NVIDIA GPU開発環境における アクセラレータプログラミング

Shuichi Gojuki, Senior Solution Architect
Solution Architecture & Engineering, NVIDIA G.K.

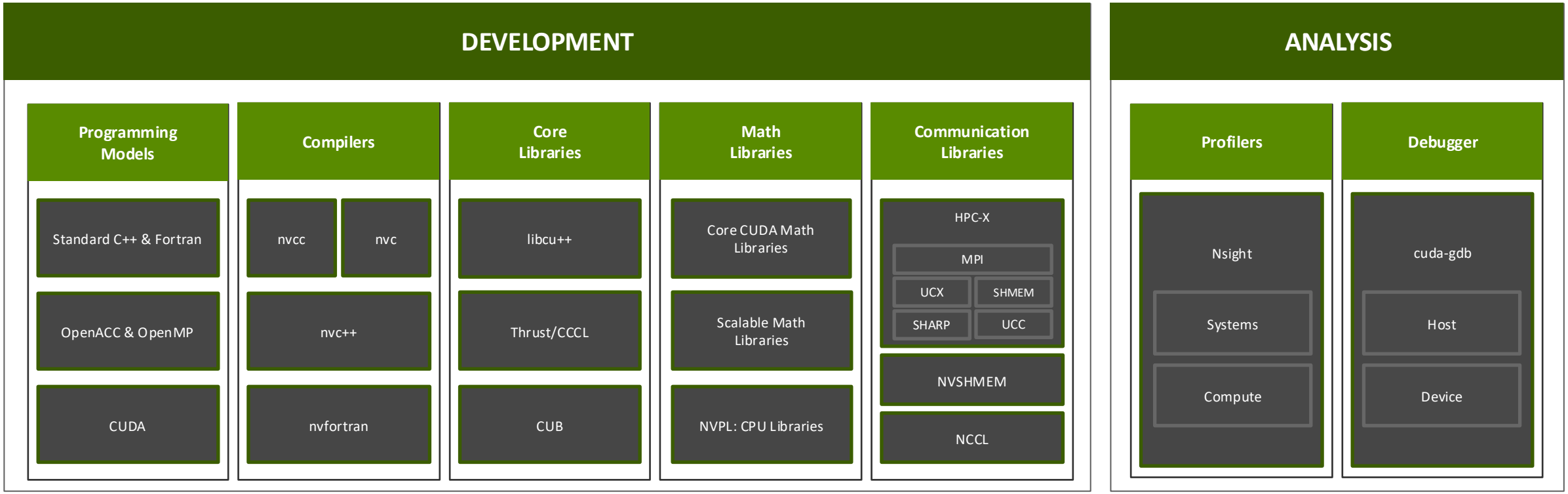
Agenda

- NVIDIA HPC ソフトウェア スタック
- Math ライブラリ
- Developer ツール
- GPUプログラミング

NVIDIA HPC ソフトウェアスタック

NVIDIA HPC SDK

Available at developer.nvidia.com/hpc-sdk, on NGC, via Spack, and in the Cloud



NVIDIAプラットフォームのプログラミング

比類なき開発者の柔軟性

	アクセラレートされた標準言語	段階的な最適化	プラットフォーム特化
Languages & Programming Models	  Fortran	OpenACC More Science, Less Programming OpenMP	NVIDIA CUDA C++ | Fortran | Python
Libraries & Frameworks	アクセラレーション・ライブラリ: Core, Math, Communication, Data Analytics, AI, DSLs		
Hardware	GPU	CPU	Interconnect

実行環境を選ばない

PCs | Workstations | On Prem | Cloud | At the Edge

FortranにおけるGPUプログラミングスタイルの例

Standard Parallelism

```
do concurrent (i = 1:n)
   y(i) = a * x(i) + y(i)
end do
```

OpenACC

```
!$acc data copyin(x) copy(y)
!$acc parallel loop gang vector
do j = 1,n
     y(i) = a * x(i) + y(i)
end do
!$acc end parallel
!$acc end data
```

CUDA

```
attributes(global)
subroutine daxpy(a, x, y )
 …
 i = (blockidx%x - 1) *
          blockdim%x + threadidx%x
 y(i) = a * x(i) + y(i)
end subroutine


 …
real :: x(N), y(N)
real, device :: xd(N), yd(N)
…
xd = x
yd = y
call
  daxpy<<<n/64, 64>>>(a, xd, yd)
y = yd
```

記述的 => 生産性／可搬性
アプリケーションに関連する特性を強調

指示的 => 性能重視
コンパイラに実行方法を指示する

Standard Parallel Languages (StdPar)

標準言語を用いて CPU と GPU を高い生産性でプログラミング

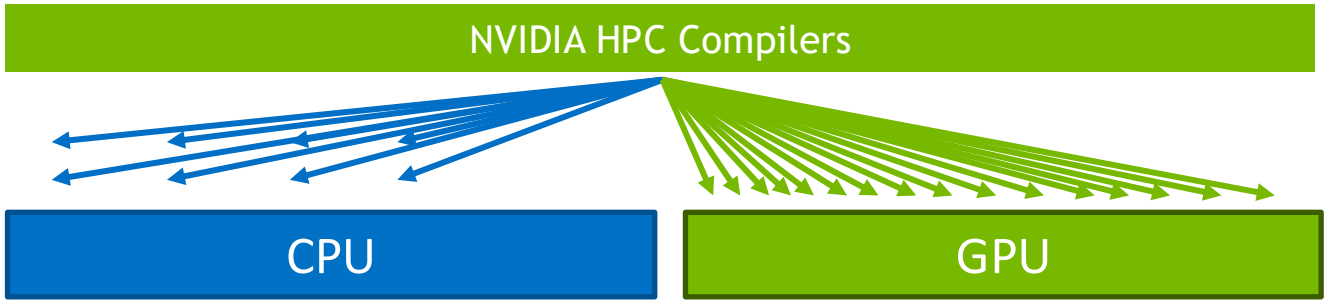
- 2020 年以降、NVIDIA HPC コンパイラは**標準 C++** および **Fortran** の並列化とオフロードをサポートしています。
  - C++ 並列アルゴリズム
  - Fortran の do concurrent 構文および配列組み込み演算
- NVIDIA は ISO 標準言語に対して継続的かつ積極的に投資しています
  - 各言語コミュニティのアクティブなメンバーとして活動しています
  - NVIDIA HPC コンパイラにおいて継続的な開発を行っています
  - LLVM におけるオープンソース開発を推進しています
- **HPC コンパイラ：Stdpar 機能ハイライト**
  - ユニファイドメモリ（Unified Memory）モードの改善
  - C++ における OpenACC との相互運用性（Interop）の改善
  - Stdpar の GPU／マルチコア対応ユニファイド・バイナリ（単一バイナリ）（-stdpar=gpu,multicore）
  - C++23 および C++26 のサポート：mdspan、線形代数、リフレクション、std::execution、拡張浮動小数点型 など
  - Fortran 2023 の reduction（リダクション）サポート

ISO C++

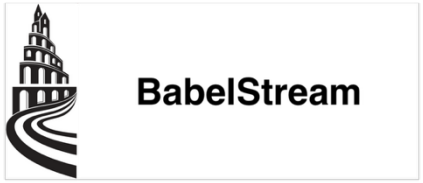
```
std::transform(par, x, x+n, y,
   y,[=](float x, float y){
      return y + a*x;
    }
);
```

ISO Fortran

```
do concurrent (i = 1:n)
   y(i) = y(i) + a*x(i)
enddo
```



標準言語における並列プログラミングモデル(Stdpar) の普及を加速

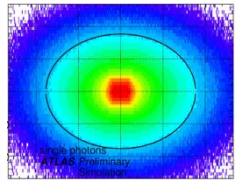


BabelStream

Cloverleaf

Diffuse

Echo

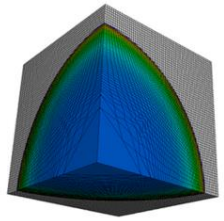


FastCaloSim

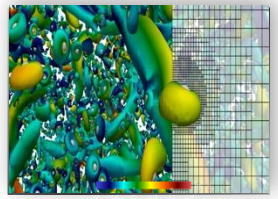
GAMESS

HipFT

JAEA MiniApps

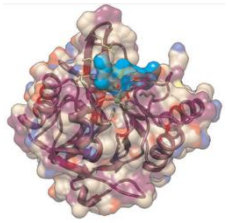


Lulesh

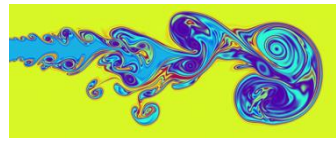


M-AIA

MAS



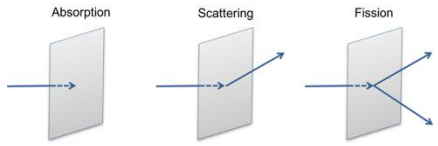
miniBUDE



MiniWeather

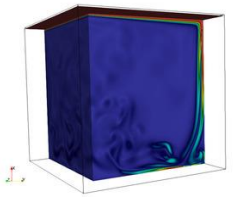
Palabos

POT3D



Quicksilver

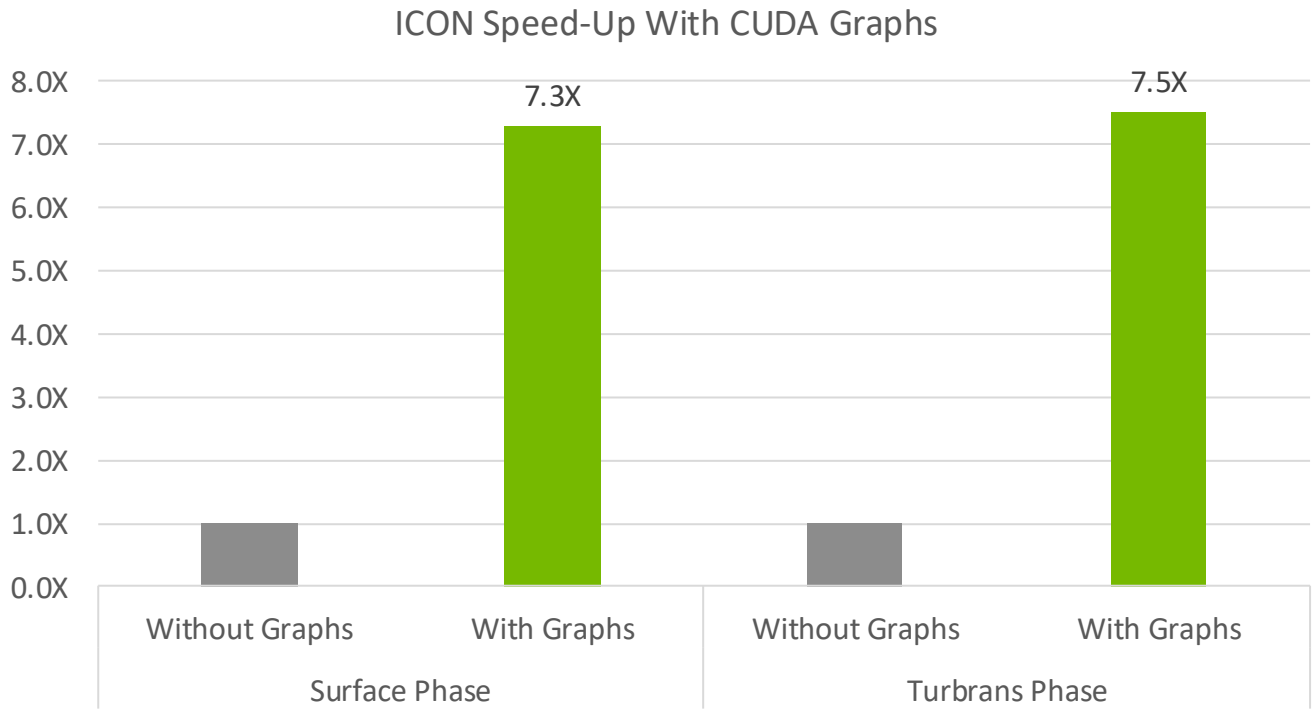
RAJAPerf



STLBM

ICON

OpenACC と CUDA Graphs の組み合わせにより、ICON の実行効率が向上



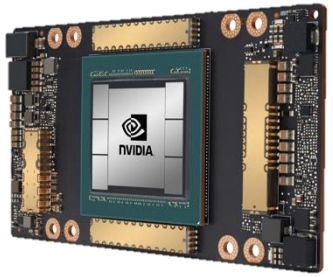
ICON 数値気象予報モデルにおける性能向上

Speed-Up Source: ICON-22 Switzerland weather prediction by MeteoSwiss. HPC SDK 24.3. 8x NVIDIA A100 GPUs

- **OpenACC** により、ICON の開発者は容易にコードを GPU 対応（GPU 有効化）できる。
- **CUDA Graphs** は再利用可能なタスクグラフを構築することで、GPU 利用効率を最適化する。
- **CUDA Graphs と OpenACC の組み合わせ**により、アプリケーションの特定フェーズで最大 **7.5 倍**の改善、アプリケーション全体の実行時間でも約 **10%** の改善が得られた。

NVIDIA HPC コンパイラ

NVC | NVC++ | NVFORTRAN



Accelerated

Grace Hopper
Automatic

Fortran

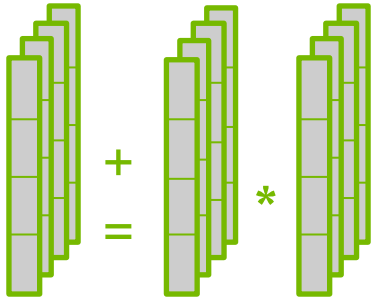
NVIDIA CUDA

OpenACC
More Science, Less Programming

OpenMP

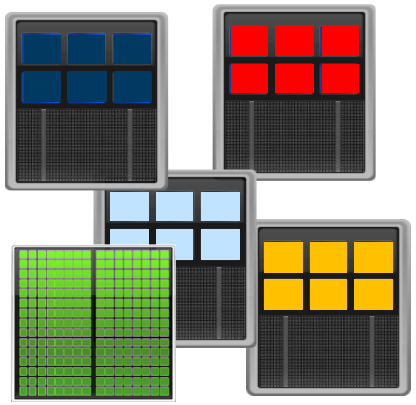
Flexible

Standard Languages
Directives
CUDA



CPU Optimized

Directives
Multi-core
Vectorization

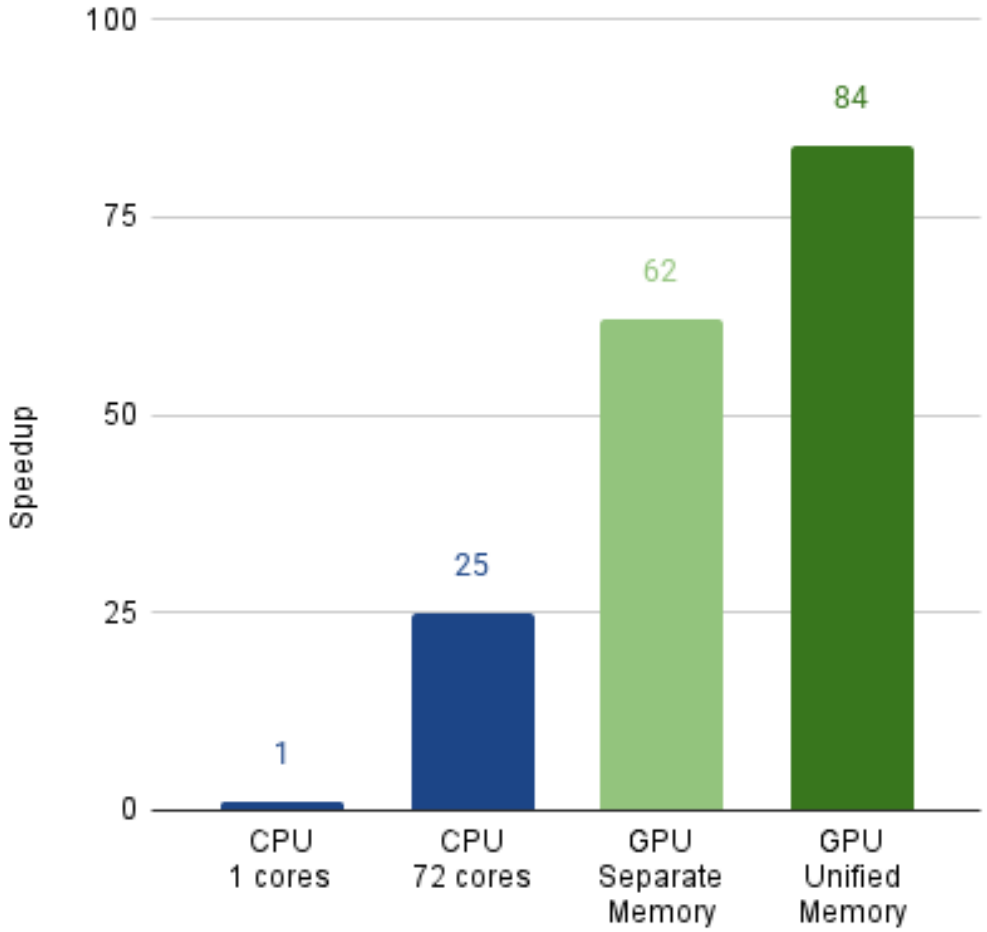


Multi-Platform

NVIDIA GPUs
x86_64
AArch64

nvfortran パフォーマンス

実アプリケーションにおける NEMO 海洋シミュレーションの高速化



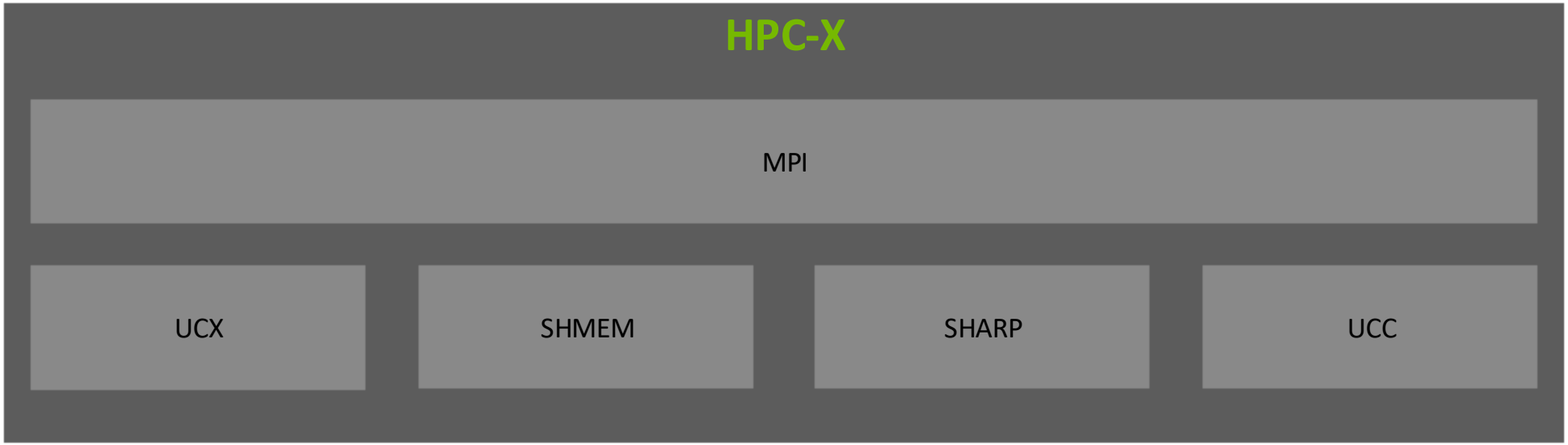
Performance of ice advection relative to CPU single core execution on Grace-Hopper (480 GB), NVHPC SDK 25.1, CUDA 12.6

- **NEMO**（**N**ucleus for **E**uropean **M**odelling of the **O**cean）は、海洋・気候科学分野における研究活動や予報業務で利用されている、最先端のモデリングフレームワークです。https://forge.nemo-ocean.eu/nemo/nemo
- NEMO v4.0.7 (運用コード)
- **Barcelona Supercomputing Centre** において移植作業を進行中
- MPI と OpenMP による CPU 並列化・高速化
- OpenACC を用いた GPU アクセラレーション
  - ユニファイドメモリ版および分離メモリ版のコードを提供
  - 現時点では、氷輸送(ice advection)のみが部分的にGPUオフロードされている

HPC-X

メッセージング通信におけるスケーラビリティと性能の向上

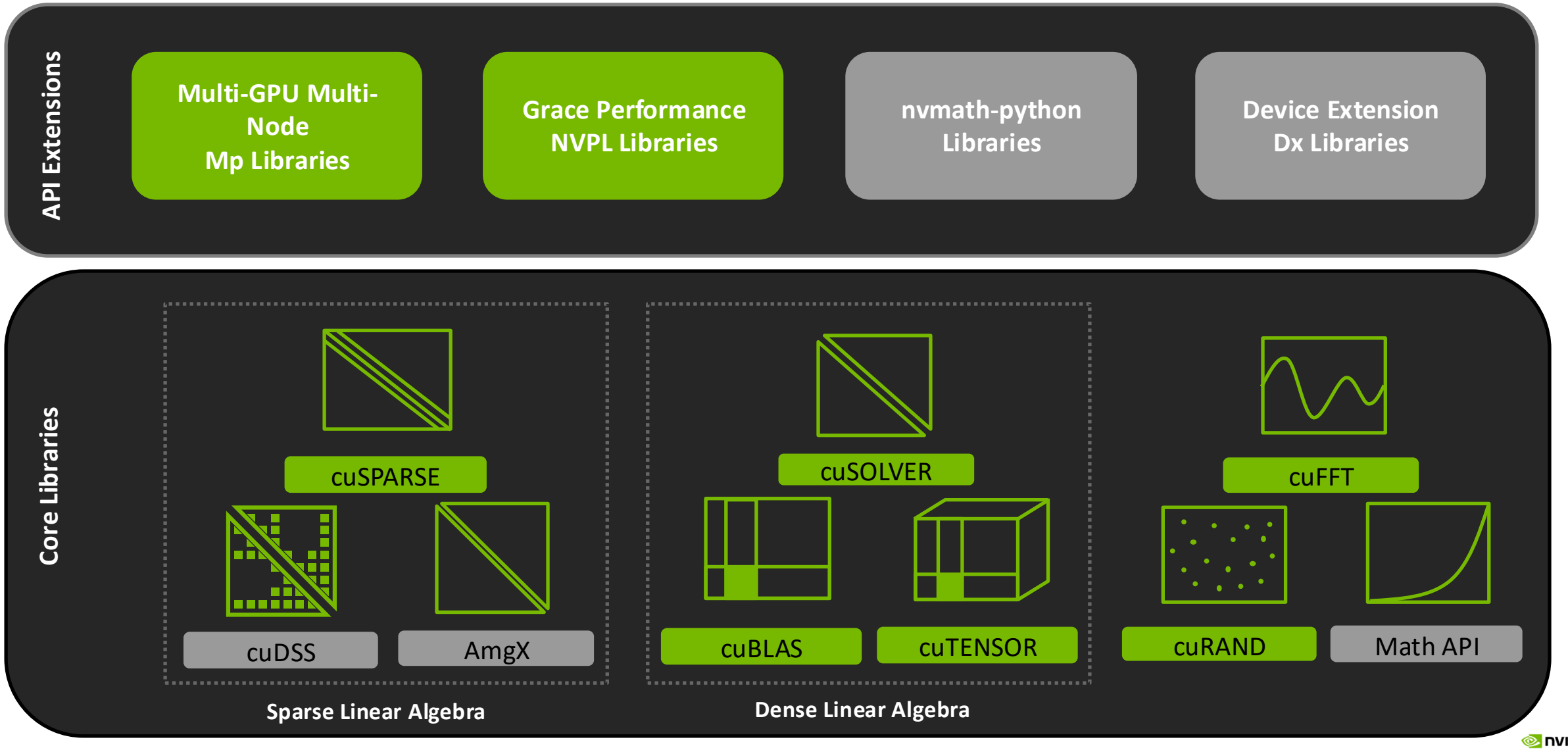
- HPC-X には、**MPI** および **OpenSHMEM** の通信フレームワークが含まれています。
  - MPI のポイントツーポイント通信、RMA（Remote Memory Access）、および集合通信は、すべて **UCX**、**UCC**、**SHARP** 上で動作します。これらは NVIDIA の高性能 RDMA 通信ミドルウェアであり、最新のネットワークオフロードやアクセラレータ機能を透過的に活用します。
  - CPU および GPU のネットワーキングを、追加設定なしでそのままサポートします。
- HPC-X 2.21 以降で **Grace Hopper** をサポート
- HPC-X は各 HPC SDK リリースにおいて、十分に**テストされ、統合されて**います。



Math ライブラリ

NVIDIA Core Math Libraries

さまざまな産業分野にわたるアプリケーションの実現

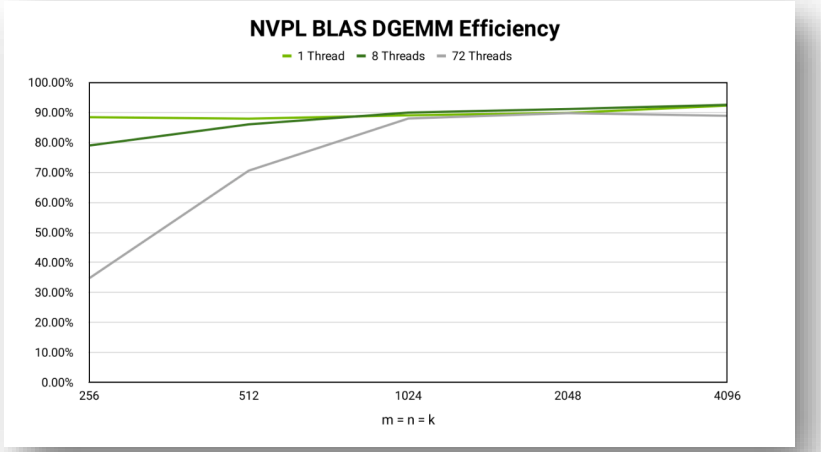


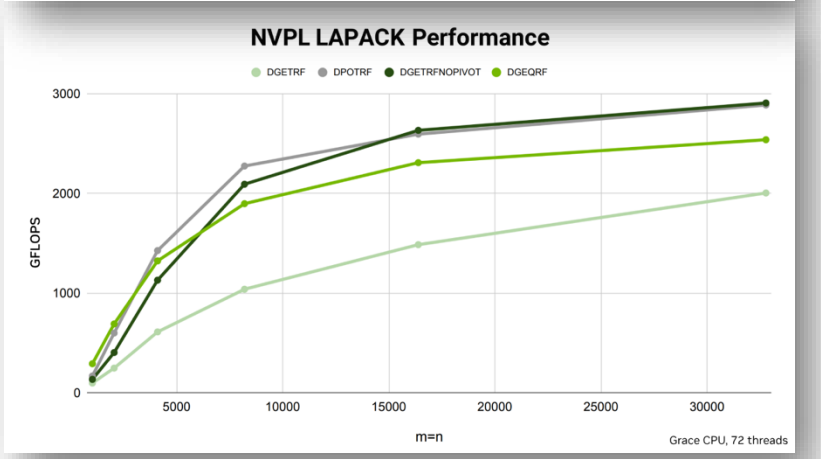
NVIDIA Performance Libraries

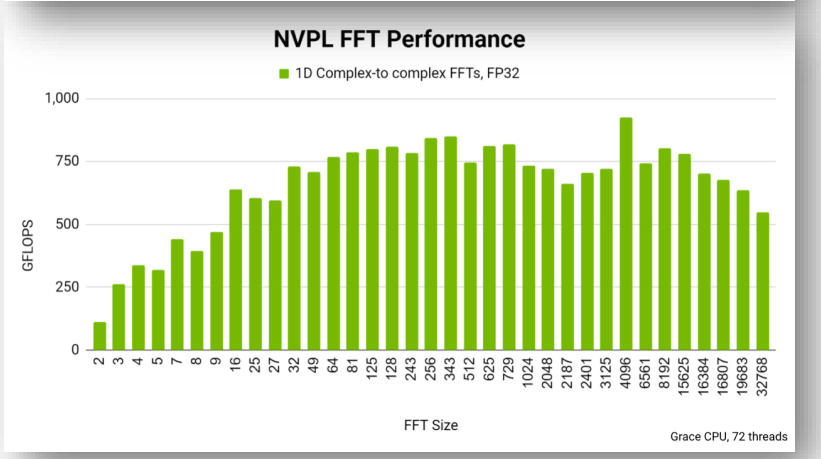
NVIDIA CPU 向けに最適化された数値計算ライブラリ

- アプリケーションを NVIDIA のデータセンター向け CPU に容易に移植可能
- 標準インターフェースを実装する数値計算ライブラリのドロップイン置き換えとして利用可能
- 高性能ライブラリ向けの新しいインターフェースを提供
- HPC SDK の一部として、またスタンドアロンでも提供

BLAS	LAPACK	PBLAS	SCALAPACK
TENSOR	SPARSE	RAND	FFT

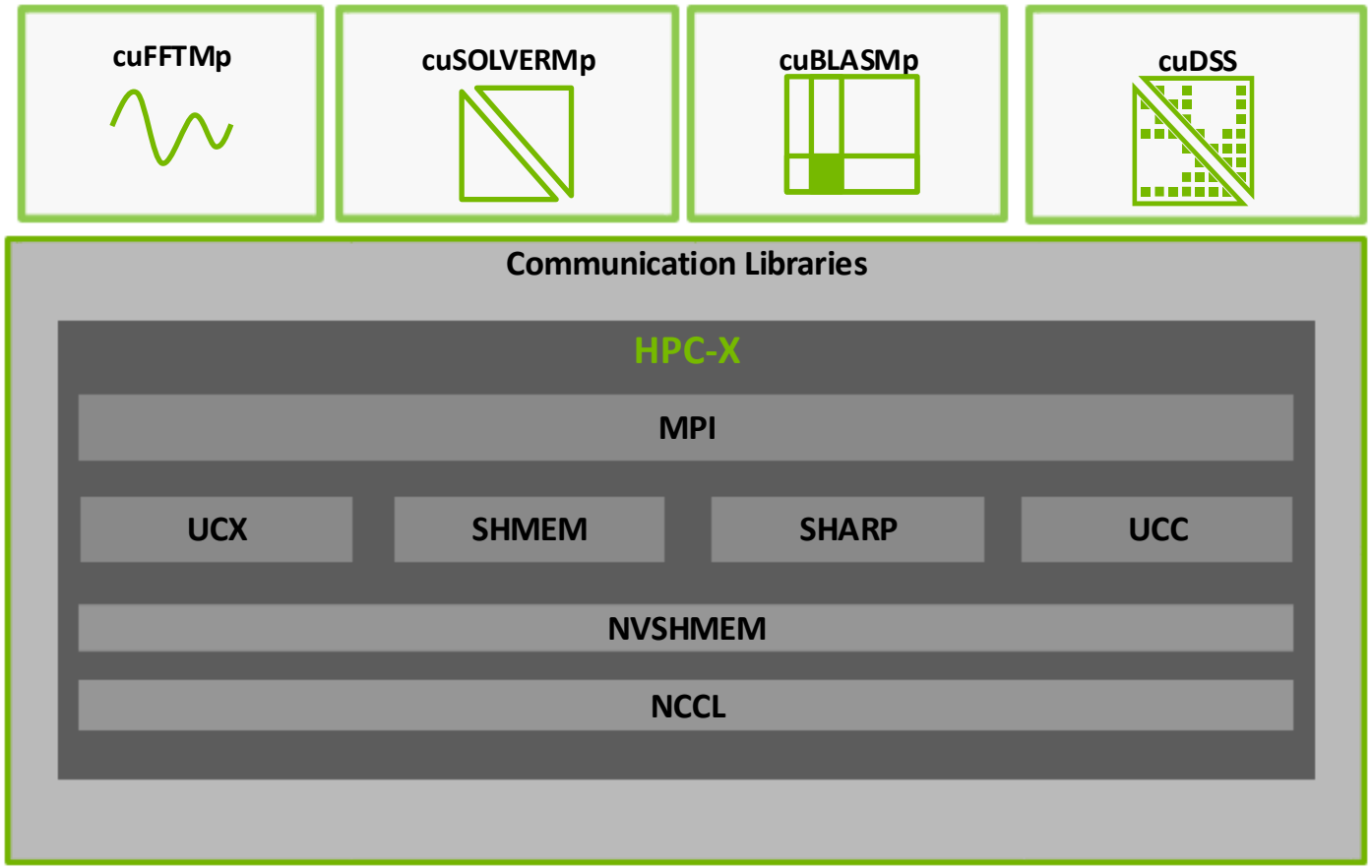






マルチGPU／マルチノード対応ライブラリ

高性能な分子動力学シミュレーションおよび出力解析



- **課題**：大規模な科学計算問題を解く際には、複雑な通信パターンがしばしば発生します。
- **解決策**：線形代数や FFT などの重要な数値計算処理に対して、NVIDIA は HPC-X 通信スタックを基盤とした、マルチノード・マルチ GPU 対応の最適化ライブラリを提供しています。

GROMACS

高性能な分子動力学シミュレーションおよび出力解析

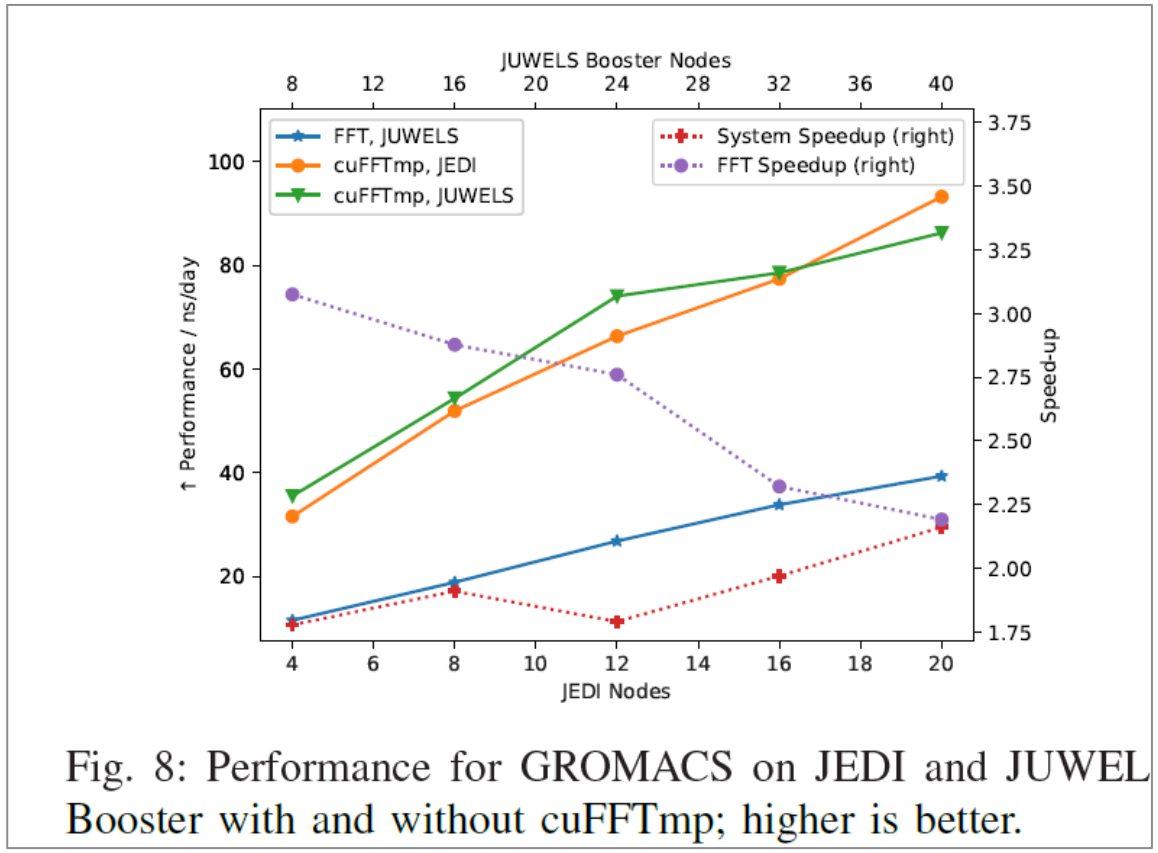


Fig. 8: Performance for GROMACS on JEDI and JUWEL Booster with and without cuFFTmp; higher is better.

GROMACS Scaling & Performance with cuFFTMp

Blog Post

- **GROMACS** は、高性能な分子動力学シミュレーションおよび出力解析のためのオープンソース・ソフトウェアスイートです。
- **課題**：PME（Particle Mesh Ewald）が単一 GPU に制限されていることによるスケーラビリティの制約。
  - PP（Particle-Particle）力計算に対して GPU を追加しても、数ノード以上にはスケールしない
- **解決策**: cuFFTMp

Vienna Ab initio Simulation Package (VASP)

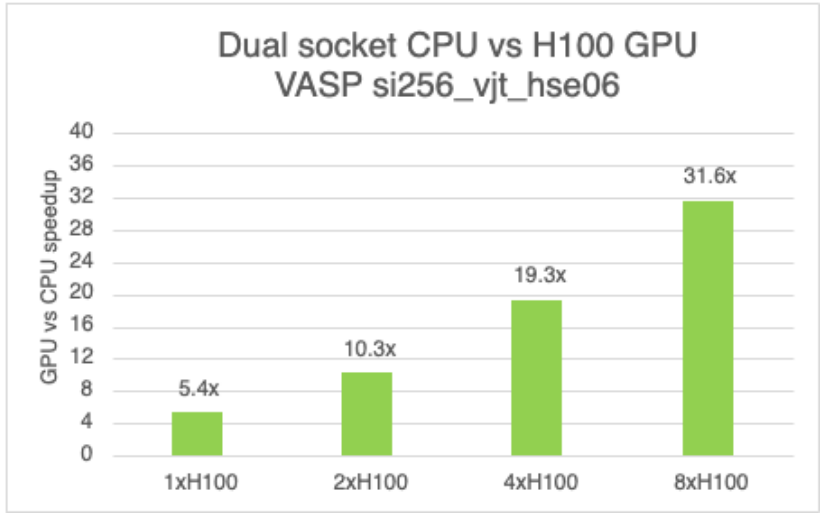
材料科学向けの OpenACC による効率的な実装

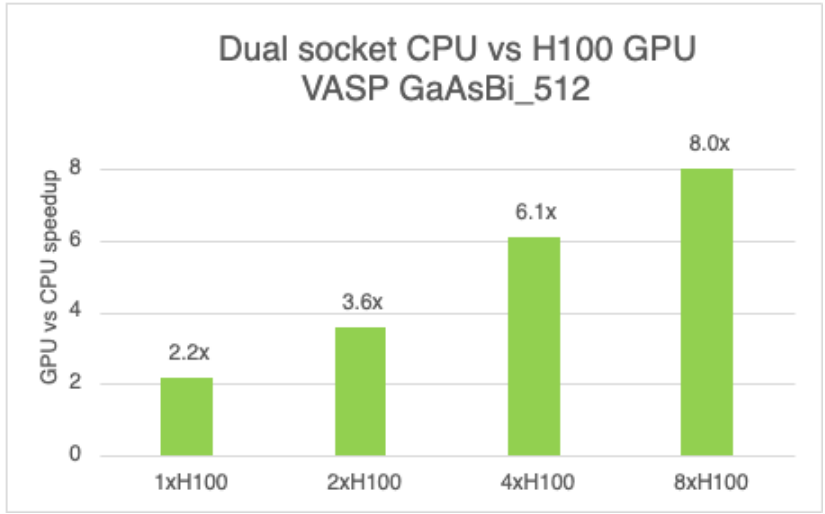
- **VASP** は、第一原理に基づく電子状態計算や量子力学的分子動力学など、原子スケールの材料モデリングを行うための計算プログラムです。

- **実装:**

OpenACC ディレクティブは、性能とスケーラビリティを最大化するために NVIDIA ライブラリと組み合わせて柔軟に活用されてきました。ベンチマークは以下を用いて実施しています：

- **NVIDIA HPC SDK**
- Mp Libraries: **cuSOLVERMp**, **cuBLASMp**



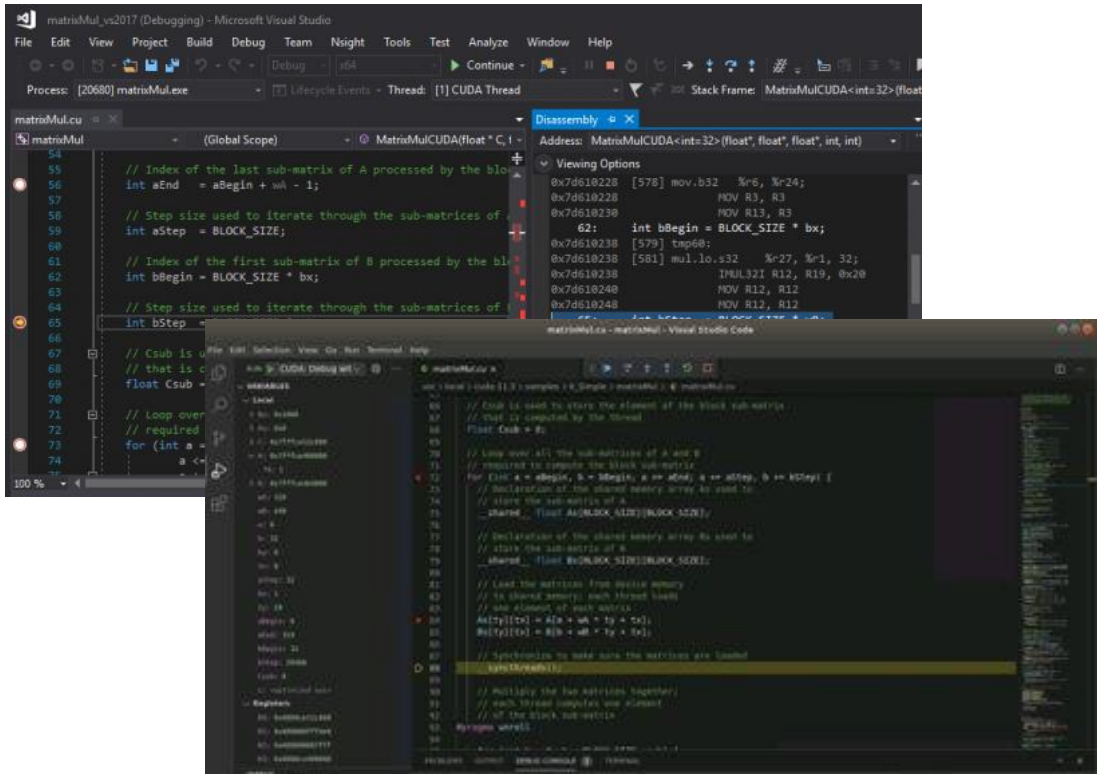


Developer ツール

開発者ツールのエコシステム

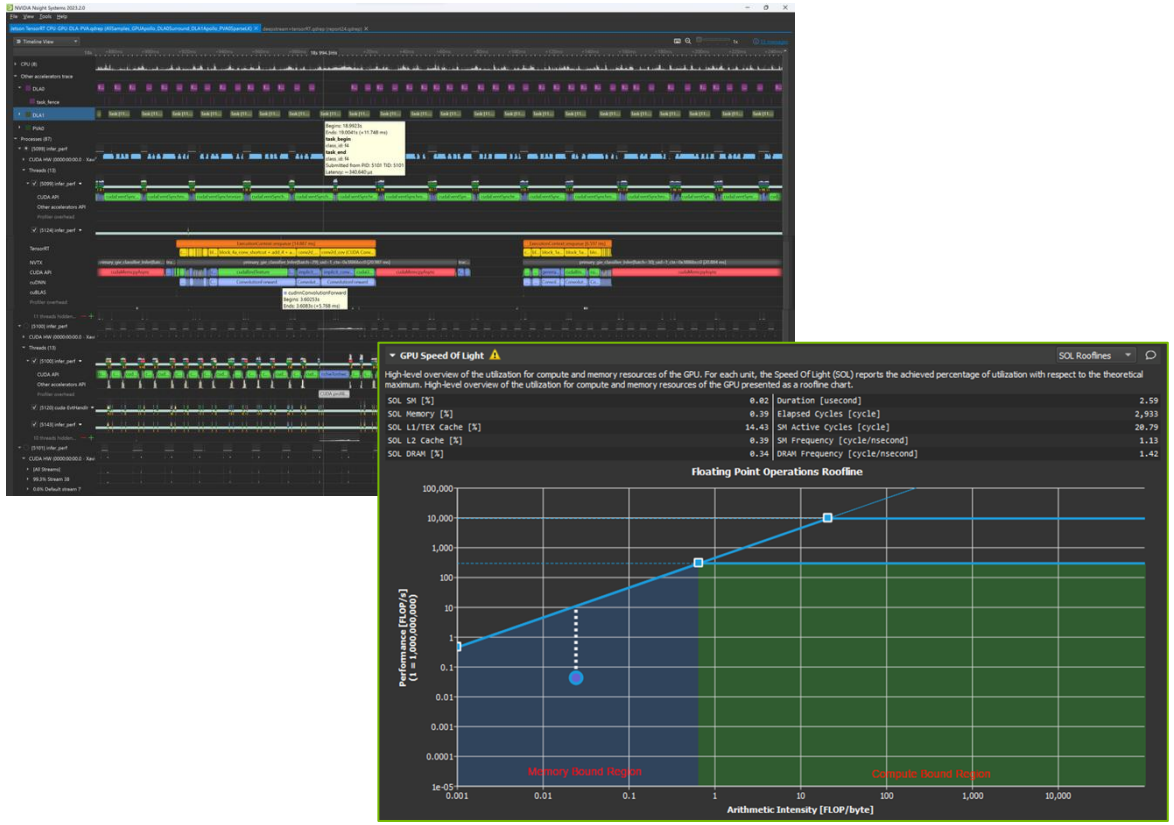
デバッガおよびIDEとの統合

cuda-gdb, Nsight Visual Studio Edition,
Nsight Visual Studio **Code** Edition, Nsight Eclipse Edition



プロファイラ

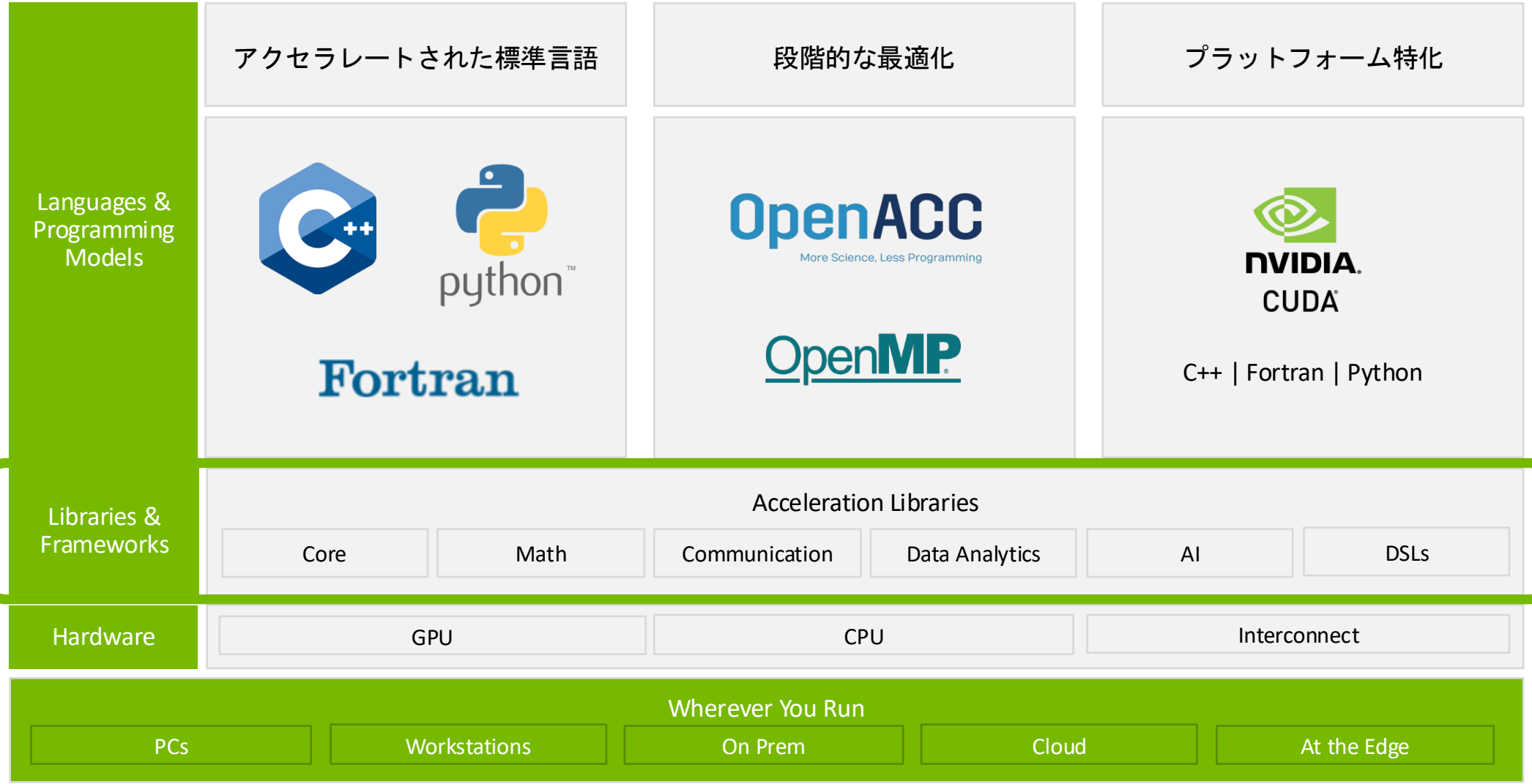
Nsight Systems, Nsight Compute, CUPTI, NVIDIA
Tools eXtension (NVTX)



GPU プログラミング

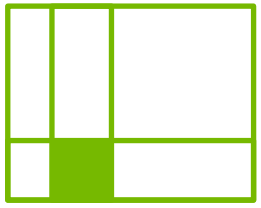
NVIDIAプラットフォームのプログラミング

比類なき開発者の柔軟性



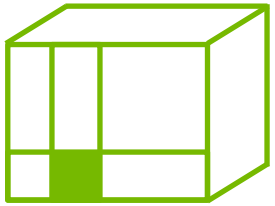
NVIDIA Math ライブラリ

Linear Algebra, FFT, RNG, and Basic Math

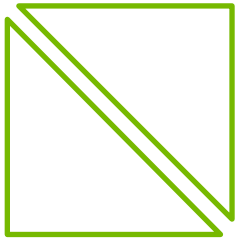


cuBLAS

cuSPARSE

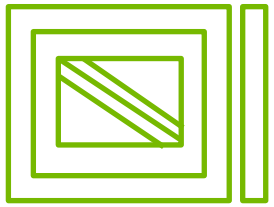


cuTENSOR

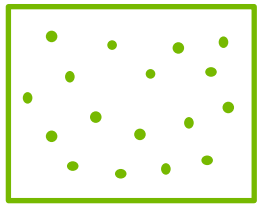


cuSOLVER

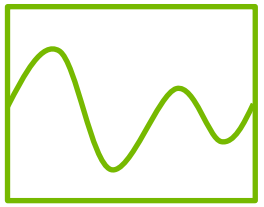
CUTLASS



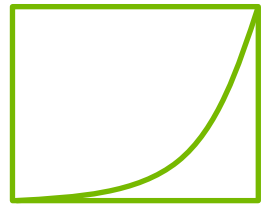
AMGX



cuRAND



cuFFT



Math API

パートナーライブラリ


OpenCV


FFmpeg


ARRAYFIRE


MAGMA
ICL UT


IMSL
RogueWave
SOFTWARE


SuiteSparse
CHOLMOD


CULA | tools


[HiPLAR]


gpp
Geometry Performance Primitives


PARALUTION


Sundog™
Software

NVIDIAプラットフォームのプログラミング

比類なき開発者の柔軟性

	アクセラレートされた標準言語	段階的な最適化	プラットフォーム特化
Languages & Programming Models	C++ / python / Fortran	OpenACC (More Science, Less Programming) / OpenMP	NVIDIA CUDA C++ | Fortran | Python
Libraries & Frameworks	Acceleration Libraries: Core / Math / Communication / Data Analytics / AI / DSLs		
Hardware	GPU	CPU	Interconnect
Wherever You Run	PCs / Workstations / On Prem / Cloud / At the Edge		

SAXPY (Y = A*X + Y)

Standard Language (C++)

```
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);
...
```

CPU

```
void saxpy(int n, float a,
           float *x, float *y)
{
  auto i = std::views::iota(0, n);

  std::for_each(std::execution::par,
                i.begin(),i.end(), [=](auto i) {
      y[i] += a*x[i];
  });
}



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

Standard Language

SAXPY

標準言語並列(C++)

- NVC++は、並列実行ポリシーを指定した標準 C++アルゴリズムをコンパイルできます。
- NVC++のコマンドラインオプション「-stdpar」を指定することで、GPU によって高速化された C++並列アルゴリズムを有効にできます。
- ホストメモリと GPU デバイスメモリ間のすべてのデータ転送は、CUDA Managed Memory の制御下で暗黙的かつ自動的に行われます。

```
void saxpy(int n, float a, float *x, float *y)
{
   auto i = std::views::iota(0, n);

   std::for_each(std::execution::par, i.begin(), i.end(),
      [=](auto i) {
      y[i] += a*x[i];
   });
}

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

SAXPY (Y = A*X + Y)

Standard Language (Fortran)

```
subroutine saxpy(n, a, x, y)
  real :: a, x(:), y(:)
  integer :: n, i

  do i = 1, n
    y(i) = a*x(i)+y(i)
  enddo

end subroutine saxpy

...

call saxpy(N, 3.0, x, y)

...
```

CPU

```
subroutine saxpy(n, a, x, y)
  real :: a, x(:), y(:)
  integer :: n, i

  do concurrent (i = 1 : n)
    y(i) = a*x(i)+y(i)
  enddo

end subroutine saxpy

...

call saxpy(N, 3.0, x, y)

...
```

Standard Language

SAXPY

標準言語並列 (Fortran)

- NVFORTRAN は、Fortran の do concurrent 構文をコンパイルできます。
- NVFORTRAN のコマンドラインオプション「-stdpar」を指定することで、do concurrent 構文を GPU によって高速化できます。
- ホストメモリと GPU デバイスメモリ間のすべてのデータ転送は、CUDA Managed Memory の制御下で、暗黙的かつ自動的に行われます。

```
subroutine saxpy(n, a, x, y)
  real :: a, x(:), y(:)
  integer :: n, i

  do concurrent (i = 1 : n)
    y(i) = a*x(i)+y(i)
  enddo

end subroutine saxpy

...

call saxpy(N, 3.0, x, y)

...
```

Standard Language Parallelism コードのビルド

NVIDIA HPC SDK

- C++: `nvc++`, Fortran: `nvfortran`
- `-stdpar=gpu` : Standard language parallelism を有効にし、NVIDIA GPU 向けにビルド
  - `-stdpar=multicore` : マルチコア CPU 向けにもビルド可能
- `-Minfo=stdpar` : 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
- `-std=c++20` : `std::views::iota()` を使うために必要

```
$ nvc++ -stdpar=gpu -Minfo=stdpar –std=c++20 saxpy.cpp
```

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

NVIDIAプラットフォームのプログラミング

比類なき開発者の柔軟性

	アクセラレートされた標準言語	段階的な最適化	プラットフォーム特化
Languages & Programming Models	C++ python Fortran	OpenACC More Science, Less Programming OpenMP	NVIDIA CUDA C++ | Fortran | Python
Libraries & Frameworks	Acceleration Libraries: Core, Math, Communication, Data Analytics, AI, DSLs		
Hardware	GPU, CPU, Interconnect		
Wherever You Run	PCs, Workstations, On Prem, Cloud, At the Edge		

SAXPY (Y = A*X + Y)

OpenACC

```
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);
...
```

CPU (OpenMP)

```
void saxpy(int n,
           float a,
           float *x,
           float *restrict y)
{
#pragma acc parallel loop 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

SAXPY

OpenACC

- OpenMP (CPU) コードと非常に似た書き方で
  - #pragma omp … -> #pragma acc …
  - データ転送に関する補足

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

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

SAXPY (Y = A*X + Y)

OpenMP Offloading

```
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);
...
```

CPU (OpenMP)

```
void saxpy(int n,
           float a,
           float *x,
           float *restrict y)
{
#pragma omp target teams loop map(tofrom:y[:n])
map(to:x[:n])
  for (int i = 0; i < n; ++i)
    y[i] += a*x[i];
}

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

OpenMP Offloading

SAXPY

OpenACC (Fortran)

- OpenMP (CPU) コードと非常に似た書き方で
  - !$omp … -> !$acc …
  - データ転送に関する補足

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

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

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

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` : 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=mem:managed` : CUDA Managed Memory 有効化

```
$ nvc -acc=gpu –gpu=mem:managed -Minfo=accel saxpy.c
```

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

NVIDIAプラットフォームのプログラミング

比類なき開発者の柔軟性

	アクセラレートされた標準言語	段階的な最適化	プラットフォーム特化
Languages & Programming Models	C++ python Fortran	OpenACC More Science, Less Programming OpenMP	NVIDIA CUDA C++ | Fortran | Python
Libraries & Frameworks	Acceleration Libraries: Core, Math, Communication, Data Analytics, AI, DSLs		
Hardware	GPU, CPU, Interconnect		
Wherever You Run	PCs, Workstations, On Prem, Cloud, At the Edge		

SAXPY (Y = A*X + Y)

CUDA

```
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);
...
```

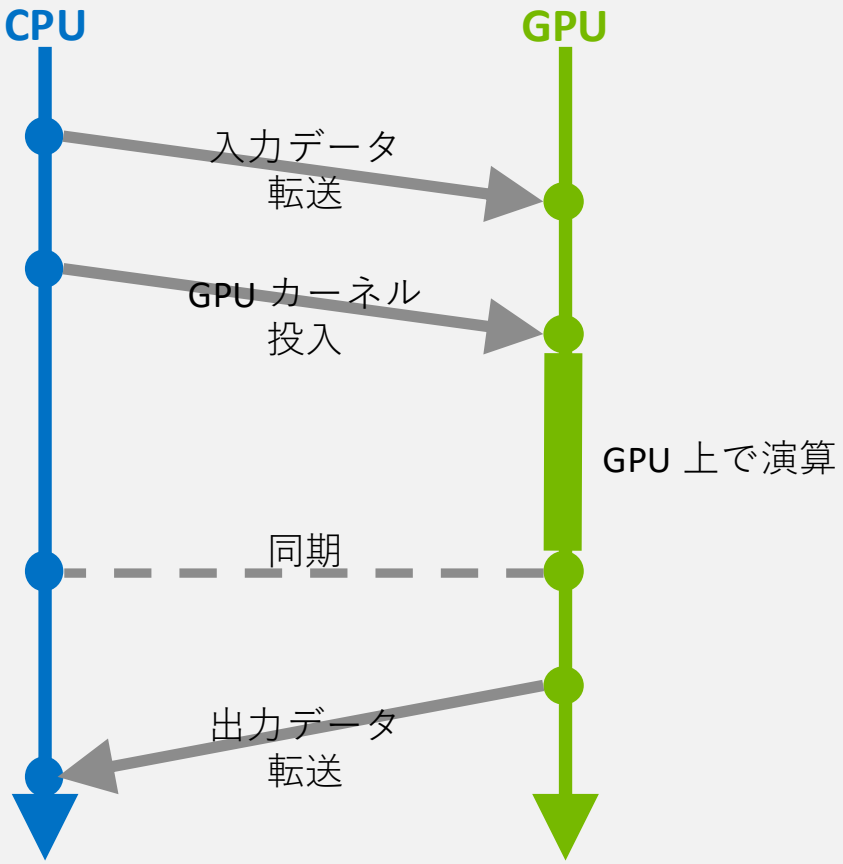
CPU

```
__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 = N * sizeof(float);
x = (float *) malloc(size);
y = (float *) malloc(size);
cudaMalloc(&d_x, size);
cudaMalloc(&d_y, size);
...

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);
...
```

CUDA

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)



SAXPY

CUDA

- **GPU メモリ確保**
- 入力データ転送
- カーネル起動
- 同期
- 出力データ転送

```
__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 = N * sizeof(float);
x = (float *) malloc(size);
y = (float *) malloc(size);
cudaMalloc(&d_x, size);
cudaMalloc(&d_y, size);
...

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);
```

SAXPY

CUDA

- GPU メモリ確保
- **入力データ転送**
- カーネル起動
- 同期
- 出力データ転送

```
__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 = N * sizeof(float);
x = (float *) malloc(size);
y = (float *) malloc(size);
cudaMalloc(&d_x, size);
cudaMalloc(&d_y, size);
...

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);
```

SAXPY

CUDA

- GPU メモリ確保
- 入力データ転送
- **カーネル起動**
- 同期
- 出力データ転送

```
__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 = N * sizeof(float);
x = (float *) malloc(size);
y = (float *) malloc(size);
cudaMalloc(&d_x, size);
cudaMalloc(&d_y, size);
...

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);
```

SAXPY

CUDA

- GPU メモリ確保
- 入力データ転送
- カーネル起動
- **同期**
- 出力データ転送

```
__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 = N * sizeof(float);
x = (float *) malloc(size);
y = (float *) malloc(size);
cudaMalloc(&d_x, size);
cudaMalloc(&d_y, size);
...

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);
```

SAXPY

CUDA

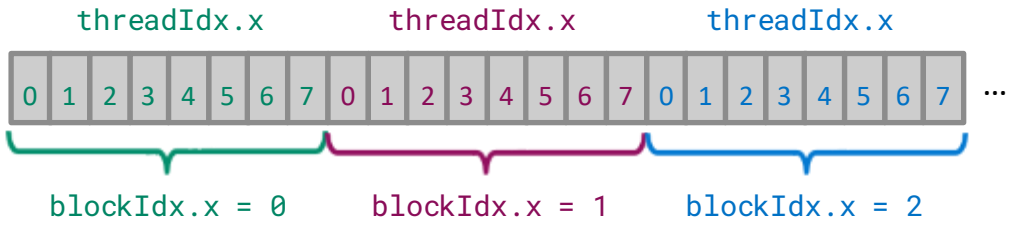
- GPU メモリ確保
- 入力データ転送
- カーネル起動
- 同期
- **出力データ転送**

```
__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 = N * sizeof(float);
x = (float *) malloc(size);
y = (float *) malloc(size);
cudaMalloc(&d_x, size);
cudaMalloc(&d_y, size);
...

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 カーネル

- 1 つの GPU スレッドの処理内容を記述
- 1 つの GPU スレッドが、1 つの配列要素の処理を担当
  - threadIdx.x ：Thread ID within the block
  - blockDim.x ：Dimensions of the block
  - blockIdx.x ：Block index within the grid

blockDim.x = 8 (8 threads/block) の例




```
__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 = N * sizeof(float);
x = (float *) malloc(size);
y = (float *) malloc(size);
cudaMalloc(&d_x, size);
cudaMalloc(&d_y, size);
...

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);
```

Global thread ID


NVIDIA