Introduction To GPU Programming

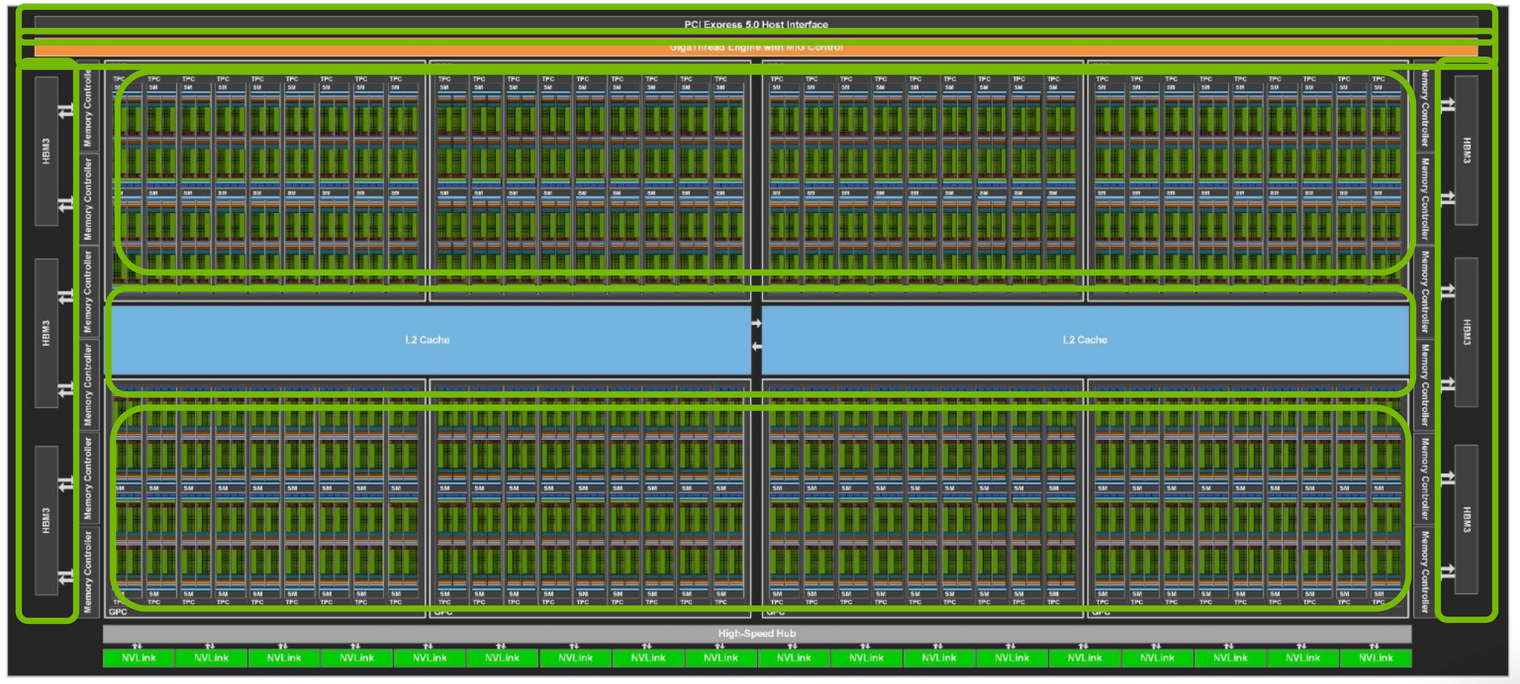
Agenda

- NVIDIA H100
- Overview of GPU Programming
- Standard Language Parallelism
- OpenACC
- CUDA
- NVIDIA Developer Tools

NVIDIA H100

GPU の構造

NVIDIA H100



- 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 (HBM3, 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 (50 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、H100 : 132

SM (Streaming Multi-processor)

- 演算ユニット
  - INT32: 64 個
  - FP32: 128 個
  - FP64: 64 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: 256 KB

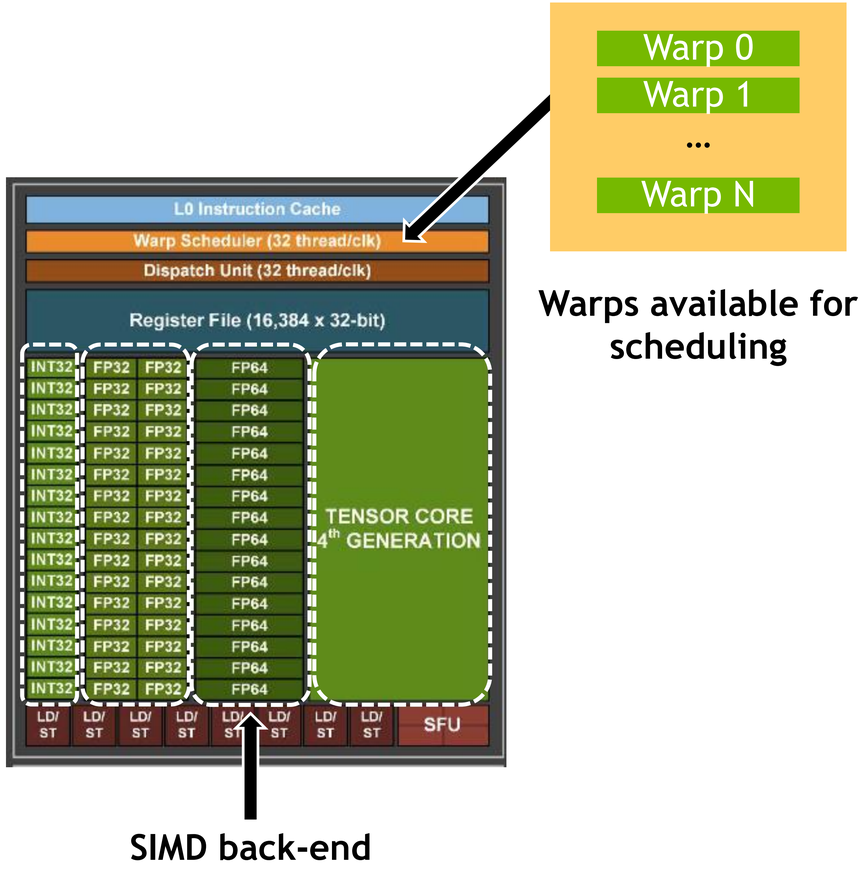
- Tensor Memory Accelerator

NVIDIA.

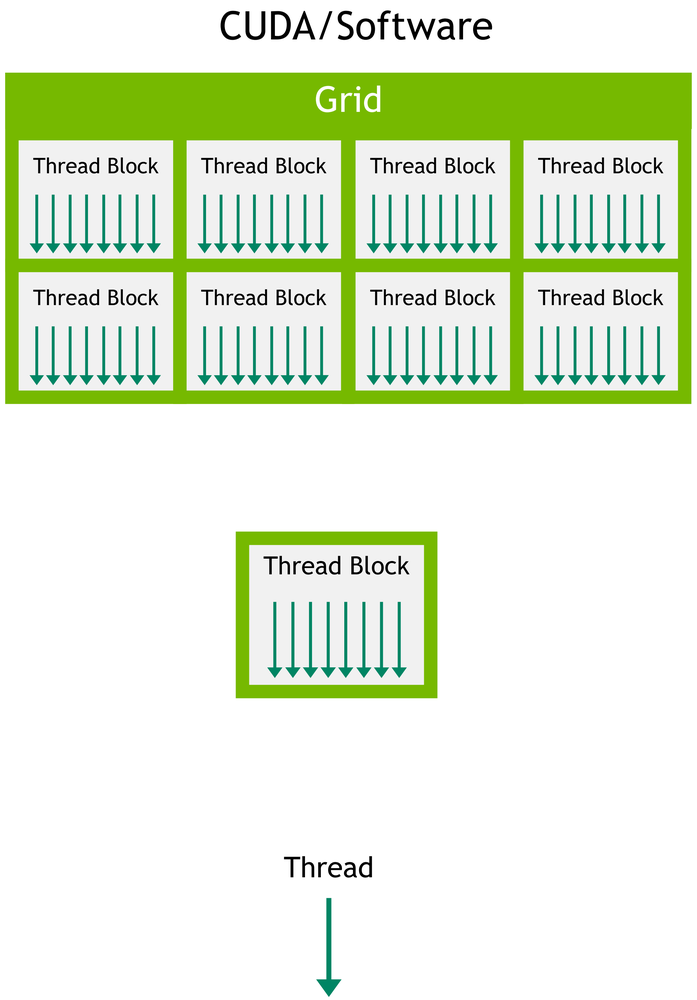
SIMT Architecture

Single-Instruction, Multiple-Thread

- Akin to a single-instruction multiple-data (SIMD) array processor per Flynn's taxonomy combined with fine-grained multithreading.
- SIMT architectures expose a large set of hardware threads, which is partitioned into groups called warps.
  - Interleave warp execution to hide latencies.
  - Execution context for earch warp is kept on-chip for fast interleaving.
- When scheduled, each thread of a warp executes on a given lane of a SIMD function unit.
- Each SM sub-partition can be thought of as a SIMT engine that creates, manages, schedules, and executes warps of 32 parallel threads.



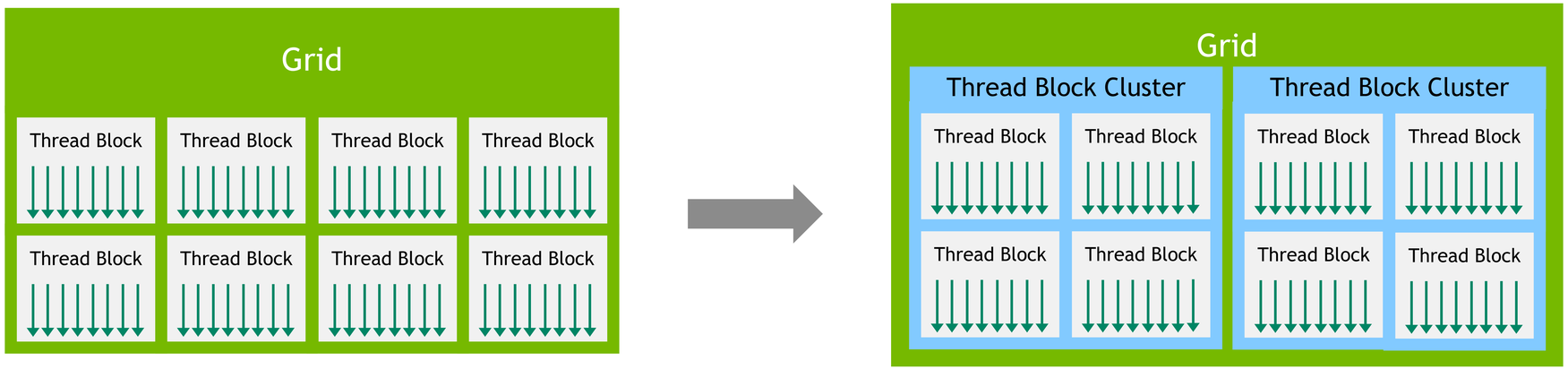
Thread Hierarchy



- A CUDA kernel is launched on a grid of thread blocks, which are completely independent.
- Thread blocsk are executed on SMs.
  - Several concurrent thread blocks can reside on an SM.
  - Thread blocks do not migrate.
  - Each block can be scheduled on any of the available SMs, in any order, concurrently, or in series.
- Individual threads execute on scalar CUDA cores.

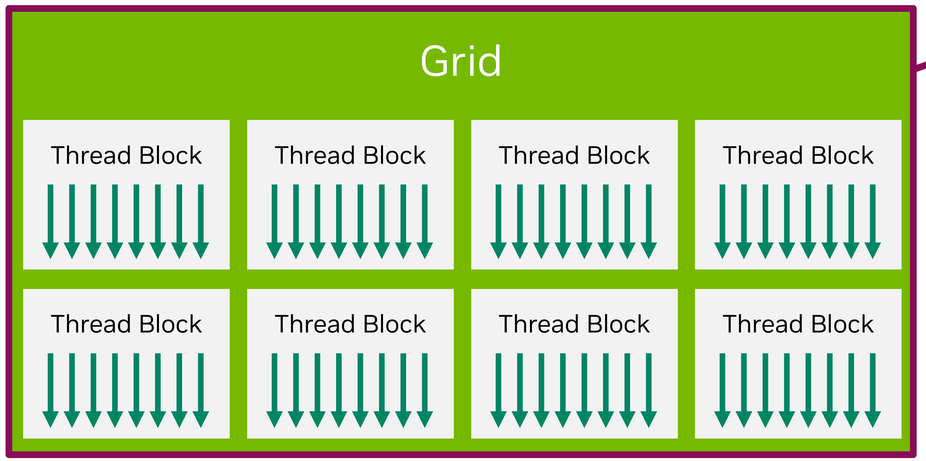
Thread Block Clusters

- For Hopper GPUs, CUDA introduced an optional level in the thread hierarchy called **Thread Block Clusters**.
- Thread blocks in a cluster are guaranteed to be concurrently scheduled and enable efficient cooperation and data sharing for threads across multiple SMs.
- For more information on this topic visit GTC session **[S62192]: "Advanced Performance Optimization in CUDA"**.



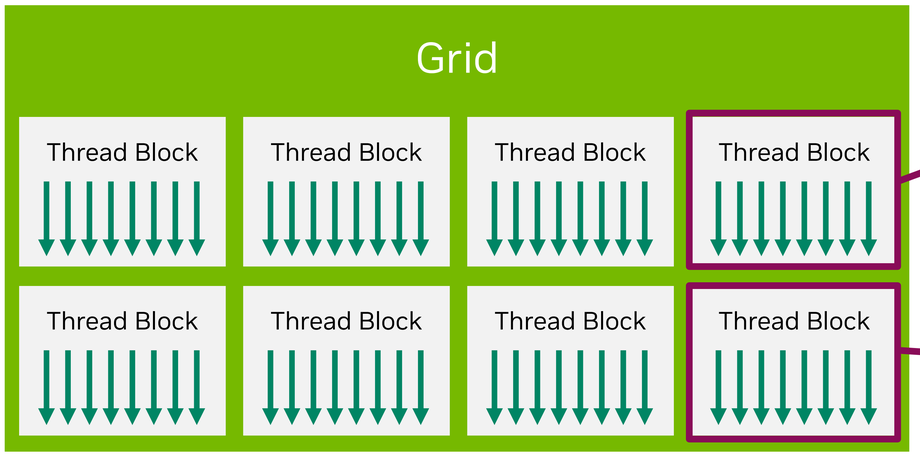
GPU カーネル実行の流れ

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



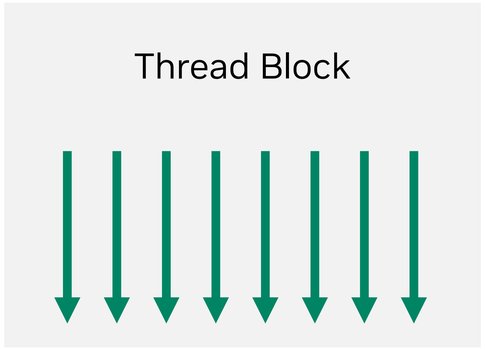
GPU カーネル実行の流れ

- 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、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



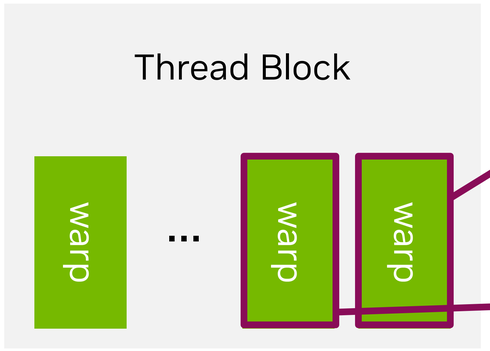
GPU カーネル実行の流れ

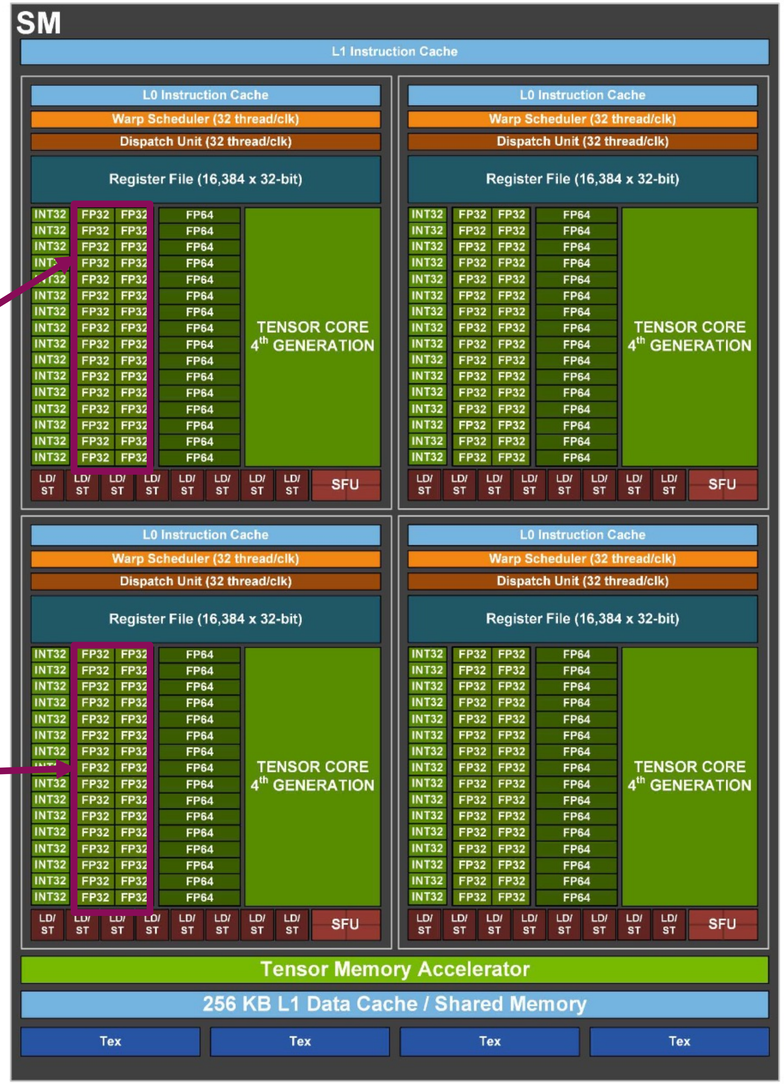
- SM 内のスケジューラが**ワープ**を CUDA コアに投入
  - ワープ: 32 スレッドのかたまり
  - ブロックをワープに分割、実行可能なワープを空 CUDA コアに割り当てる
- ワープ内に 32 スレッドは同期して同じ命令を実行
- 各ワープは互いに独立して実行
  - 同じブロック内のワープは、明示的に同期することも可能



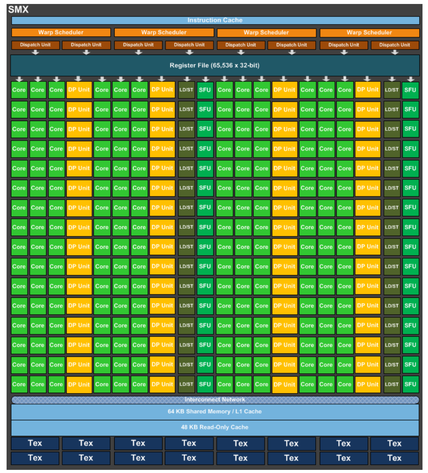
GPU カーネル実行の流れ

- SM 内のスケジューラが**ワープ**を CUDA コアに投入
  - ワープ: 32 スレッドのかたまり
  - 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
- ワープ内に 32 スレッドは同期して同じ命令を実行
- 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、明示的に同期することも可能

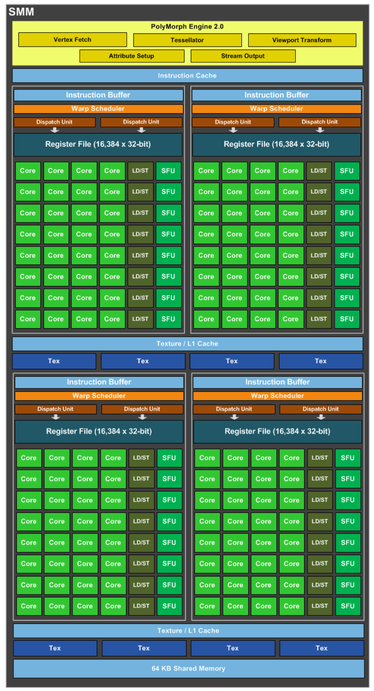




GPU アーキの変化を問題としないプログラミングモデル



Kepler, CC3.5
192 cores /SM



Maxwell, CC5.0
128 cores /SM

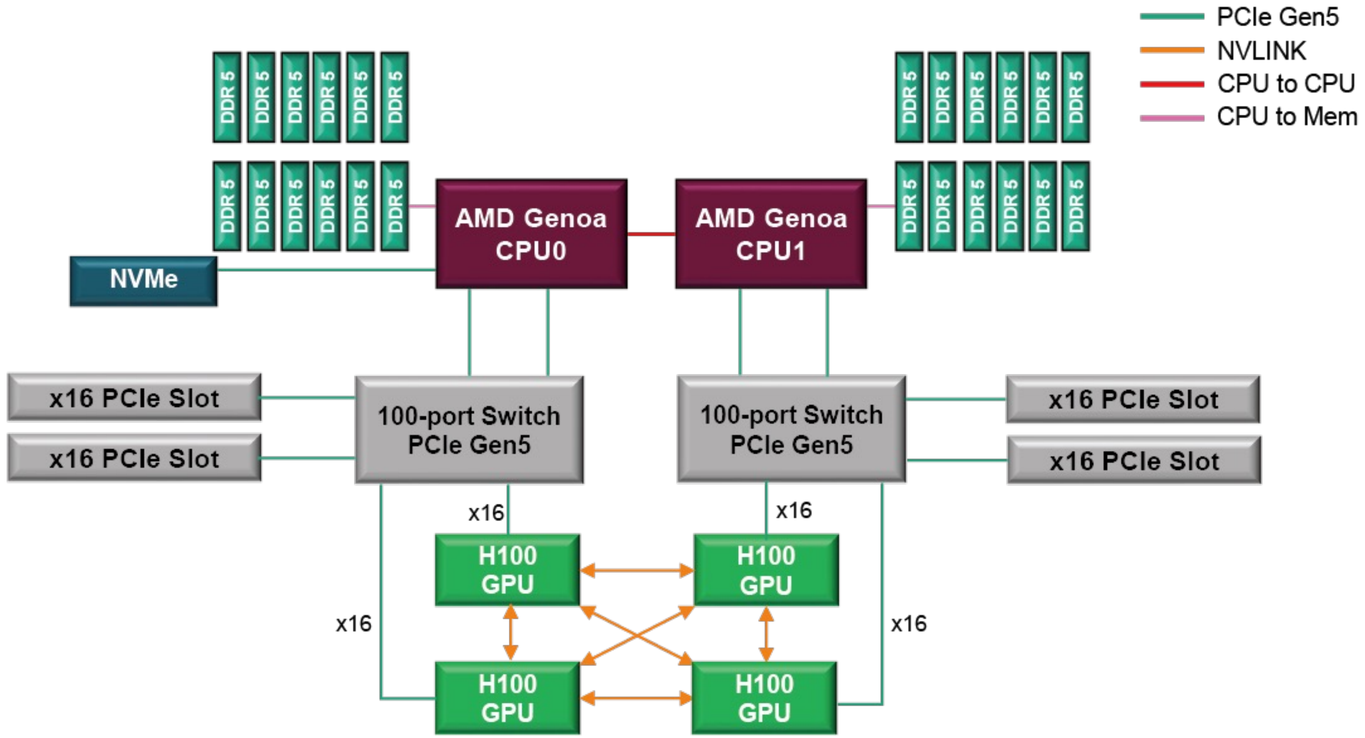
Pascal, CC6.0
64 cores /SM

Ampere, CC8.0
64 cores /SM

Hopper, CC9.0
128 cores /SM

TSUBAME4.0 Compute Node

- GPU : NVIDIA H100 SXM5 x 4
- GPU Memory : HBM2e 94GB / GPU
- Intra-node connection: Fully connected by NVLink
- Inter-node connection : InfiniBand NDR200 x 4



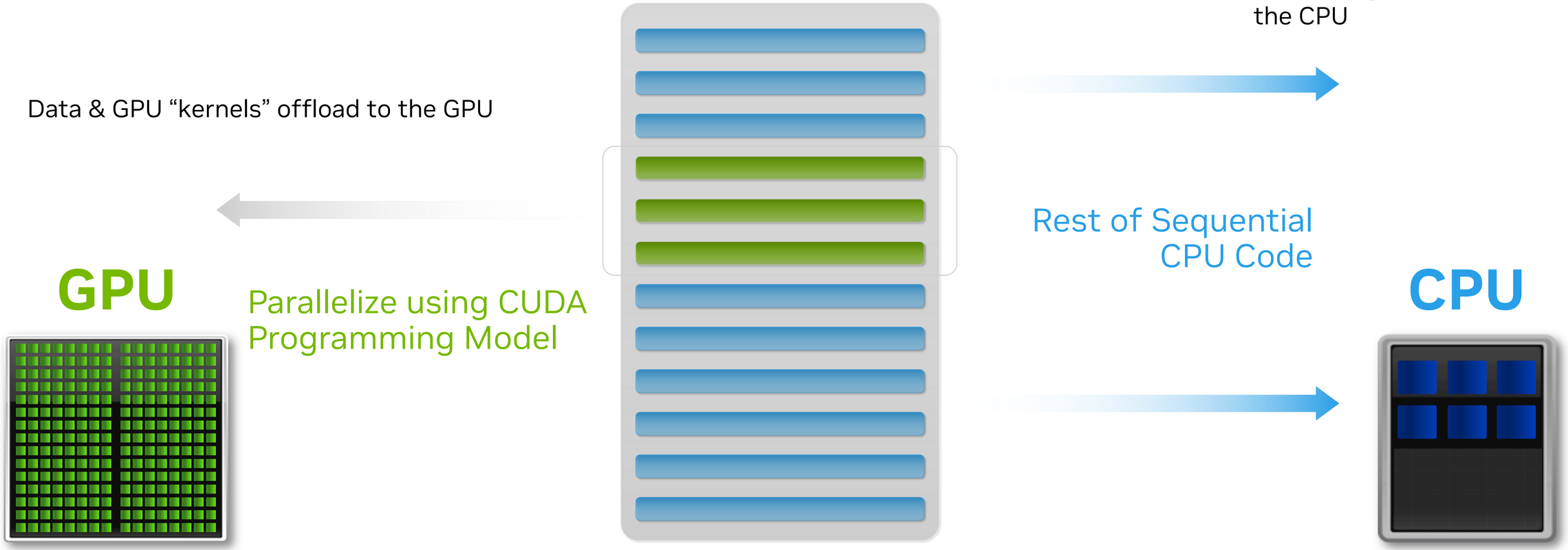
https://www.t4.gsic.titech.ac.jp/hardware

GPU Computing in a Nutshell

All GPU programming models follow this pattern

Program flow and resource allocation is managed by the CPU

Data & GPU “kernels” offload to the GPU



GPU

Parallelize using CUDA Programming Model

Rest of Sequential CPU Code

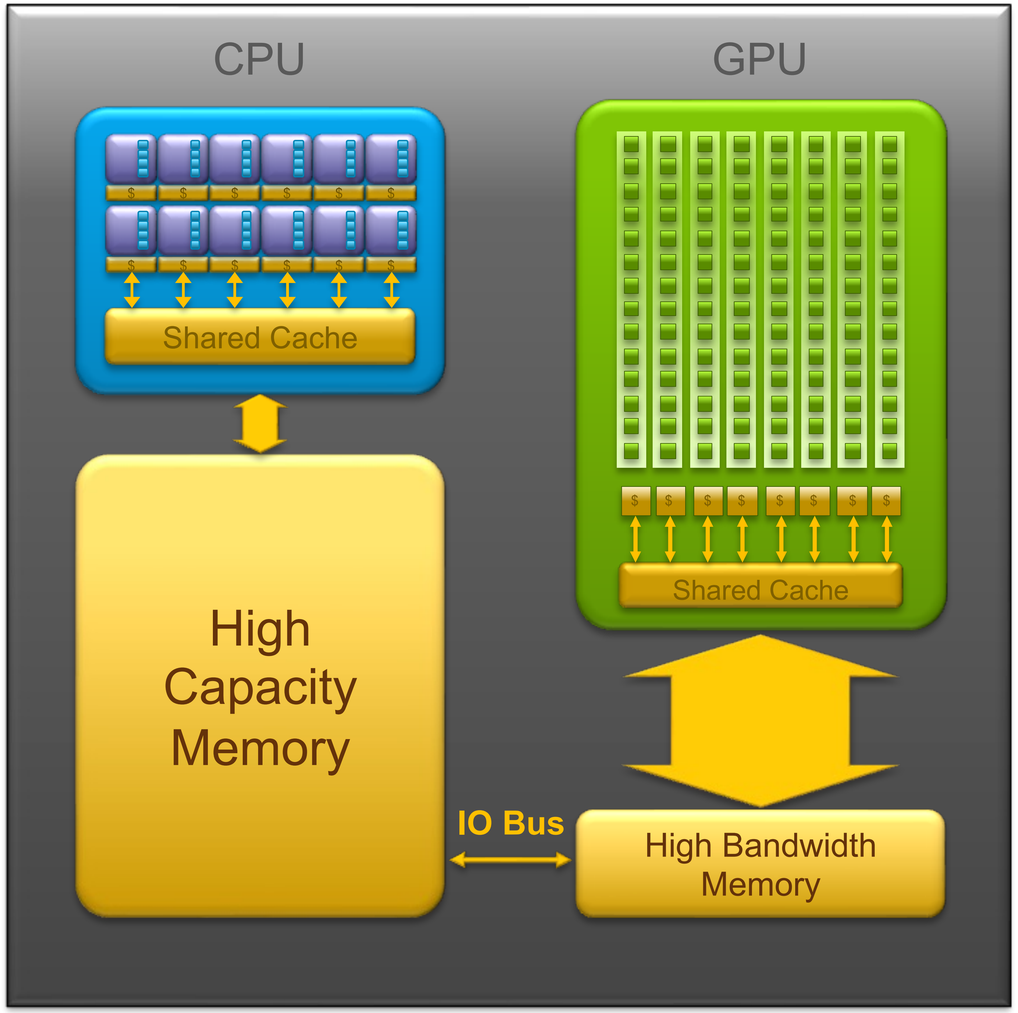
CPU

CPU & GPU work together

NVIDIA

CPU + GPU

- CPU memory is larger, GPU memory has more bandwidth
- CPU and GPU memory are usually separate, connected by an I/O bus (traditionally PCI-e)
- Any data transferred between the CPU and GPU will be handled by the I/O Bus
- The I/O Bus is relatively slow compared to memory bandwidth
- Basically, programmers have to handle explicit data transfer between CPU and GPU

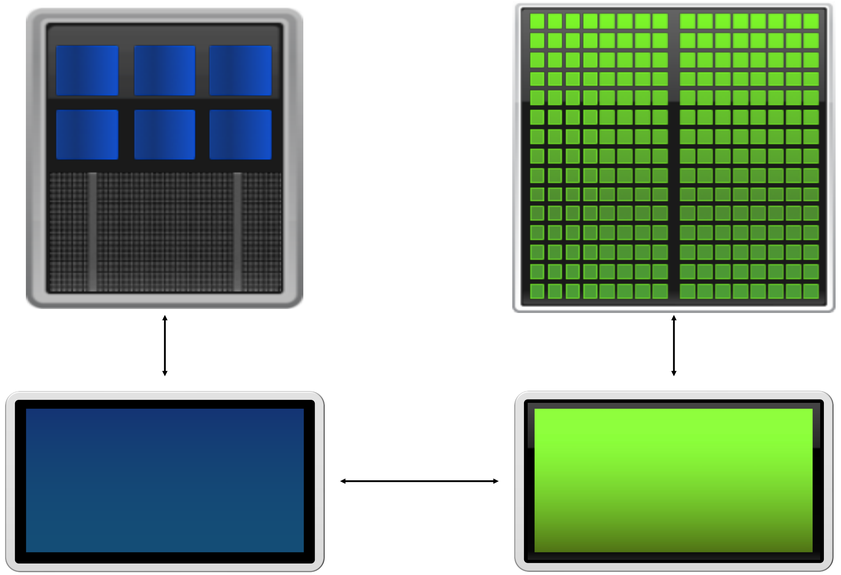


CUDA Unified Memory

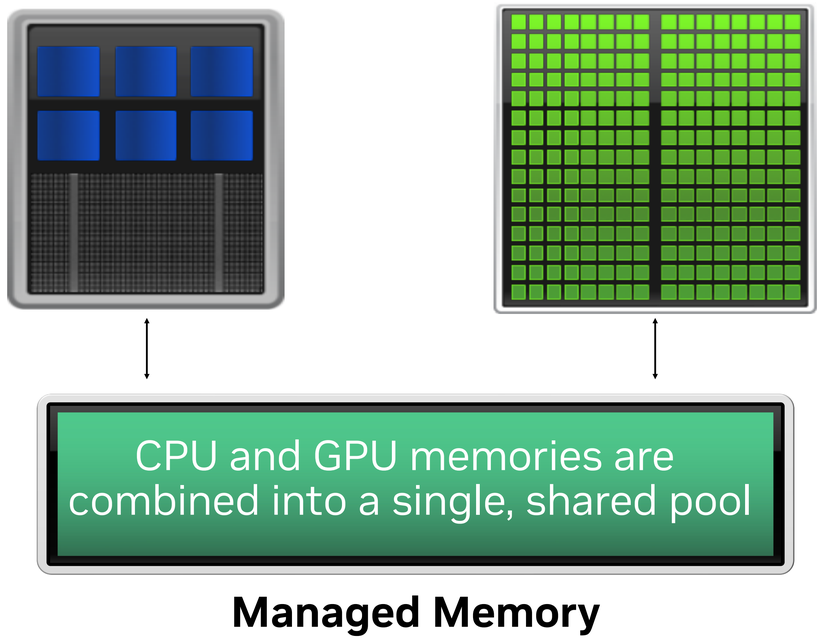
Simplifed Developer Effort

Commonly referred to as *“managed memory.”*

Without Managed Memory



With Managed Memory

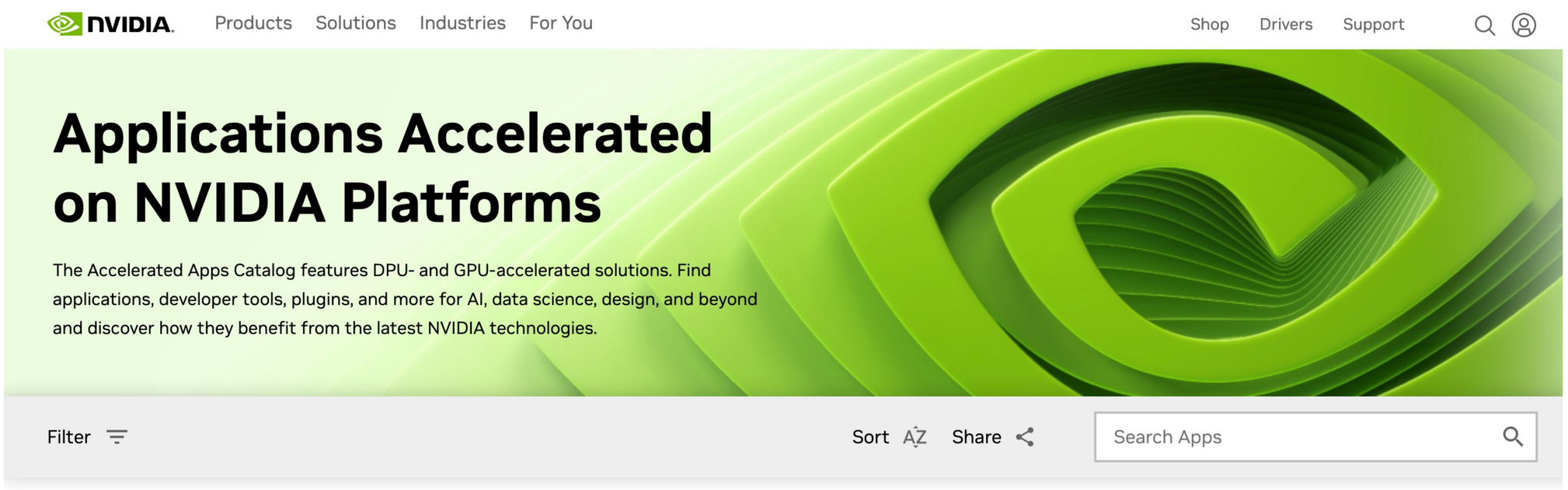


NVIDIA

Overview of GPU Programming

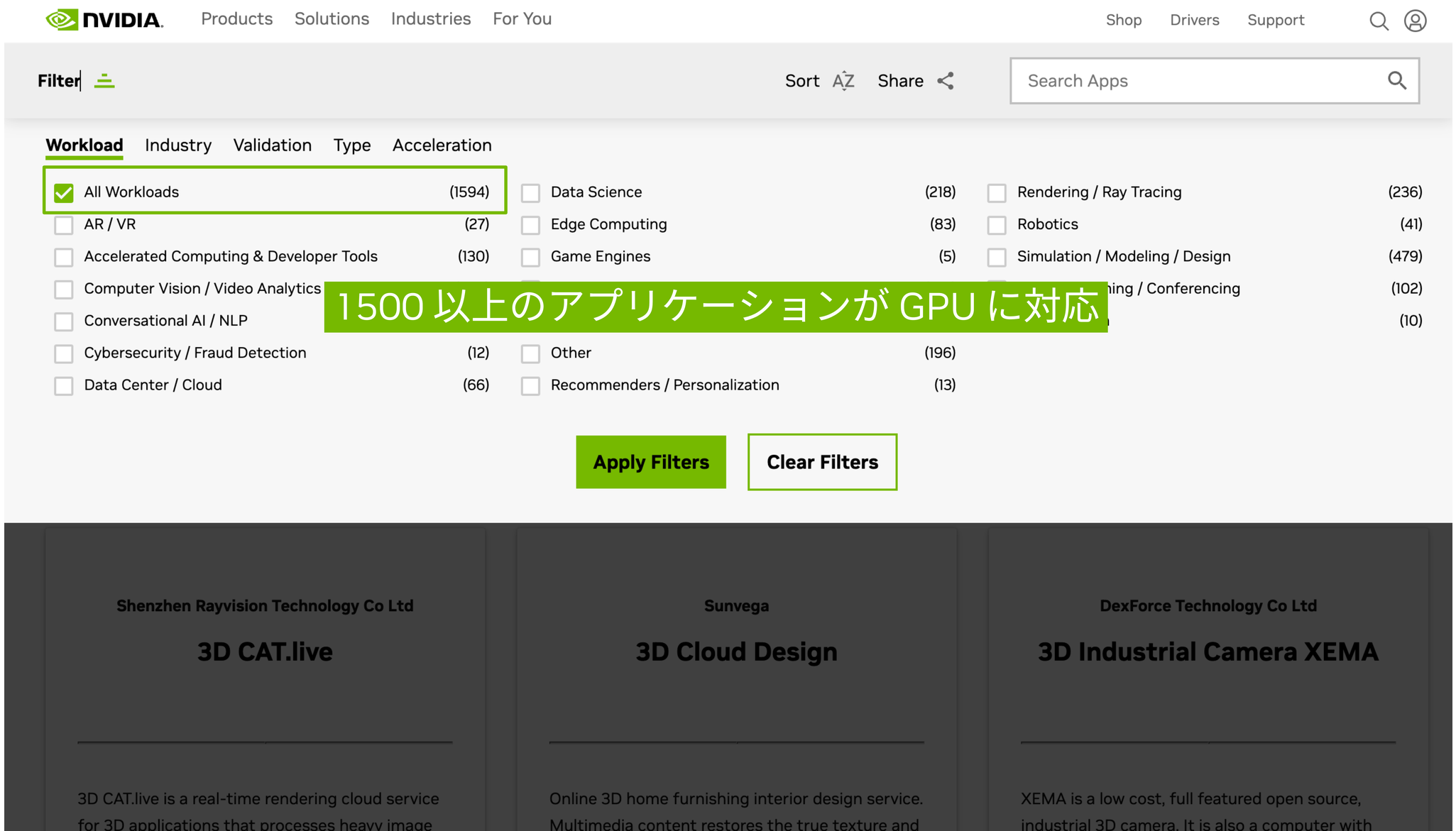
GPU Applications

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



GPU Applications

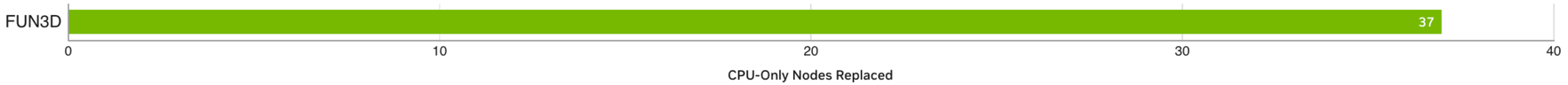
https://www.nvidia.com/en-us/gpu-accelerated-applications/



Leading Applications

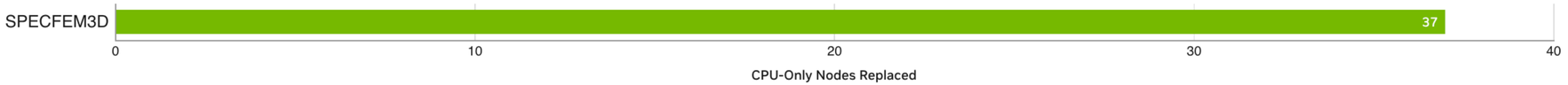
https://developer.nvidia.com/hpc-application-performance

Engineering



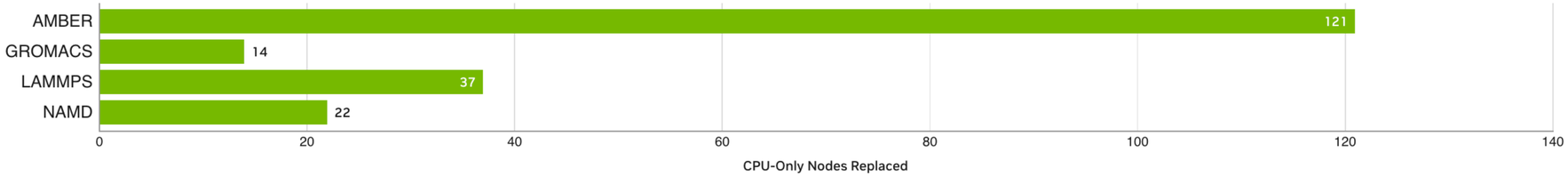
CPU Server: Dual Xeon Platinum 8480+ @2GHz | GPU Server: Dual Xeon Platinum 8480+ @2GHz with 4x NVIDIA H200 | FUN3D Benchmark: waverider-20M w/chemistry, CUDA Version: 12.4

Geoscience



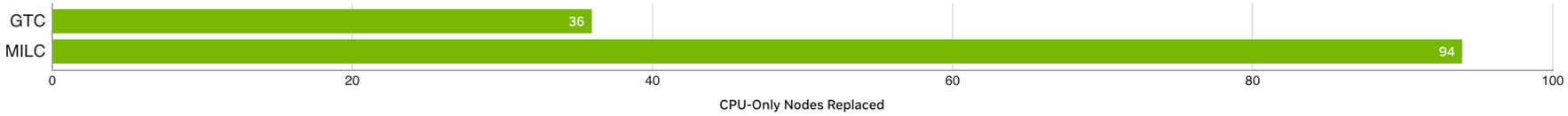
CPU Server: Dual Xeon Platinum 8480+ @2GHz | GPU Server: Dual Xeon Platinum 8480+ @2GHz with 4x NVIDIA H200 | SPECFEM3D Benchmark: four_material_simple_model, CUDA Version: 12.4

Molecular Dynamics



CPU Server: Dual Xeon Platinum 8480+ @2GHz | GPU Server: Dual Xeon Platinum 8480+ @2GHz with 4x NVIDIA H200 | AMBER Benchmark: DC-STMV_NPT, CUDA Version: 12.4 | GROMACS Benchmark: ADH Dodec, CUDA Version: 12.4 | LAMMPS Benchmark: Tersoff, CUDA Version: 12.4 | NAMD Benchmark: apoa1_nve_cuda, CUDA Version: 12.4

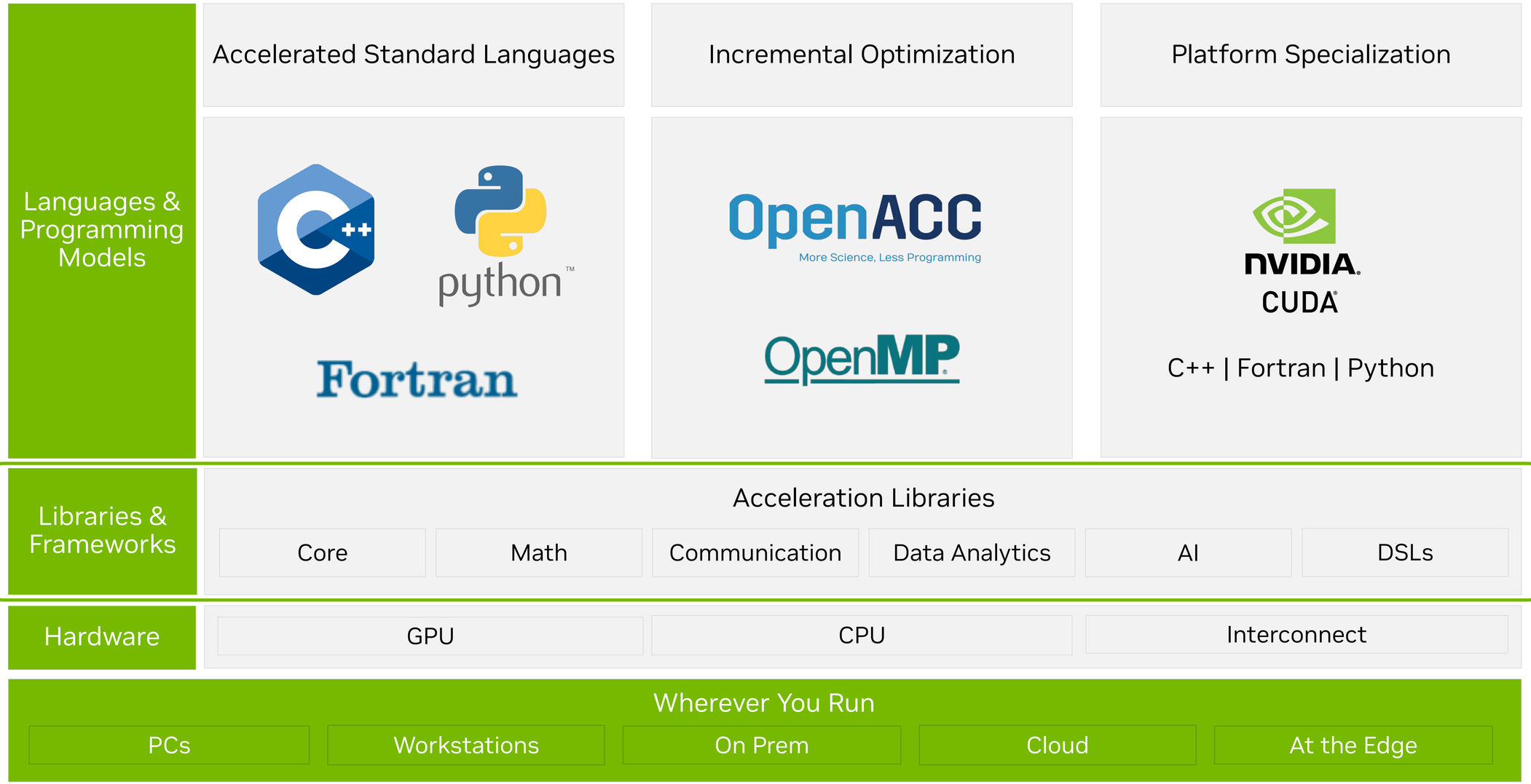
Physics



CPU Server: Dual Xeon Platinum 8480+ @2GHz | GPU Server: Dual Xeon Platinum 8480+ @2GHz with 4x NVIDIA H200 | GTC Benchmark: moi#proc.in, CUDA Version: 12.4 | MILC Benchmark: Apex Medium, CUDA Version: 12.4

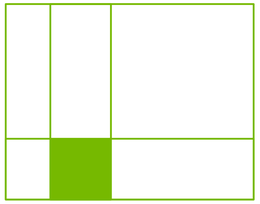
Programming the NVIDIA Platform

Unmatched developer flexibility

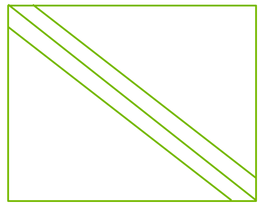


NVIDIA Math Libraries

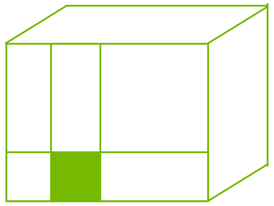
Linear Algebra, FFT, RNG, and Basic Math



cuBLAS

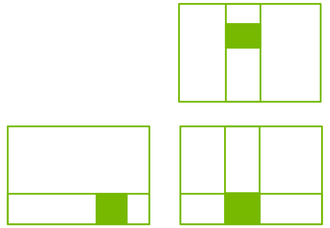


cuSPARSE

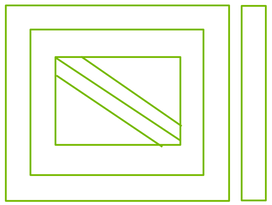


cuTENSOR

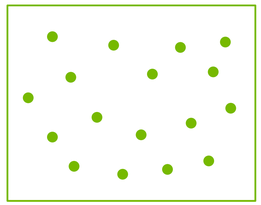
cuSOLVER



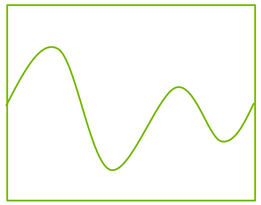
CUTLASS



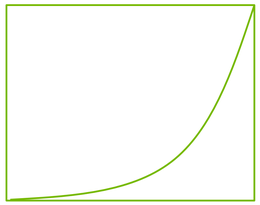
AMGX



cuRAND



cuFFT



Math API

Partner Libraries

Programming the NVIDIA Platform

Unmatched Developer Flexibility

	Accelerated Standard Languages	Incremental Optimization	Platform Specialization
Languages & Programming Models	  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)
{
  std::transform(std::execution::par, x, x+n, y, y,
    [=](float xi, float yi) {
      return a*xi + yi;
    }
  );
}


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

Standard Language

NVIDIA

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

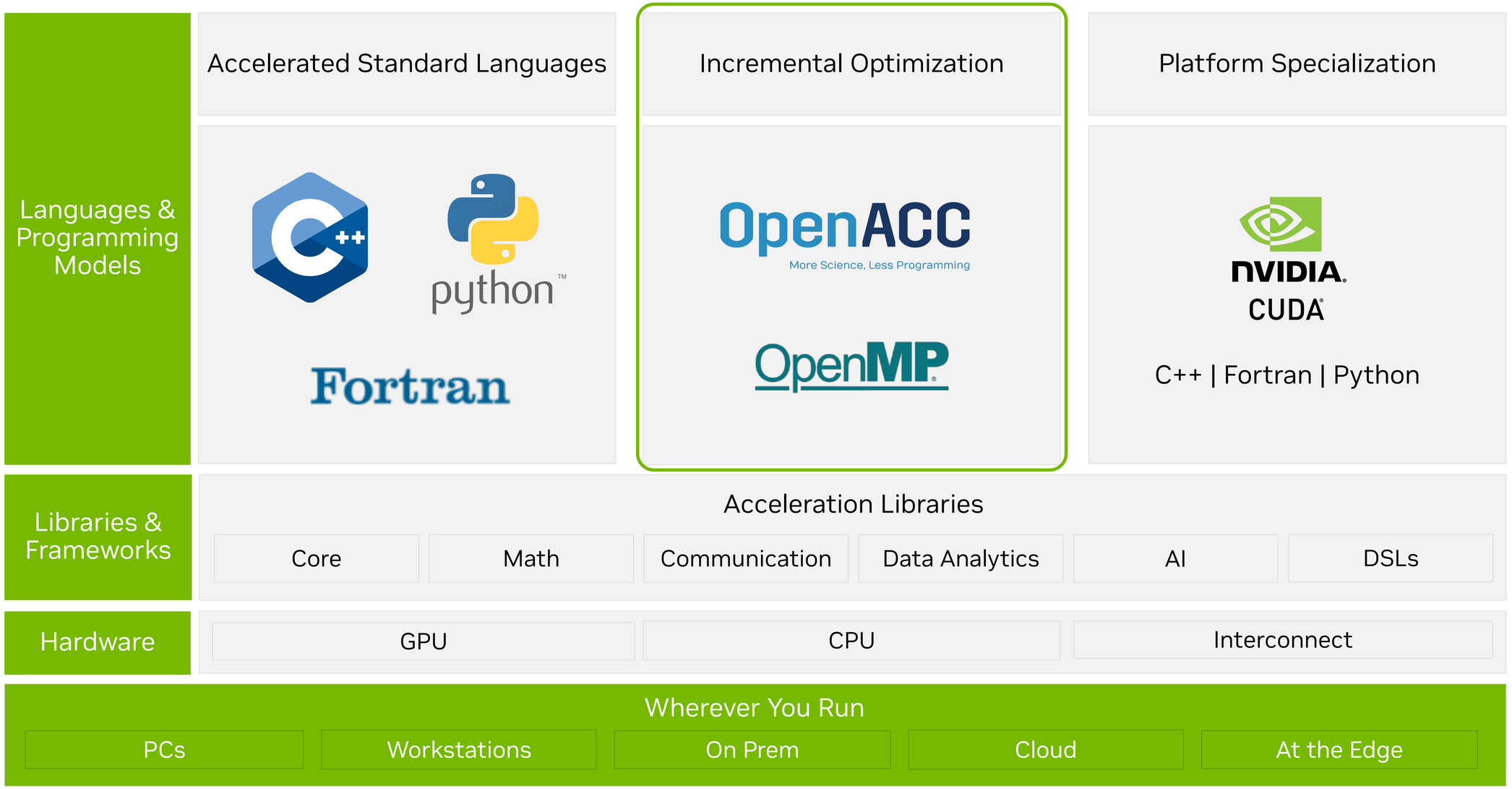
```
subroutine saxpy(n, a, x, y)
  real :: a, x(:), y(:)
  integer :: n, i

  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

NVIDIA

Programming the NVIDIA Platform

Unmatched Developer Flexibility



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

NVIDIA

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

NVIDIA.

Programming the NVIDIA Platform

Unmatched Developer Flexibility

Languages & Programming Models	Accelerated Standard Languages	Incremental Optimization	Platform Specialization
	  Fortran	OpenACC More Science, Less Programming 	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

NVIDIA

Standard Language Parallelism

HPC Programming in ISO C++

ISO is the place for portable concurrency and parallelism

C++17 & C++20

Parallel Algorithms

- Parallel and vector concurrency

Forward Progress Guarantees

- Extend the C++ execution model for accelerators

Memory Model Clarifications

- Extend the C++ memory model for accelerators

Ranges

- Simplifies iterating over a range of values

Scalable Synchronization Library

- Express thread synchronization that is portable and scalable across CPUs and accelerators

Preview support coming to NVC++

C++23

`std::mdspan`

- HPC-oriented multi-dimensional array abstractions.
- Preview Available Now

Range-Based Parallel Algorithms

- Improved multi-dimensional loops

Extended Floating Point Types

- First-class support for formats new and old: `std::float16_t/float64_t`

And Beyond

Senders/Receivers

- Standardized mechanism for asynchrony in the C++ standard library
- Simplify launching and managing parallel work across CPUs and accelerators
- Preview Available Now

Linear Algebra

- C++ standard algorithms API to linear algebra
- Maps to vendor optimized BLAS libraries
- Preview Available Now

MDArray and SubMDSpan

- Expands the capabilities of C++23 MDSpan
- Preview Available Now

SAXPY

Standard Language Parallelism (C++)

- NVC++ can compile Standard C++ algorithms with the parallel execution policies
- An NVC++ command-line option, -stdpar, is used to enable GPU-accelerated C++ Parallel Algorithms
- All data movement between host memory and GPU device memory is performed implicitly and automatically under the control of CUDA Unified Memory

```
void saxpy(int n, float a,
           float *x, float *y)
{
  std::transform(std::execution::par, x, x+n, y, y,
    [=](float xi, float yi) {
      return a*xi + yi;
    }
  );
}


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

HPC Programming in ISO Fortran

ISO is the place for portable concurrency and parallelism

Fortran 2018

Fortran Array Intrinsics

- NVFORTRAN 20.5
- Accelerated matmul, reshape, spread, …

DO CONCURRENT

- NVFORTRAN 20.11
- Auto-offload & multi-core

Co-Arrays

- Not currently available
- Accelerated co-array images

Preview support available now in NVFORTRAN

Fortran 202x

DO CONCURRENT Reductions

- NVFORTRAN 21.11
- REDUCE subclause added
- Support for +, *, MIN, MAX, IAND, IOR, IEOR.
- Support for .AND., .OR., .EQV., .NEQV on LOGICAL values

SAXPY

Standard Language Parallelism (Fortran)

- NVFORTRAN can compile `do concurrent`
- An NVFORTRAN command-line option, -stdpar, is used to enable GPU-accelerated `do concurrent`
- All data movement between host memory and GPU device memory is performed implicitly and automatically under the control of CUDA Unified Memory

```
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

```
$ nvc++ -stdpar=gpu -Minfo=stdpar saxpy.c
```

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

OpenACC

Why Developers Love OpenACC

Incremental

- Maintain existing sequential code
- Add annotations to expose parallelism
- After verifying correctness, annotate more of the code

Single Source

- Rebuild the same code on multiple architectures
- Compiler determines how to parallelize for the desired machine
- Sequential code is maintained

Low Learning Curve

- OpenACC is meant to be easy to use, and easy to learn
- Programmer remains in familiar C, C++, or Fortran
- No reason to learn low-level details of the hardware.

SAXPY

OpenACC

- Similar to OpenMP CPU code
  - `#pragma omp ... -> #pragma acc ...`
  - Additional description about data transfer

```
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

OpenACC (Fortran)

- Similar to OpenMP CPU code
  - `!$ omp ... -> !$acc ...`
  - Additional description about data transfer

```
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)
...
```

Parallel Directive

- Parallel directive with loop directive marks the region for parallel execution and distributes the iterations of the loop.

```
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);
...
```

Kernels Directive

- The compiler will analyze the loops and parallelize those if it finds safe and profitable to do so.
- If the compiler decides that the loop is not parallelizable, it will not parallelize the loop.

```
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);
...
```

Kernels vs. Parallel

Kernels	Parallel
• Compiler decides what to parallelize with direction from user	• Programmer decides what to parallelize and communicates that to the compiler
• Compiler guarantees correctness	• Programmer guarantees correctness
• Can cover multiple loop nests	• Must decorate each loop
When fully optimized, both will give similar performance.	

NVIDIA

Loop Directive

- Must be contained within an OpenACC compute region (either a parallel or a kernels region)
- Allows the programmer to give additional information and/or optimizations about the loop
  - `independent` : 並列化可能
  - `seq` : 逐次実行
  - `collapse` : ループ融合
  - `gang/vector` : 並列化粒度
  - …

```
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);
...
```

Data Clause

- Used with parallel / kernels directive

- Define data movement between host and device
  - `copyin` : parallel 領域開始時に CPU -> GPU
  - `copyout` : parallel 領域終了時に CPU <- GPU
  - `copy` : copyin と copyout の両方
  - `create` : GPU 上にメモリを確保

```
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);
...
```

Data Directive

- Defines a lifetime for data on the device beyond individual loops
- During the region data is essentially “owned by “ the accelerator
- Data movement between host and device
  - `copyin` ： データ領域開始時に CPU -> GPU
  - `copyout` ： データ領域終了時に CPU <- GPU
  - `copy` ： copyin と copyout の両方
  - `create` ： GPU 上にメモリを確保

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

CUDA Unified Memory

- Handling explicit data transfers between the host and device (CPU and GPU) can be difficult
- The NVIDIA HPC Compilers can utilize CUDA Managed Memory to defer data management
- Dynamically allocated memory would be under control of Unified Memory with an appropriate compiler flag (`-gpu=managed`)

```
void saxpy(int n,
           float a,
           float *x,
           float *restrict y)
{
  #pragma acc parallel loop
  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` : 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` : CUDA Unified Memory 有効化

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

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

CUDA

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

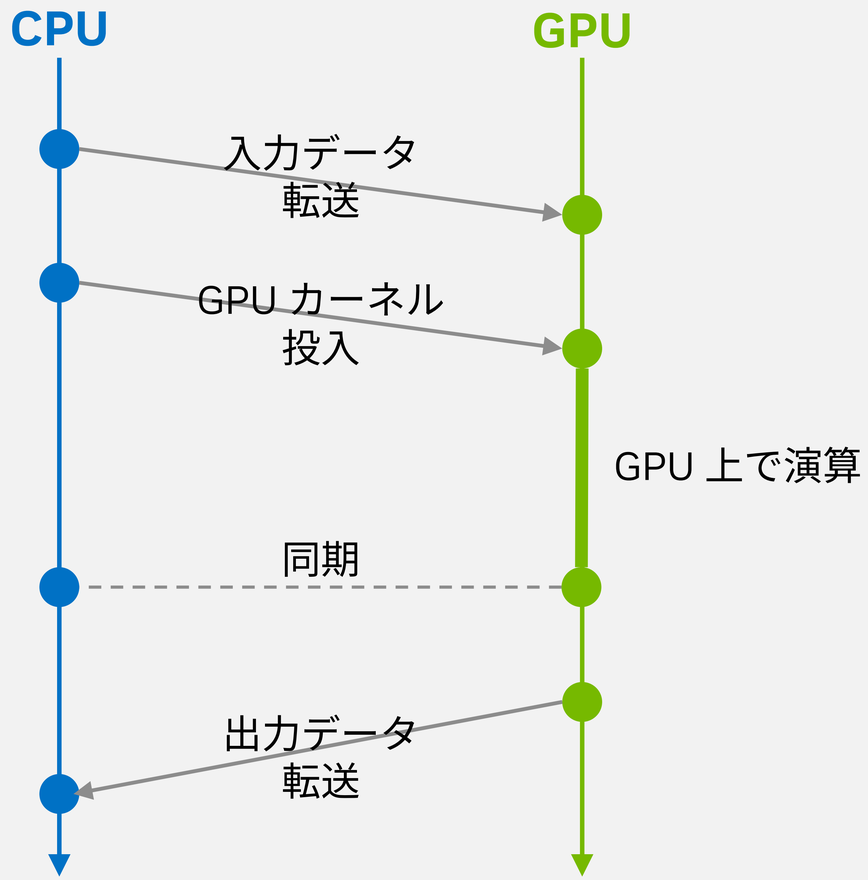
```
__global__ void saxpy(int n, float a,
                      float *x, float *y)
{
  int i = threadIdx.x + blodkDim.x * blockIdx;
  if (i < n)
    y[i] += a*x[i];
}
...

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

NVIDIA

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

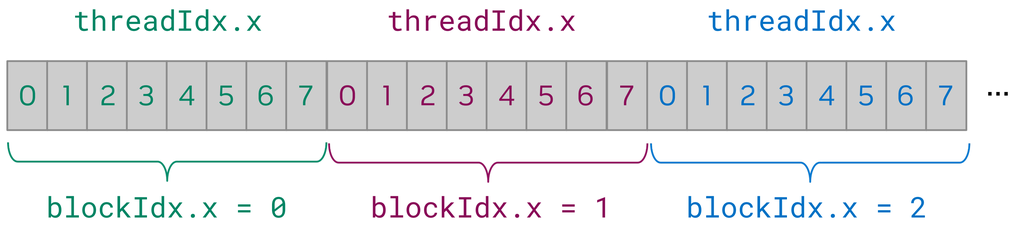
- GPU メモリ確保
- 入力データ転送
- カーネル起動
- 同期
- **出力データ転送**

```
__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

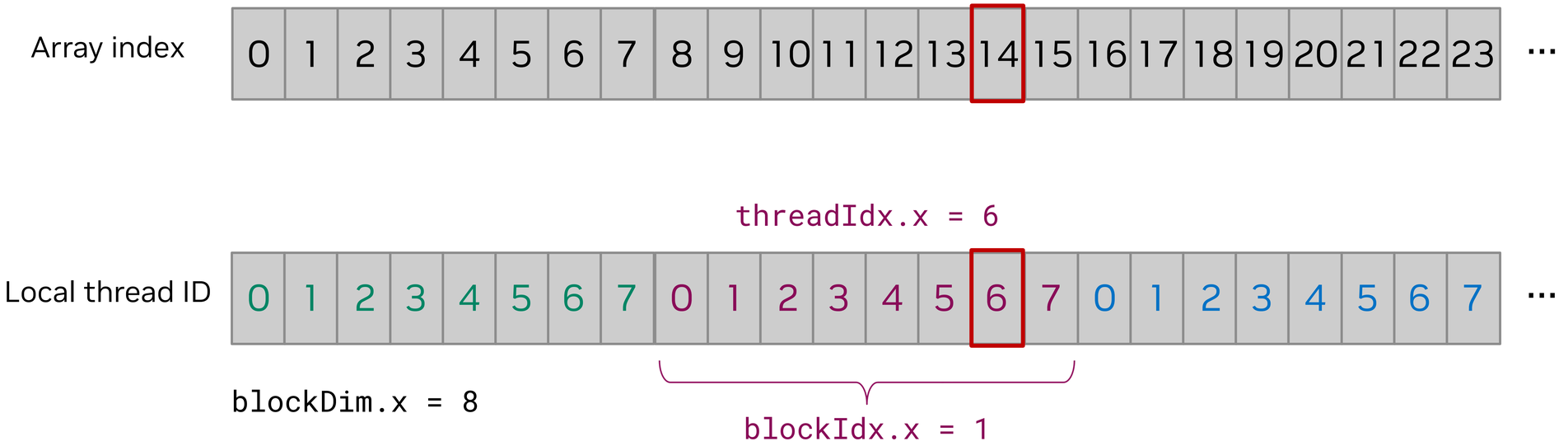



```
__global__ void saxpy(int n, float a,
                      float *x, float *y)
{
  int i = threadIdx.x + blodkDim.x * blockIdx;
  if (i < n)
    y[i] += a*x[i];
}
...

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

Global Thread ID

- Which thread will operate on the red element?




```
int i = threadIdx.x + blockIdx.x * blockDim.x;
      =     6       +     1      *      8    ;
      = 14;
```

NVIDIA.

Execution Configuration

- <<< number_of_blocks, block_size >>>
- Block_size should be multiple of 32
- For block_size good numbers to start with would be 128 or 256

```
__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 Unified Memory

- A single pointer value enables CPUs and GPUs to access a single Unified Memory pool
- GPU programs may access Unified Memory from GPU and CPU threads concurrently without needing to create separate allocations (`cudaMalloc()`) and copy memory manually back and forth (`cudaMemcpy*()`)
- CUDA APIs to allocate Unified Memory (`cudaMallocManaged()`)

```
__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);


cudaMallocManaged(&x, size);
cudaMallocManaged(&y, size);
...



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

CUDA Unified Memory

Prefetch

- CUDA Unified Memory may not always have all the information necessary to make the best performance decisions related to unified memory

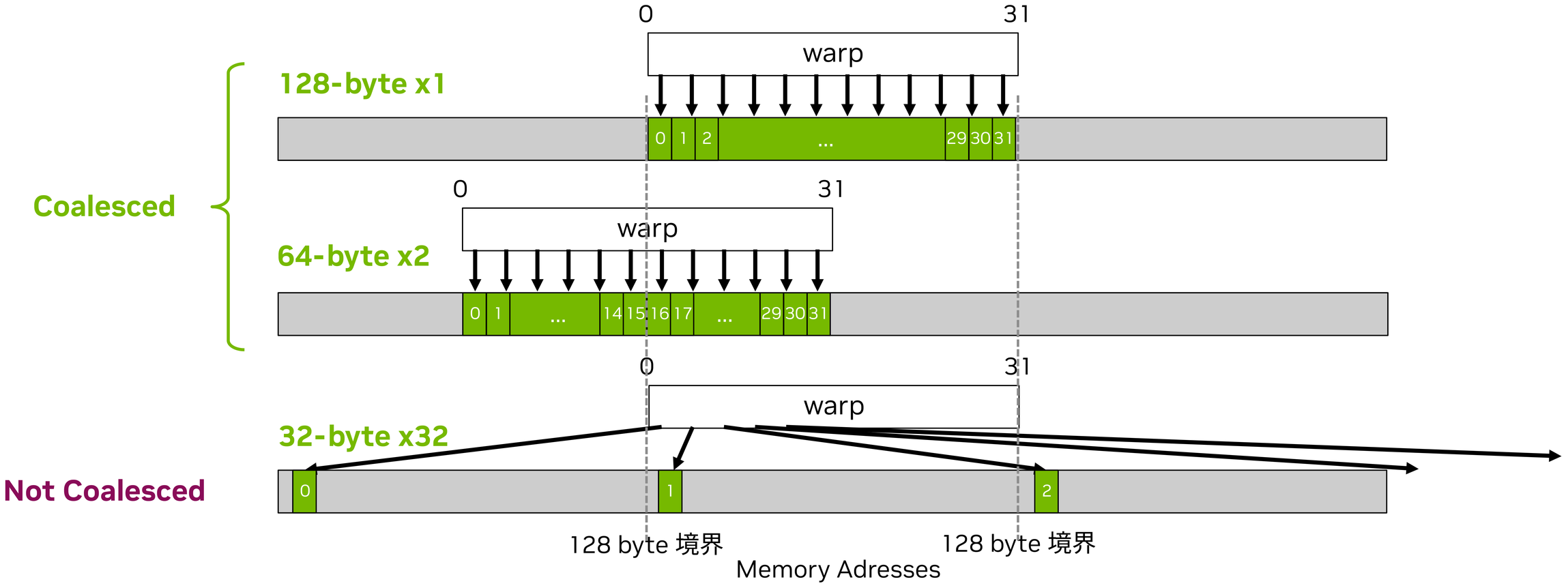
- The `cudaMemPrefetchAsync` API is an asynchronous API that may migrate data to reside closer to the specified processor

```
__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);


cudaMallocManaged(&x, size);
cudaMallocManaged(&y, size);
...

cudaMemPrefetchAsync(x, size, dstDevice);
cudaMemPrefetchAsync(y, size, dstDevice);
saxpy<<< N/128, 128 >>>(N, 3.0, x, y);
cudaDeviceSynchronize();
```

デイバイスメモリへのアクセスは、まとめて

性能 Tips

- 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: 32, 64, or 128 bytes
- トランザクション数は、少ないほどよい
  - 例: 4-byte element access, 128-byte transaction



NVIDIA Developer Tools

- Nsight Systems -

NSIGHT SYSTEMS

System profiler

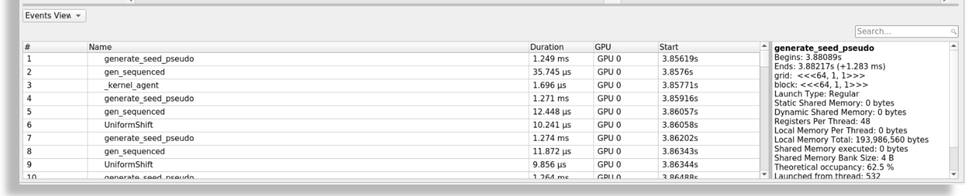
Key Features:

- System-wide application algorithm tuning
  - Multi-process tree support
- Locate optimization opportunities
  - Visualize millions of events on a very fast GUI timeline
  - Or gaps of unused CPU and GPU time
- Balance your workload across multiple CPUs and GPUs
  - CPU algorithms, utilization and thread state
    GPU streams, kernels, memory transfers, etc
- Command Line, Standalone, IDE Integration

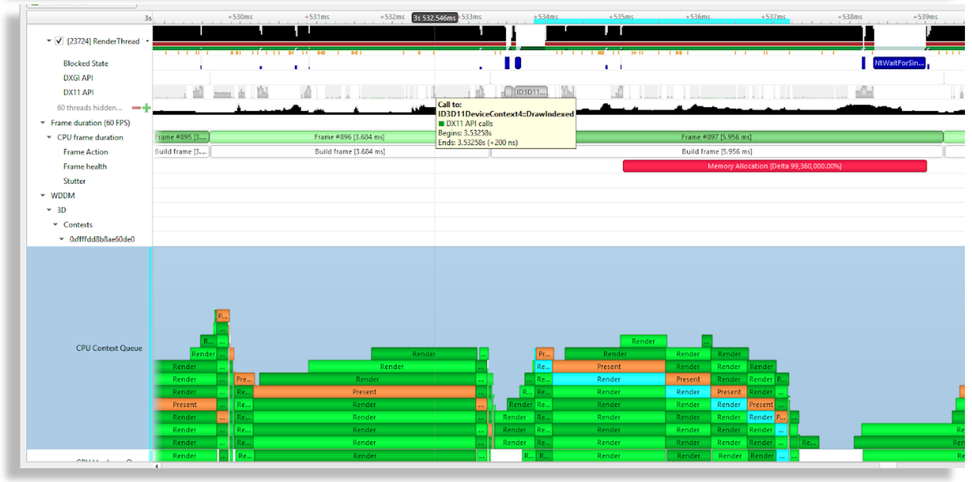
OS: Linux (x86, Power, Arm SBSA, Tegra), Windows, MacOSX (host)

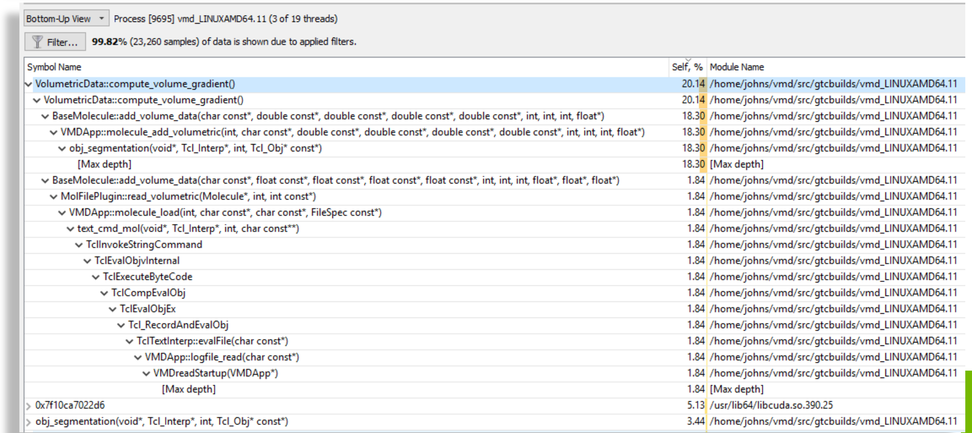
GPUs: Pascal+

Docs/product: https://developer.nvidia.com/nsight-systems

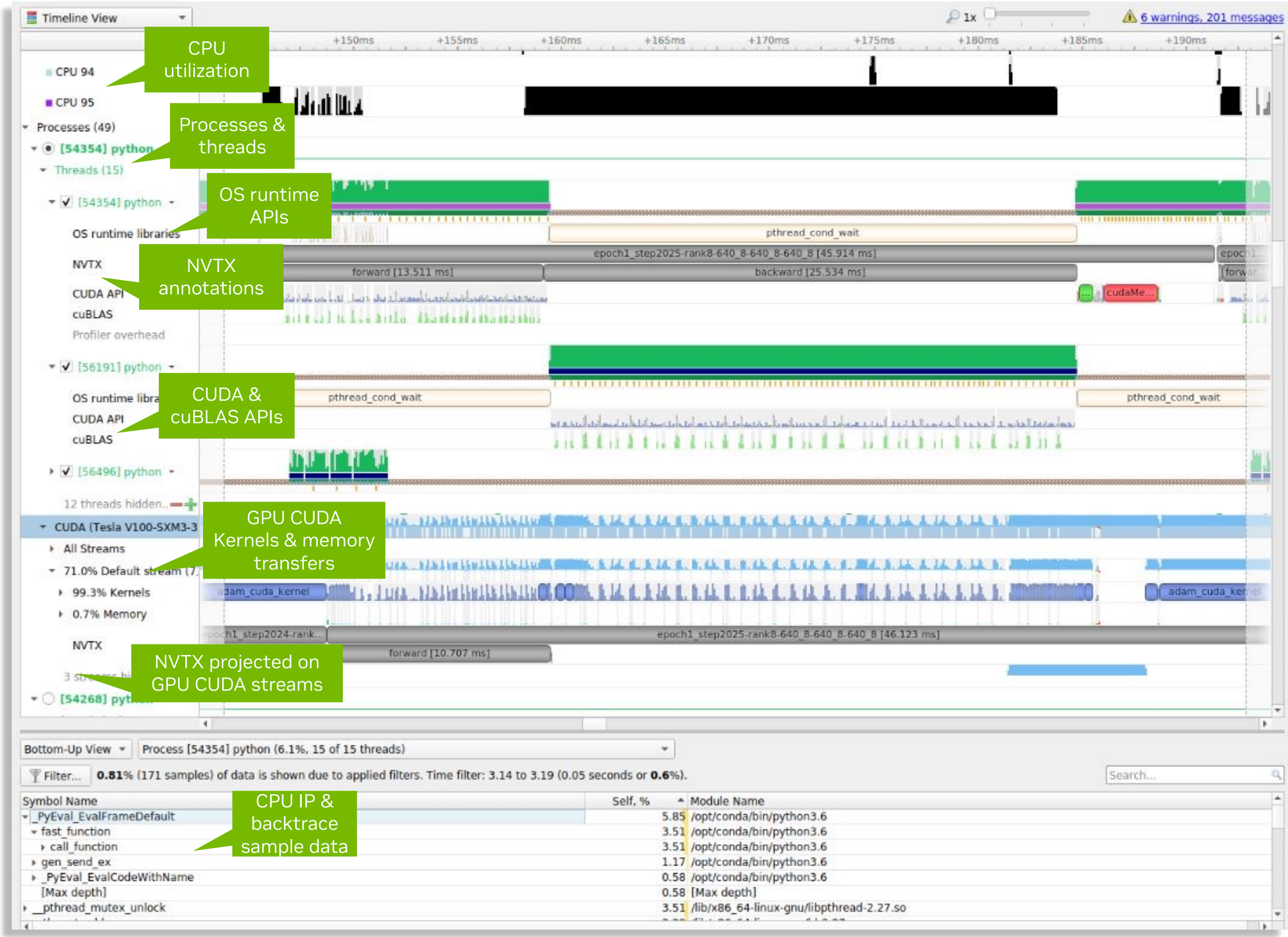


#	Name	Duration	GPU	Start
1	generate_seed_pseudo	1.249 ms	GPU 0	3.85619s
2	gen_sequenced	35.745 µs	GPU 0	3.8576s
3	_kernel_agent	1.696 µs	GPU 0	3.85771s
4	generate_seed_pseudo	1.271 ms	GPU 0	3.85916s
5	gen_sequenced	12.448 µs	GPU 0	3.86057s
6	UniformShift	10.241 µs	GPU 0	3.86058s
7	generate_seed_pseudo	1.274 ms	GPU 0	3.86202s
8	gen_sequenced	11.872 µs	GPU 0	3.86343s
9	UniformShift	9.856 µs	GPU 0	3.86344s





Symbol Name	Self, %	Module Name
VolumetricData::compute_volume_gradient()	20.14	/home/johns/vmd/src/gtcbuilds/vmd_LINUXAMD64.11
VolumetricData::compute_volume_gradient()	20.14	/home/johns/vmd/src/gtcbuilds/vmd_LINUXAMD64.11
BaseMolecule::add_volume_data(char const*, double const*, double const*, double const*, double const*, int, int, int, float*)	18.30	/home/johns/vmd/src/gtcbuilds/vmd_LINUXAMD64.11
VMDApp::molecule_add_volumetric(int, char const*, double const*, double const*, double const*, double const*, int, int, int, float*)	18.30	/home/johns/vmd/src/gtcbuilds/vmd_LINUXAMD64.11
obj_segmentation(void*, Tcl_Interp*, int, Tcl_Obj* const*)	18.30	/home/johns/vmd/src/gtcbuilds/vmd_LINUXAMD64.11
[Max depth]	18.30	[Max depth]
BaseMolecule::add_volume_data(char const*, float const*, float const*, float const*, float const*, int, int, int, float*, float*, float*)	1.84	/home/johns/vmd/src/gtcbuilds/vmd_LINUXAMD64.11
MolFilePlugin::read_volumetric(Molecule*, int, int const*)	1.84	/home/johns/vmd/src/gtcbuilds/vmd_LINUXAMD64.11
VMDApp::molecule_load(int, char const*, char const*, FileSpec const*)	1.84	/home/johns/vmd/src/gtcbuilds/vmd_LINUXAMD64.11
text_cmd_mol(void*, Tcl_Interp*, int, char const**)	1.84	/home/johns/vmd/src/gtcbuilds/vmd_LINUXAMD64.11
TclInvokeStringCommand	1.84	/home/johns/vmd/src/gtcbuilds/vmd_LINUXAMD64.11
TclEvalObjvInternal	1.84	/home/johns/vmd/src/gtcbuilds/vmd_LINUXAMD64.11
TclExecuteByteCode	1.84	/home/johns/vmd/src/gtcbuilds/vmd_LINUXAMD64.11
TclCompEvalObj	1.84	/home/johns/vmd/src/gtcbuilds/vmd_LINUXAMD64.11
TclEvalObjEx	1.84	/home/johns/vmd/src/gtcbuilds/vmd_LINUXAMD64.11
Tcl_RecordAndEvalObj	1.84	/home/johns/vmd/src/gtcbuilds/vmd_LINUXAMD64.11
TclTextInterp::evalFile(char const*)	1.84	/home/johns/vmd/src/gtcbuilds/vmd_LINUXAMD64.11
VMDApp::logfile_read(char const*)	1.84	/home/johns/vmd/src/gtcbuilds/vmd_LINUXAMD64.11
VMDreadStartup(VMDApp*)	1.84	/home/johns/vmd/src/gtcbuilds/vmd_LINUXAMD64.11
[Max depth]	1.84	[Max depth]
0x7f10ca7022d6	5.13	/usr/lib64/libcuda.so.390.25
obj_segmentation(void*, Tcl_Interp*, int, Tcl_Obj* const*)	3.44	/home/johns/vmd/src/gtcbuilds/vmd_LINUXAMD64.11


Timeline View
1x
6 warnings, 201 messages
+150ms
+155ms
+160ms
+165ms
+170ms
+175ms
+180ms
+185ms
+190ms
CPU utilization
CPU 94
CPU 95
Processes & threads
Processes (49)
[54354] python
Threads (15)
[54354] python
OS runtime APIs
OS runtime libraries
pthread_cond_wait
NVTX annotations
NVTX
epoch1_step2025-rank8-640_8-640_8-640_8 [45.914 ms]
forward [13.511 ms]
backward [25.534 ms]
CUDA API
cuBLAS
Profiler overhead
[56191] python
CUDA & cuBLAS APIs
OS runtime libra
pthread_cond_wait
pthread_cond_wait
CUDA API
cuBLAS
[56496] python
12 threads hidden...
GPU CUDA Kernels & memory transfers
CUDA (Tesla V100-SXM3-3
All Streams
71.0% Default stream (7
99.3% Kernels
0.7% Memory
adam_cuda_kernel
adam_cuda_kernel
epoch1_step2025-rank8-640_8-640_8-640_8 [46.123 ms]
NVTX
forward [10.707 ms]
NVTX projected on GPU CUDA streams
[54268] python
Bottom-Up View
Process [54354] python (6.1%, 15 of 15 threads)
Filter...
0.81% (171 samples) of data is shown due to applied filters. Time filter: 3.14 to 3.19 (0.05 seconds or 0.6%).
Search...
CPU IP & backtrace sample data
Symbol Name
Self, %
Module Name
_PyEval_EvalFrameDefault 5.85 /opt/conda/bin/python3.6
fast_function 3.51 /opt/conda/bin/python3.6
call_function 3.51 /opt/conda/bin/python3.6
gen_send_ex 1.17 /opt/conda/bin/python3.6
_PyEval_EvalCodeWithName 0.58 /opt/conda/bin/python3.6
[Max depth] 0.58 [Max depth]
__pthread_mutex_unlock 3.51 /lib/x86_64-linux-gnu/libpthread-2.27.so

Nsight Systems 101

Quick start

- Nsight Systems CLI によるプロファイリング

```
$ nsys profile [options] <application> [application-arguments]
```

  - `-t <parameters>` : トレースする API を指定。デフォルトは、`cuda, opengl nvtx, osrt`
  - `--stats <true|false>` : `true` でプログラム実行時の統計情報を標準出力に表示
  - `-o <filename>` : 出力ファイル名を指定
  - `--force-overwrite <true|false>` : `true` で出力ファイルの上書きを許可。デフォルトは `false`

  など... 詳細は、`nsys --help or nsys [specific command] --help` で確認可能

  - `<filename>.nsys-rep` が出力される
- ローカル PC に `<filename>.nsys-rep` を転送し、Nsight Systems UI で可視化

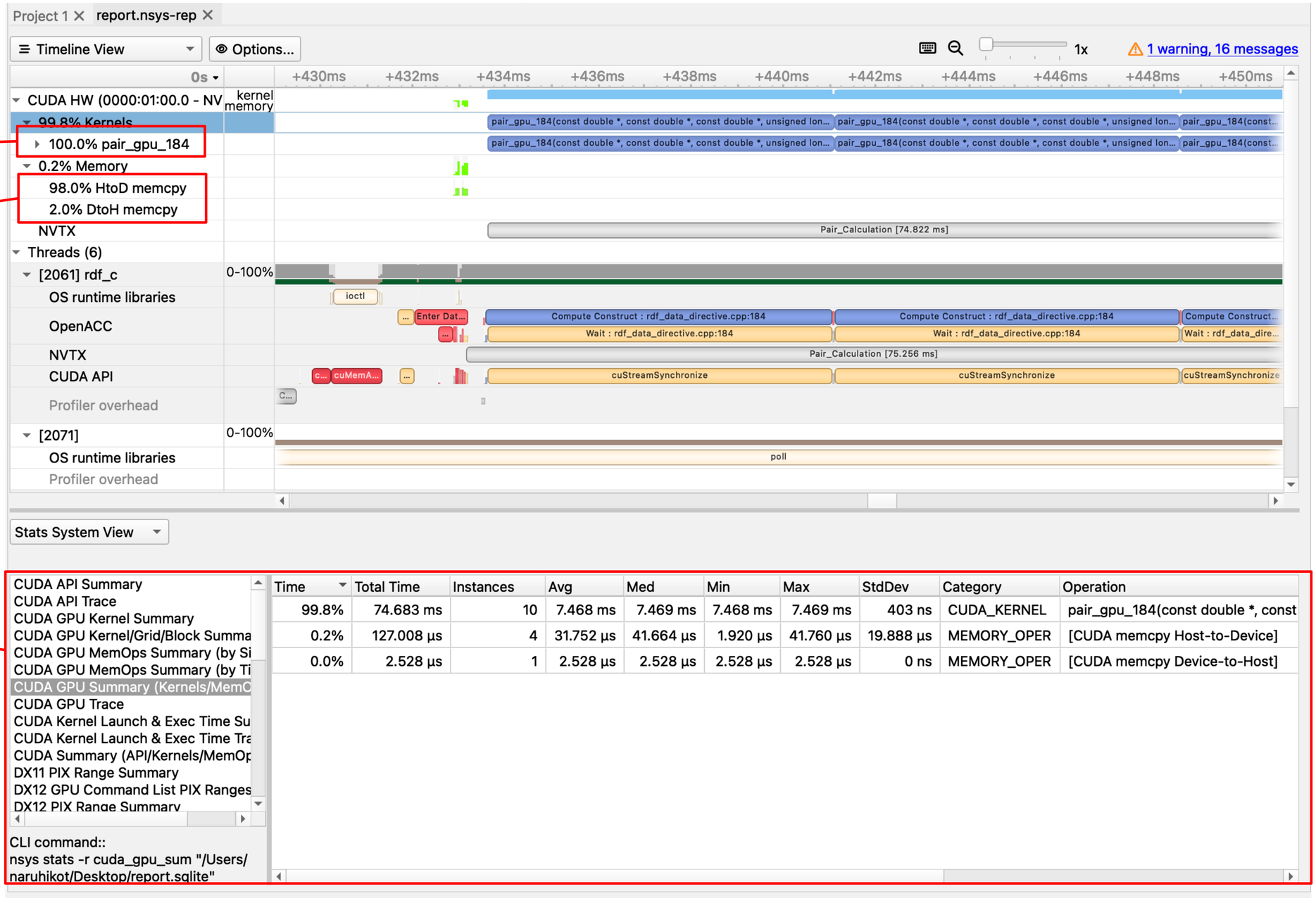
Nsight Systems user guide:
https://docs.nvidia.com/nsight-systems/UserGuide/index.html

Nsight Systems 101

GPU カーネル

データ転送

統計データ



Nsight Systems のローカル端末 へのインストール

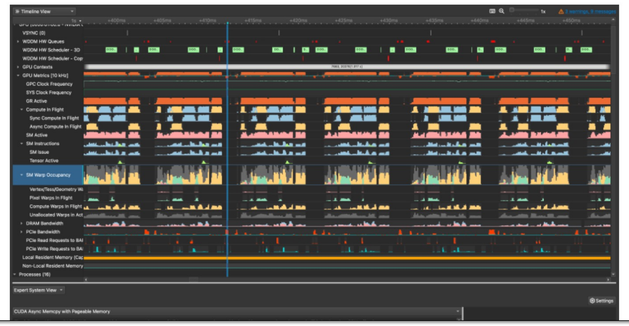
Win/Mac/Linux 用を提供

NVIDIA Nsight Systems

NVIDIA Nsight™ Systems is a system-wide performance analysis tool designed to visualize an application's algorithms, identify the largest opportunities to optimize, and tune to scale efficiently across any quantity or size of CPUs and GPUs, from large servers to our smallest system on a chip (SoC).

Get started

Nsight Systems 2024.1 is available now.



Download

- Latest version
- Supported Platforms
- System Requirements
- Release Notes
- Feature Table
- Archives

Resources

- Documentation
- Self-Paced Training
- Tutorial Sessions
- Video Series
- Support

Download NVIDIA Nsight Systems

Nsight Systems 2024.1 is Available Now

Review the supported platforms for NVIDIA Nsight™ Systems to choose the correct version for your host and profiling target.

If profiling from the CLI, pick your platform based on where the CLI will be run. If using the GUI (Full Version) to view reports, do profiling, or do remote profiling, pick your platform based on the host PC architecture where the GUI will be run.

Also review the system requirements before downloading.

Desktop, workstation, and server platforms:

Download for Windows on x86_64

Download for Linux on x86_64

Download for Linux on Power9

Download for Linux on Arm Servers and NVIDIA Grace

Download for macOS

https://developer.nvidia.com/nsight-systems

Nsight Systems 101

Quick start

- コマンドラインと標準出力で完結する、簡易的なプロファイリングも可能

```
$ nsys profile --stats=true –o <filename> <application> [application-arguments]
$ nsys stats --report cuda_gpu_sum <filename>.sqlite
```

  - `--stats=true` を付加することで、統計情報が含まれた `<filename>.sqlite` が出力される
  - `nsys stats` コマンドで `<filename>.sqlite` を後処理

```
Processing [report1.sqlite] with [/opt/nvidia/hpc_sdk/Linux_x86_64/23.7/profilers/Nsight_Systems/host-linux-
x64/reports/cuda_gpu_sum.py]...

 ** CUDA GPU Summary (Kernels/MemOps) (cuda_gpu_sum):

 Time (%)  Total Time (ns)  Instances  Avg (ns)  Med (ns)  Min (ns)  Max (ns)  StdDev (ns)   Category        Operation
 --------  ---------------  ---------  --------  --------  --------  --------  -----------  -----------  ------------------
     35.0            1,536          1   1,536.0   1,536.0     1,536     1,536          0.0  CUDA_KERNEL  saxpy_4_gpu
     32.8            1,439          2     719.5     719.5       575       864        204.4  MEMORY_OPER  [CUDA memcpy HtoD]
     32.1            1,408          1   1,408.0   1,408.0     1,408     1,408          0.0  MEMORY_OPER  [CUDA memcpy DtoH]
```

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