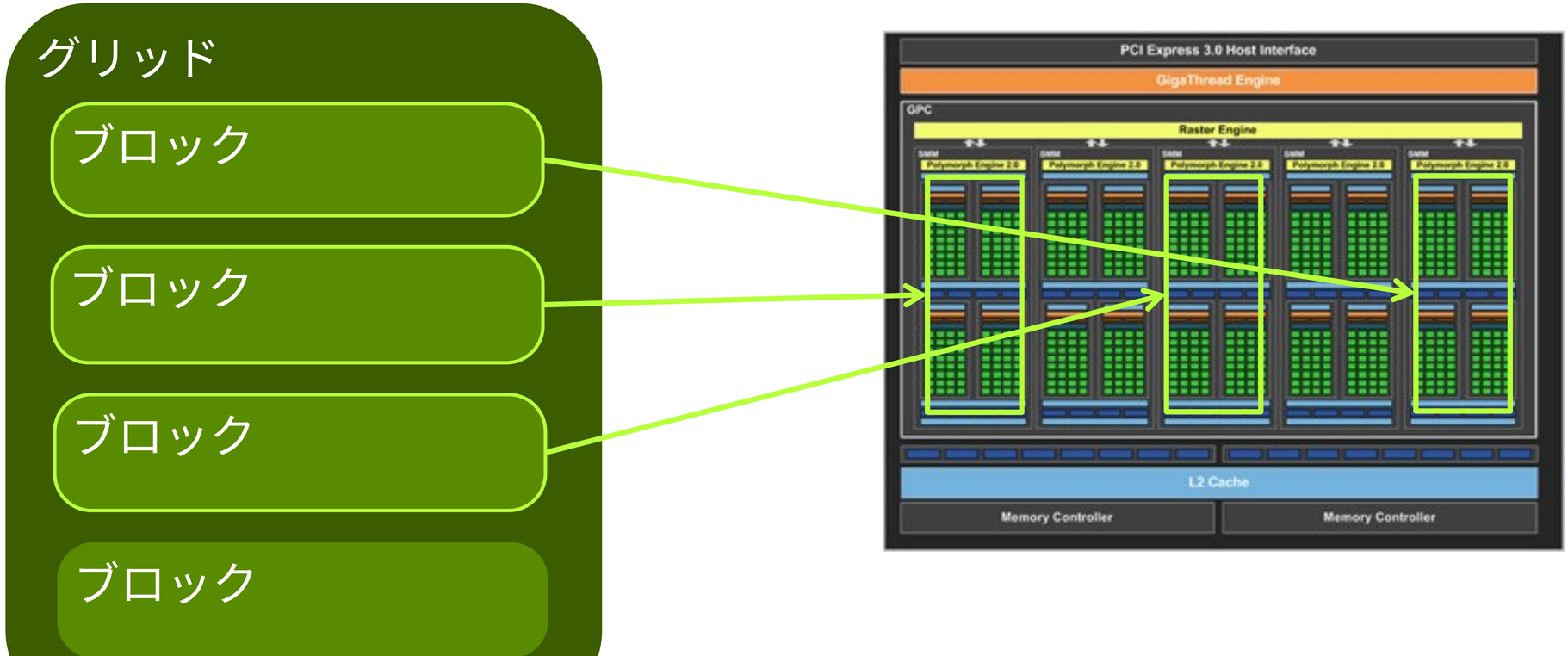
- CPU が、GPU にグリッドを投入
  - 具体的な投入先は、Giga Thread Engine





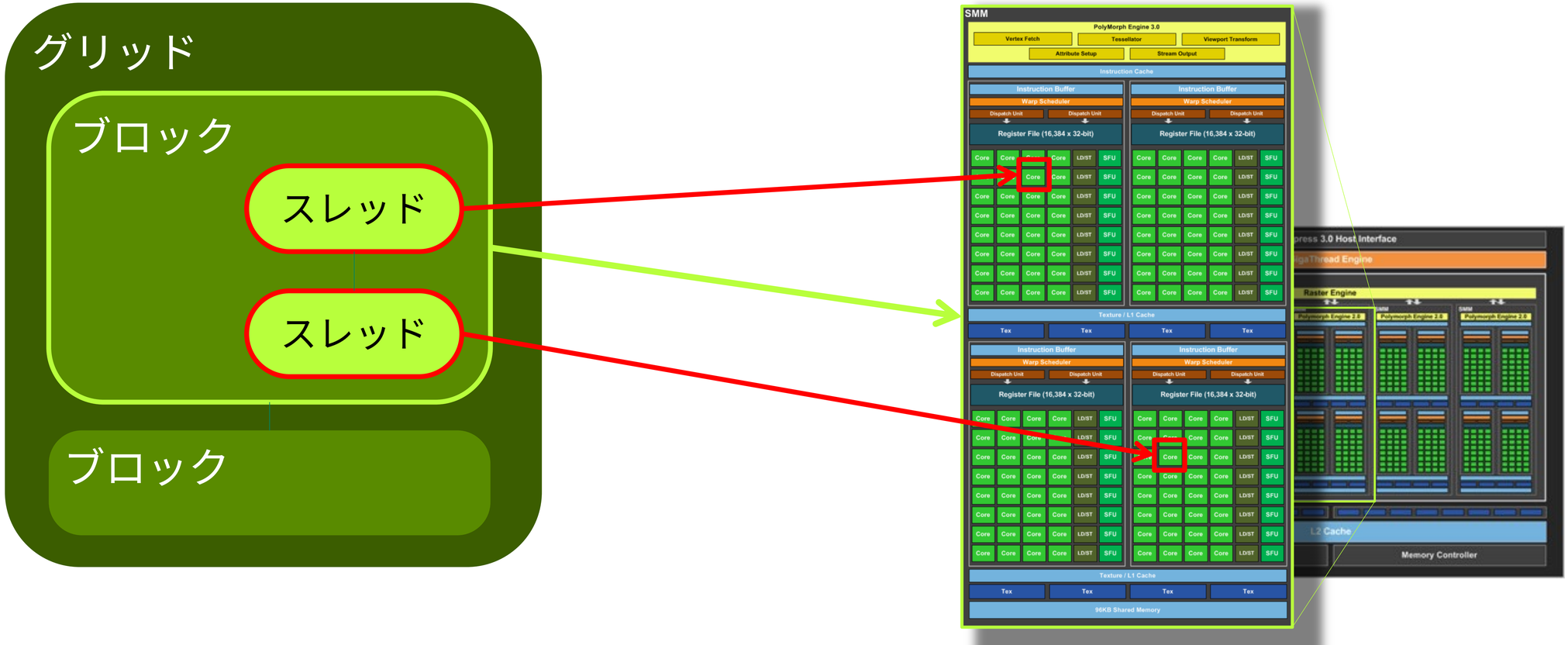
ブロックを SM に割り当て

- 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り当てされることはある



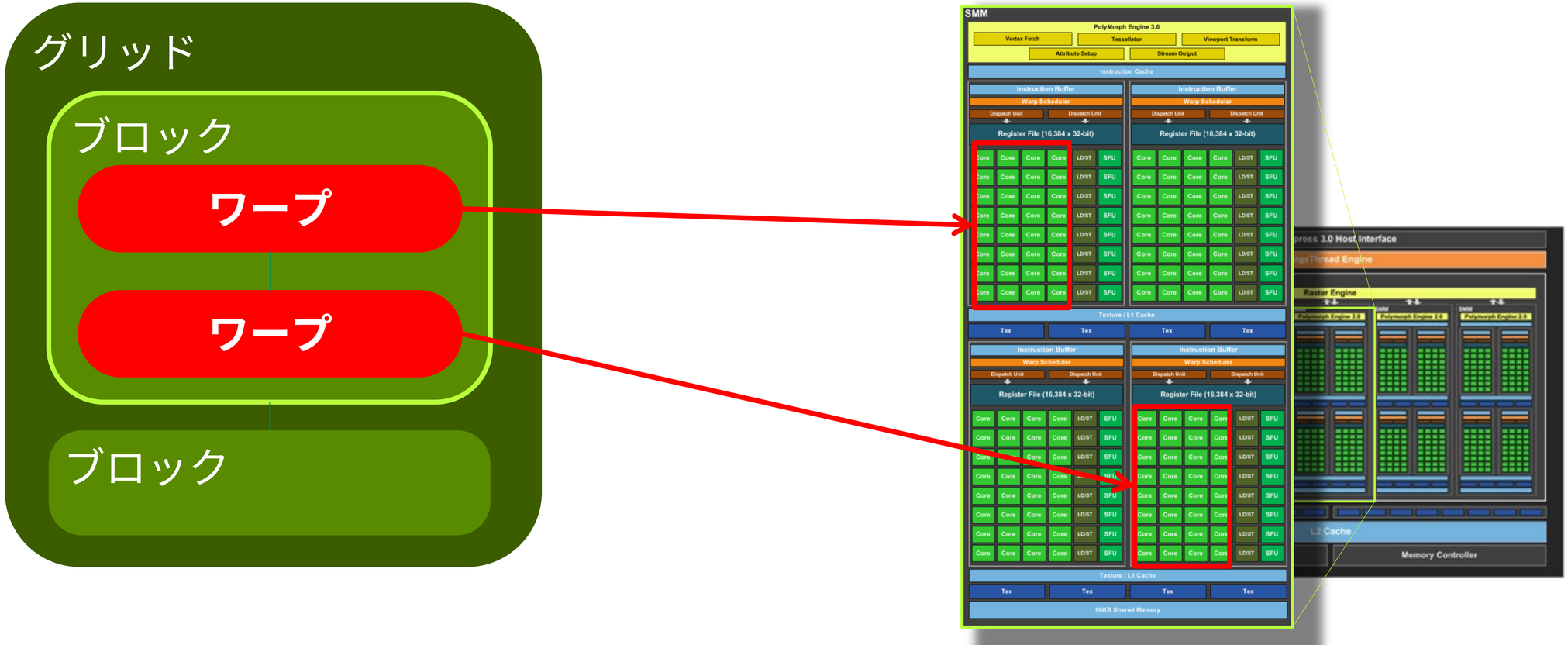
GPU カーネル実行の流れ

- ~~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

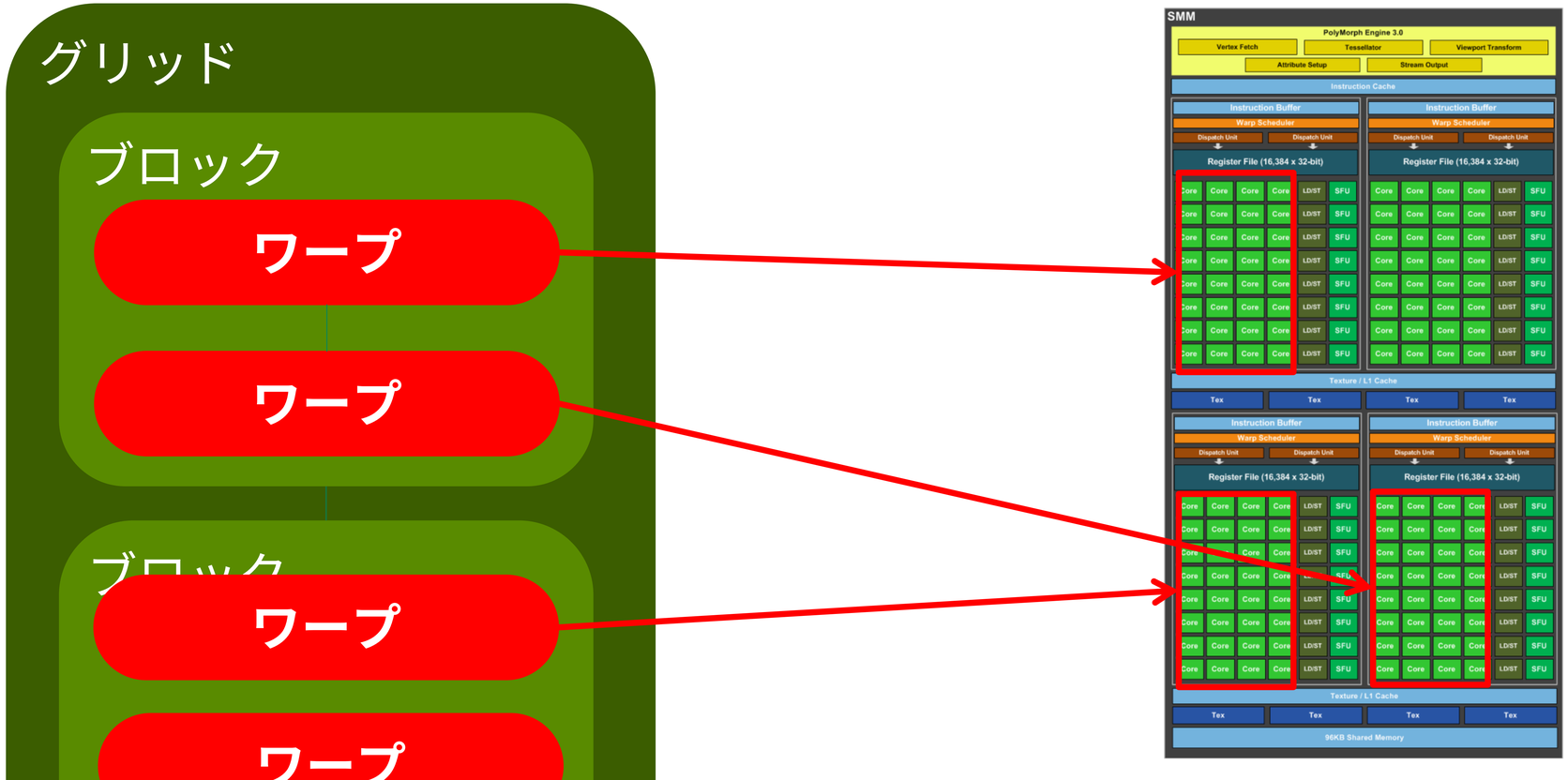
- SM 内のスケジューラが、**ワープ**を CUDA コアに投入
  - ワープ: 32 スレッドの塊
  - ブロックをワープに分割、実行可能なワープを、空 CUDA コアに割り当てる



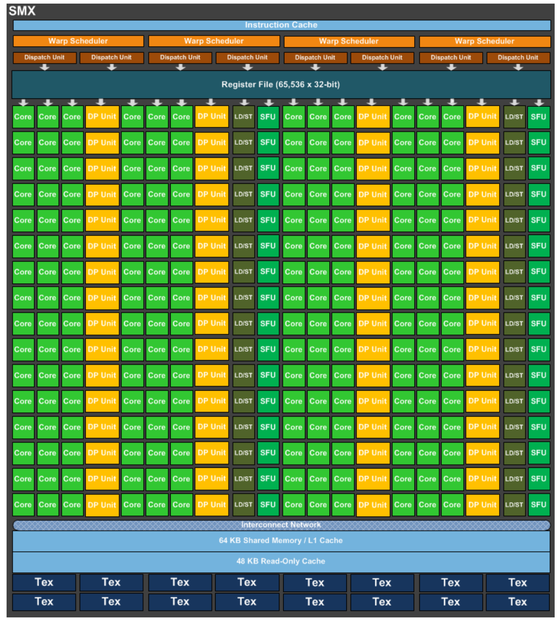
ワープの CUDA コアへの割り当て

- 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)

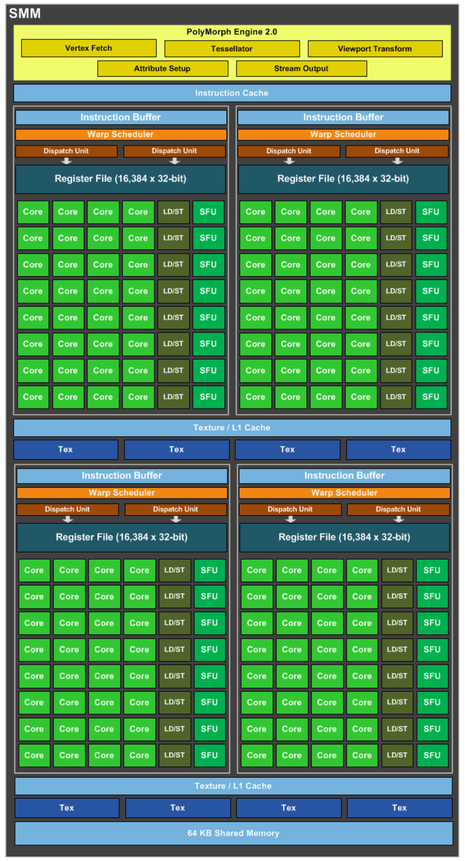


GPU アーキの変化を問題としないプログラミングモデル



Kepler, CC3.5

192 cores /SM



Maxwell, CC5.0

128 cores /SM

Pascal, CC6.0

64 cores /SM

Ampere, CC8.0

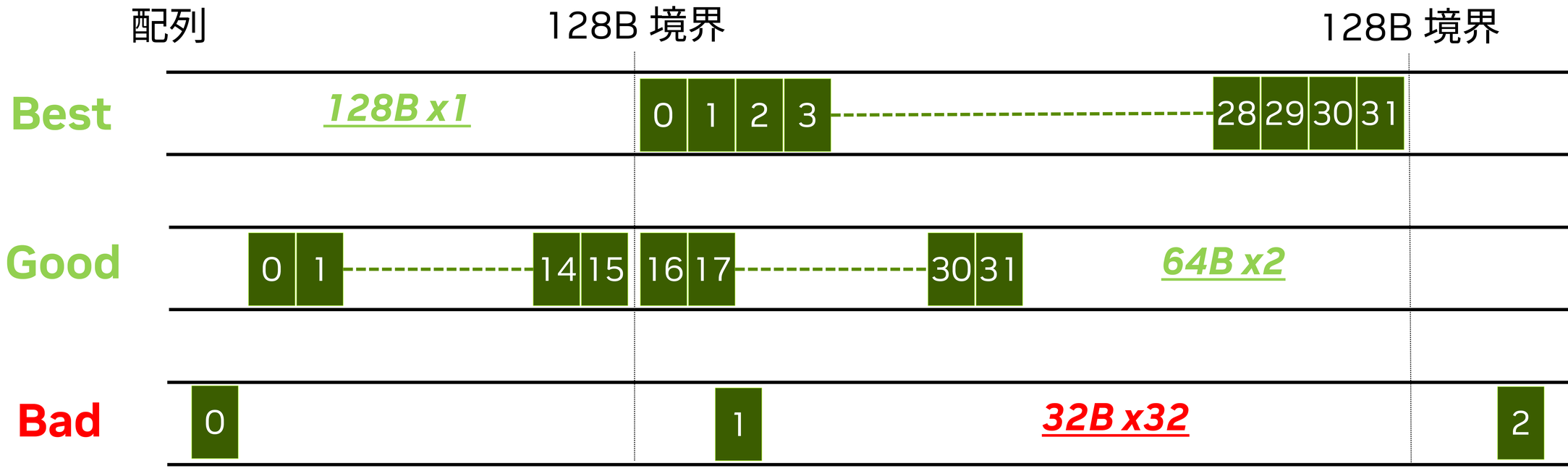
64 cores /SM

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

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、少ないほどよい



リソース使用率 (Occupancy)

SM の利用率を上げる

≒ SM に割り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
  - 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、割り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: 64
最大スレッド数: 2048
最大ブロック数: 32
共有メモリ: 160KB
レジスタ数 (32-bit): 64K 個

リソース量/SM (A100)

リソース使用率 (Occupancy)

SM の利用率を上げる

≒ SM に割り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、割り当て可能なブロック数の関係
    - 32 KB: 2 ブロック, 16 KB: 4 ブロック, 8 KB: 8 ブロック

CUDA コア数: 64
最大スレッド数: 2048
最大ブロック数: 32
共有メモリ: 160KB
レジスタ数 (32-bit): 64K 個

リソース量/SM (A100)

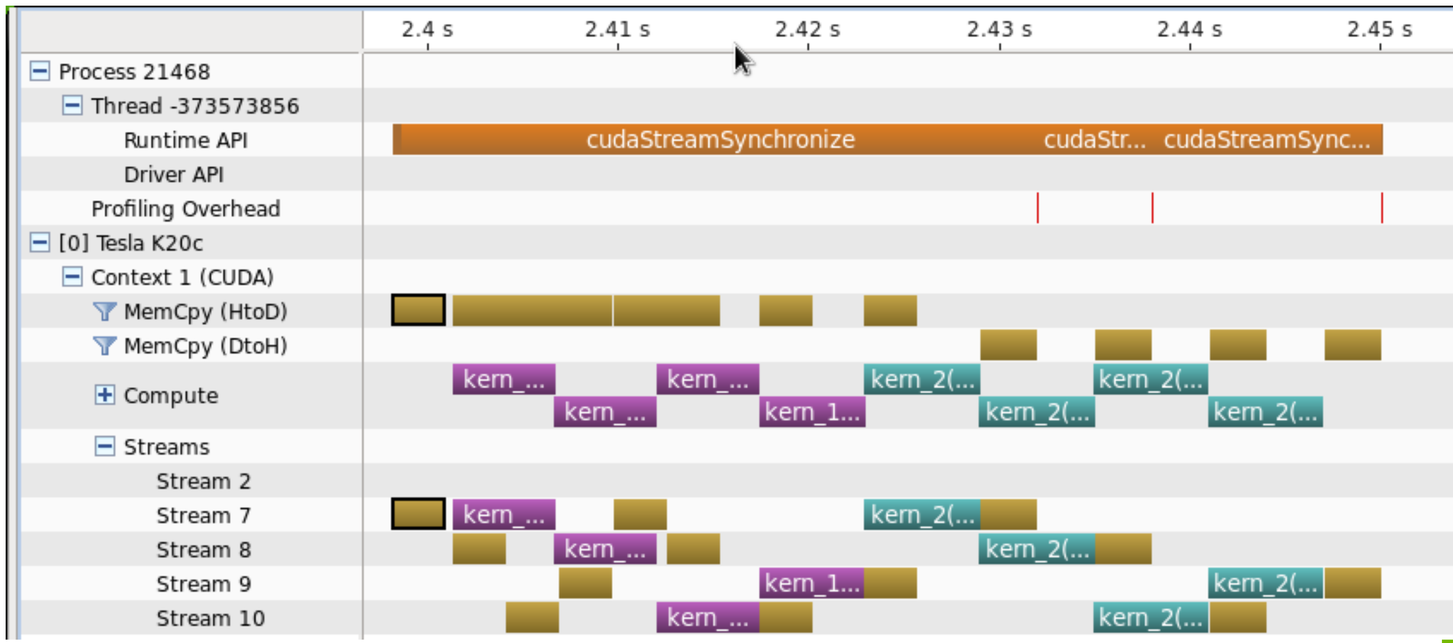
リソース使用率 (Occupancy)

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)

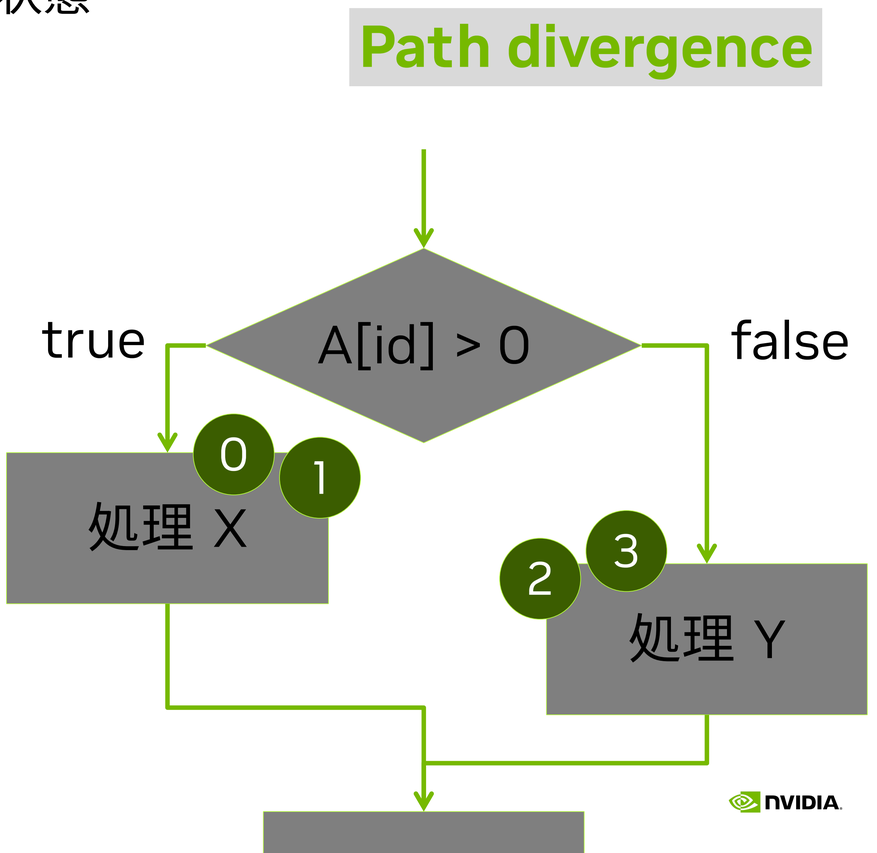
[CUDA ストリームの効果例]
GPU カーネルとデータ転送が
オーバラップして
同時に実行されている



分岐を減らす

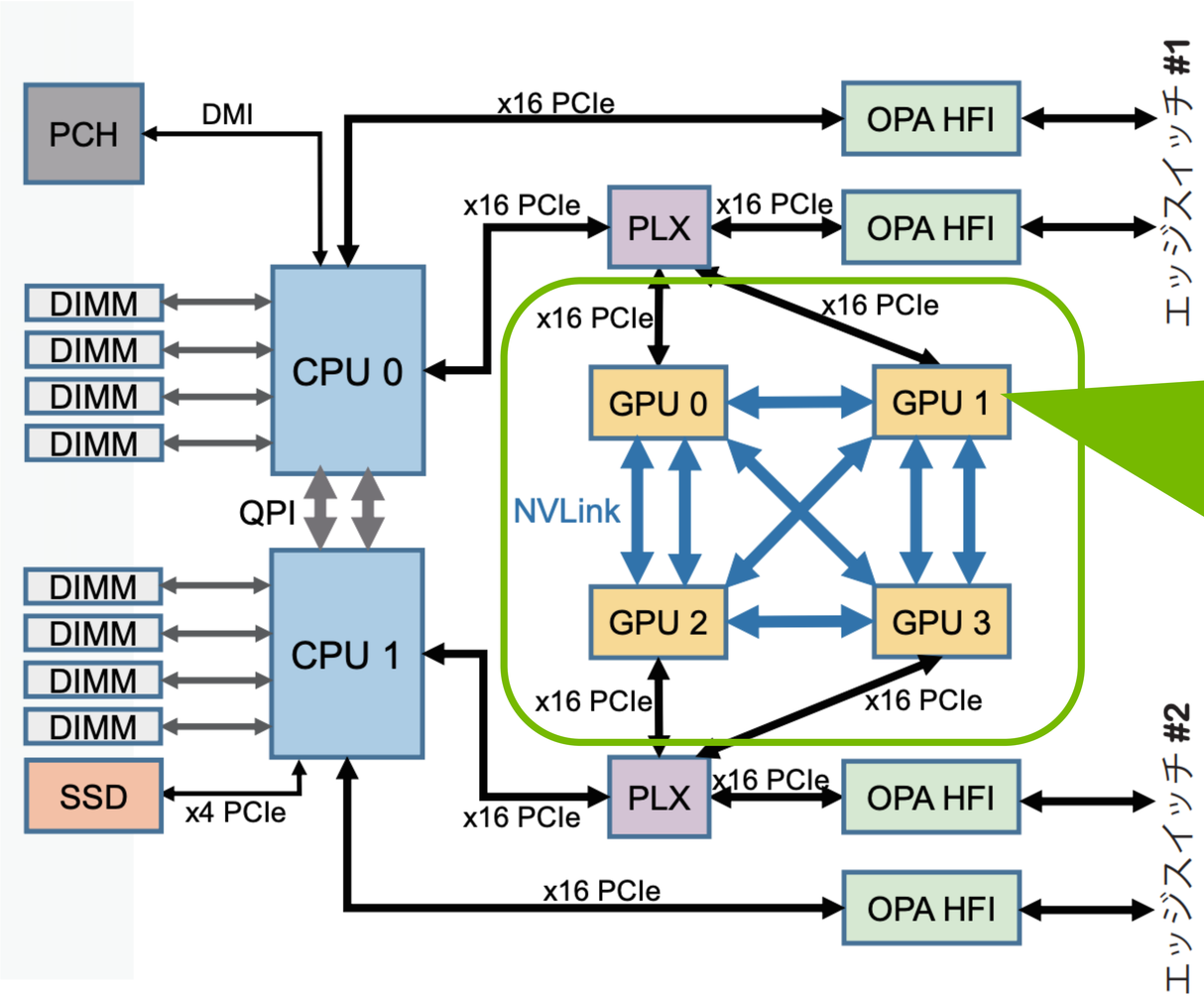
- ワープ内のスレッドが別パスを選択すると遅くなる
  - ワープ内のスレッドは、命令を共有 (SIMT)
  - ワープ内のスレッドが選んだ全パスの命令を実行
  - あるパスの命令を実行中、そのパスにいないスレッドは inactive 状態

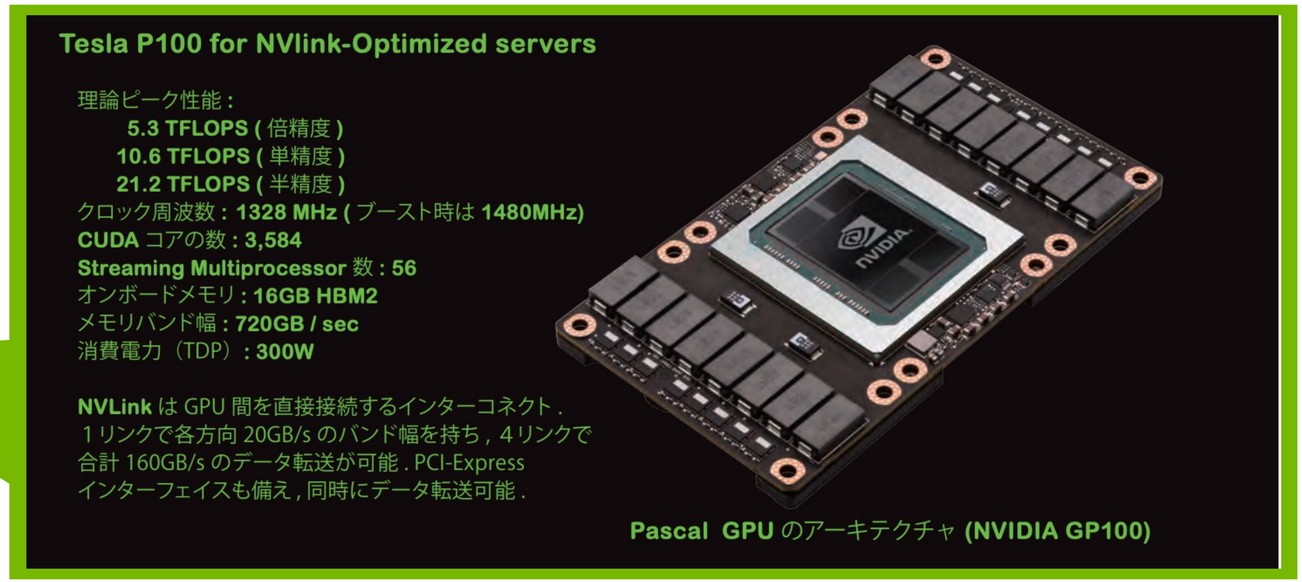
- Path divergence を減らす
  - できる限り、同ワープ内のスレッドは同じパスを選択させる



GPU on TSUBAME3.0 and the latest GPU

TSUBAME3.0 Compute Node



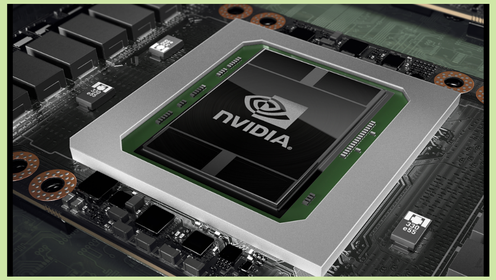


- GPU: Tesla P100
- 4 GPUs / node
- Fully connected by NVLink

https://www.gsic.titech.ac.jp/sites/default/files/spec30j.pdf

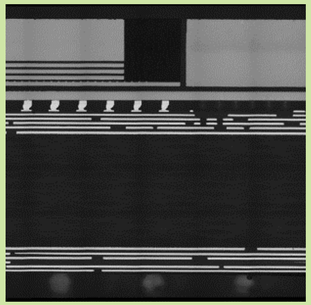
Tesla P100

Pascal アーキテクチャ



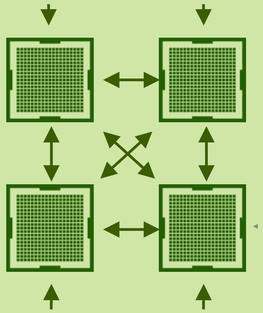
高い演算能力

HBM2 スタックメモリ



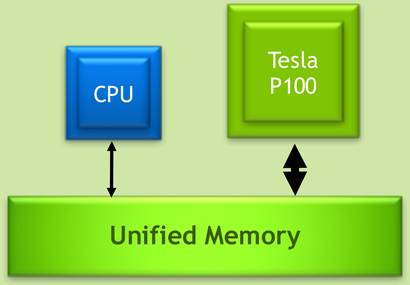
高いメモリバンド幅

NVLink

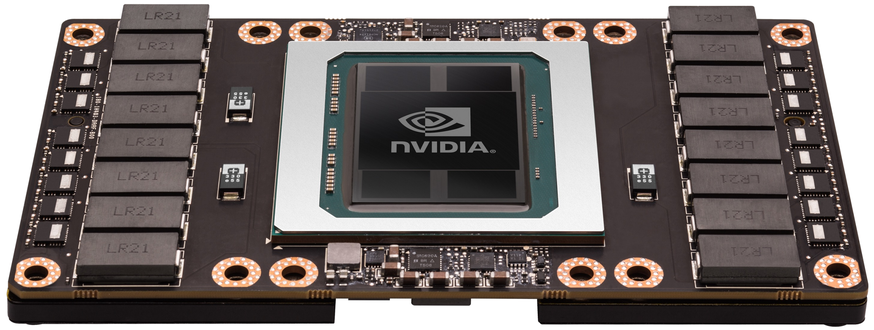


高速なシステム インタコネクト

Page Migration Engine



シンプルなプログラミング



NVIDIA H100

Unprecedented Performance, Scalability, and Security for Every Data Center

Highest AI and HPC Performance

4PF FP8 (6X)| 2PF FP16 (3X)| 1PF TF32 (3X)| 60TF FP64 (3.4X)
3.35TB/s (1.5X), 80GB HBM3 memory

Transformer Model Optimizations

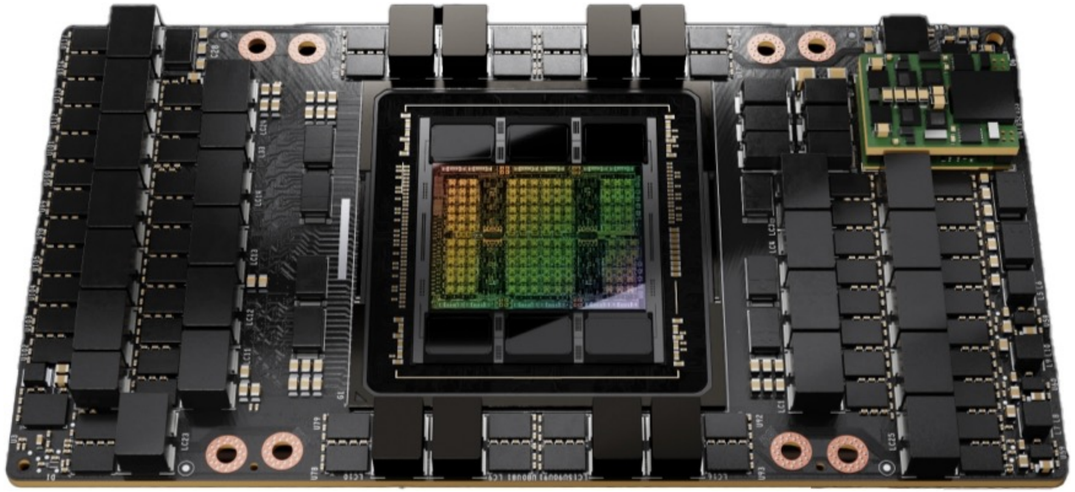
6X faster on largest transformer models

Highest Utilization Efficiency and Security

7 Fully isolated & secured instances, guaranteed QoS
2nd Gen MIG | Confidential Computing

Fastest, Scalable Interconnect

900 GB/s GPU-2-GPU connectivity (1.5X)
up to 256 GPUs with NVLink Switch | 128GB/s PCI Gen5



FP8, FP16, TF32 performance include sparsity. X-factor compared to A100

P100 and H100

	P100 SXM	V100 SXM	A100 SXM	H100 SXM
SMs	56	...	...	132
FP32 Cores / SM	64			128
FP32 Cores / GPU	3584			16896
FP64 Cores / SM	32			64
FP64 Cores / GPU	1792			8448
Tensor Cores / SM	NA			4
Tensor Cores / GPU	NA			528
Peak FP32 TFLOPS	10.6			66.9
Peak FP64 TFLOPS	5.3			33.5
Peak TF32 Tensor TFLOPS	NA			494.7/989.4*
Peak FP64 Tensor TFLOPS	NA			66.9
Memory Bandwidth	732 GB/s			3352 GB/s
L2 Cache Size	4 MB			50 MB

* 疎性なし / 疎性あり

Agenda

- GPU Computing
- OpenACC
- CUDA
- GPU on TSUBAME3.0 and the latest GPU


NVIDIA