CUDA 9 AND MORE

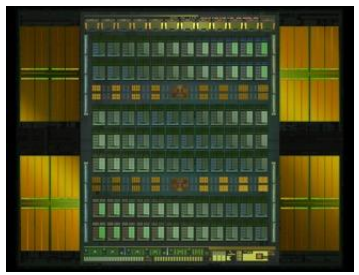
成瀬 彰, シニアデベロッパーテクノロジーエンジニア, 2017/12/12

NVIDIA

CUDA 9の概要

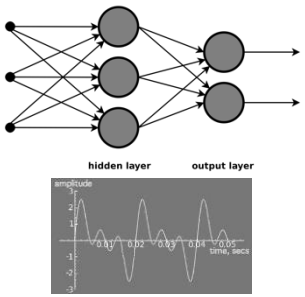
VOLTAに対応

Tesla V100
Voltaアーキテクチャ
Tensorコア
NVLink
Independentスレッドスケジューリング



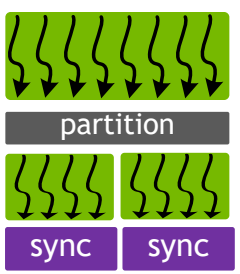
ライブラリの高速化

cuBLAS (主にDL向け)
NPP (画像処理)
cuFFT (信号処理)
cuSolver



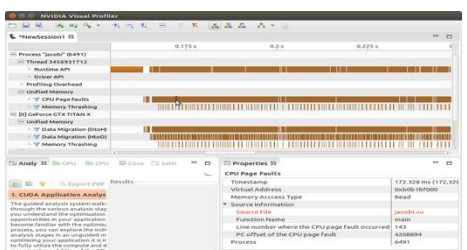
COOPERATIVE GROUPS

柔軟なスレッドグループ
並列アルゴリズムの抽象化
スレッドブロック間の同期(over SM or GPU)



開発ツールの改善

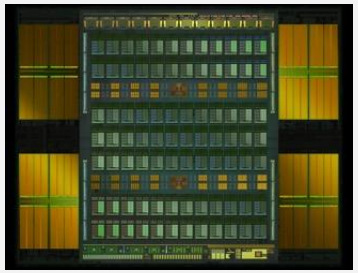
コンパイル時間の短縮
Unified Memoryプロファイル
NVLink可視化
コンパイラサポート



VOLTA対応

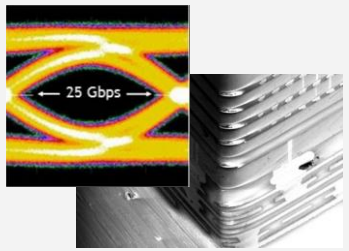
TESLA V100の概要

Voltaアーキテクチャ



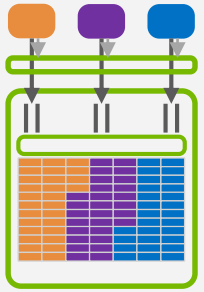
Most Productive GPU

NVLinkとHBM2の改善



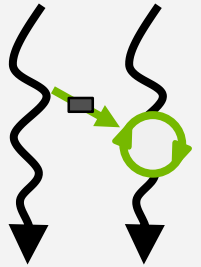
Efficient Bandwidth

Volta MPS



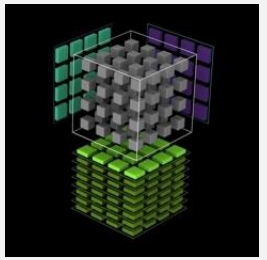
Inference Utilization

SIMTモデルの改善



New Algorithms

Tensorコア



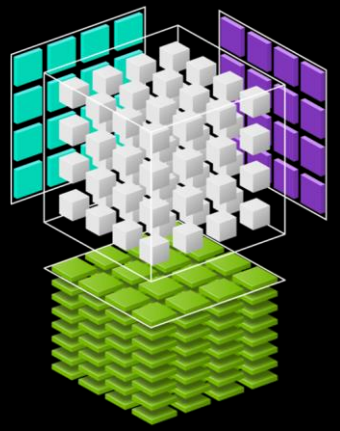
125 Programmable
TFLOPS Deep Learning

DLとHPCの両方に最適なGPU

VOLTA: 米国最大規模スパコンのエンジン

Next Milestone In AI Supercomputing

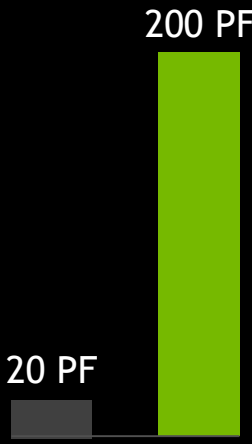
AI Exascale Today



3+EFLOPS

Tensor Ops

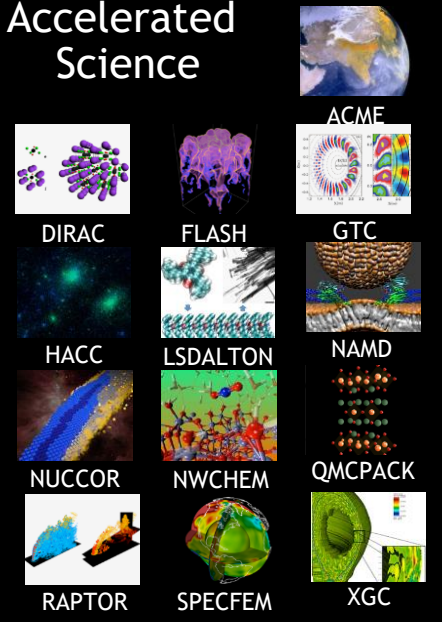
Performance Leadership



10X

Perf Over Titan

Accelerated Science

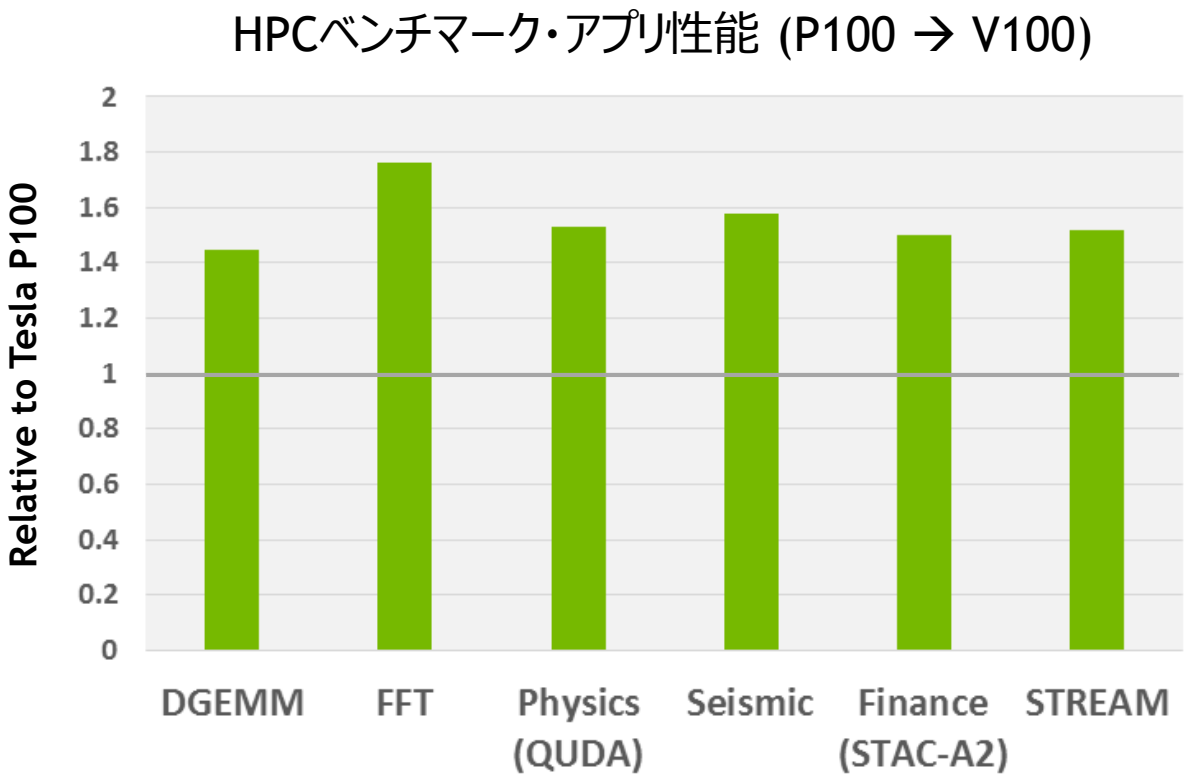


5-10X

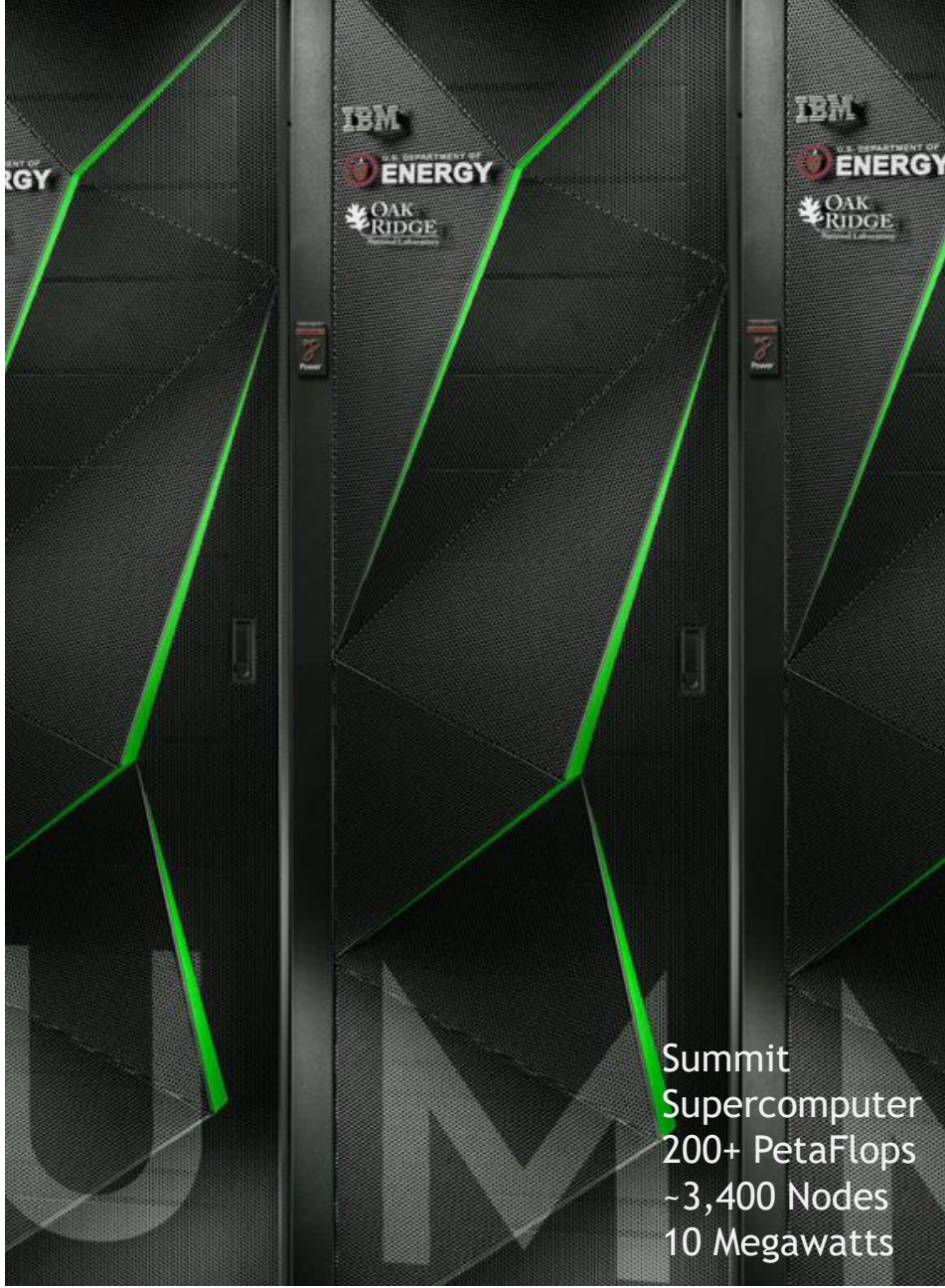
Application Perf Over Titan

エクサスケール(FP64)に向けて

Volta: 米国最大規模スパコンのエンジン

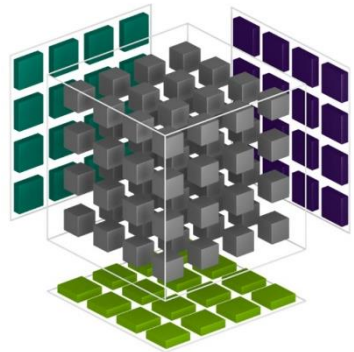


System Config Info: 2X Xeon E5-2690 v4, 2.6GHz, w/ 1X Tesla P100 or V100. V100 measured on pre-production hardware.



TENSORコア

混合精度行列計算ユニット



4x4の行列の積和演算を1サイクルで計算する性能 (128演算/サクル)

行列のFMA (Fused Multiply-Add)

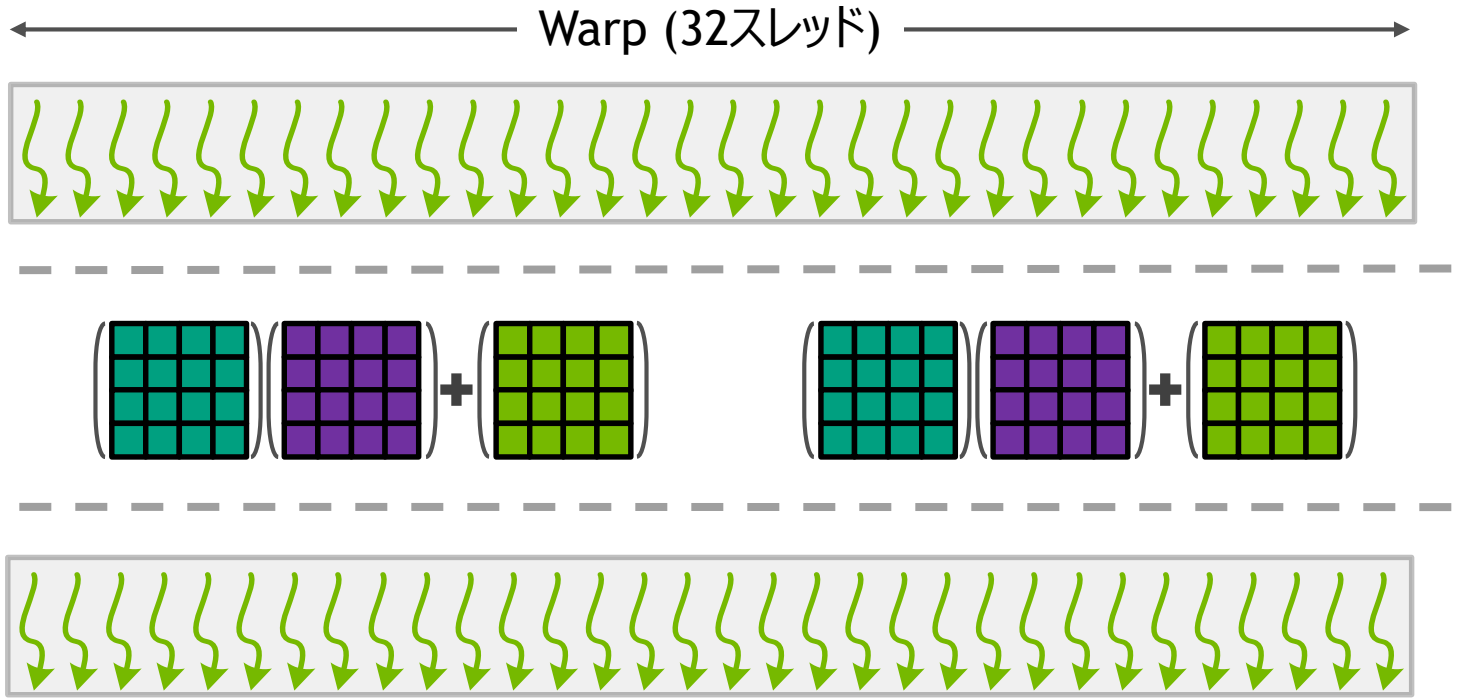
$$D = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}$$

FP16 or FP32　　FP16　　FP16　　FP16 or FP32

$$D = AB + C$$

TENSORコアの使われ方

16x16の行列の積和演算を、ワープレベル(32スレッド)で協調して実行



TENSORコアの使い方

NVIDIA cuBLAS, cuDNN, TensorRT

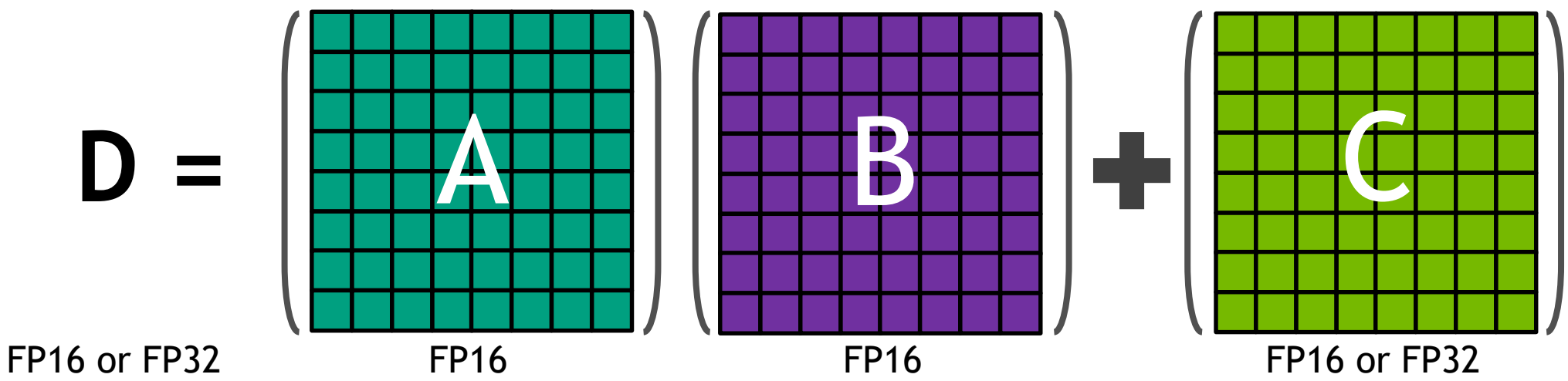
Volta向けに最適化された
フレームワーク・ライブラリ

```
__device__ void tensor_op_16_16_16(
    float *d, half *a, half *b, float *c)
{
  wmma::fragment<matrix_a, …> Amat;
  wmma::fragment<matrix_b, …> Bmat;
  wmma::fragment<matrix_c, …> Cmat;

  wmma::load_matrix_sync(Amat, a, 16);
  wmma::load_matrix_sync(Bmat, b, 16);
  wmma::fill_fragment(Cmat, 0.0f);

  wmma::mma_sync(Cmat, Amat, Bmat, Cmat);

  wmma::store_matrix_sync(d, Cmat, 16,
                          wmma::row_major);
}
```

CUDA C++
Warpレベル行列演算テンプレート

CUDA TENSORコア プログラミング

16x16x16 Warp Matrix Multiply and Accumulate (WMMA)

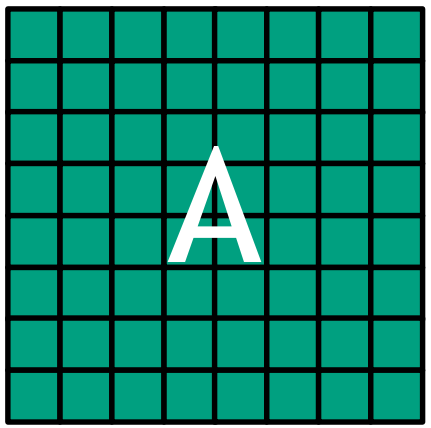


$$D = AB + C$$

CUDA TENSORコア プログラミング

WMMA: 行列データ型

```
wmma::fragment<matrix_a, …> Amat;
```



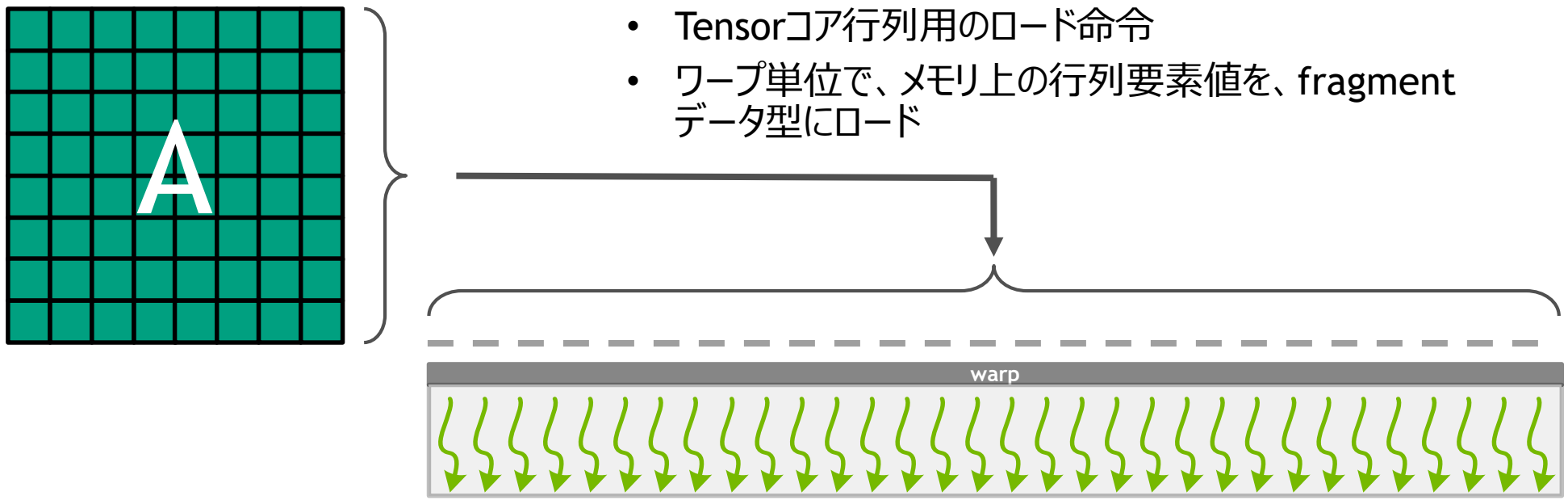
fragment

- Tensorコア用の行列データ型
- 各スレッドは、行列の要素の一部を、自分のレジスタに保持 (割当は未公開)
- ワープレベル(32スレッド)で、行列の全要素を保持すればよいという考え
- 従来のスレッド単位の行列演算と比べ、レジスタ使用量を削減

CUDA TENSORコア プログラミング

WMMA: ロード命令

```
wmma::load_matrix_sync(Amat, a, stride);
```



load_matrix_sync

- Tensorコア行列用のロード命令
- ワープ単位で、メモリ上の行列要素値を、fragment データ型にロード

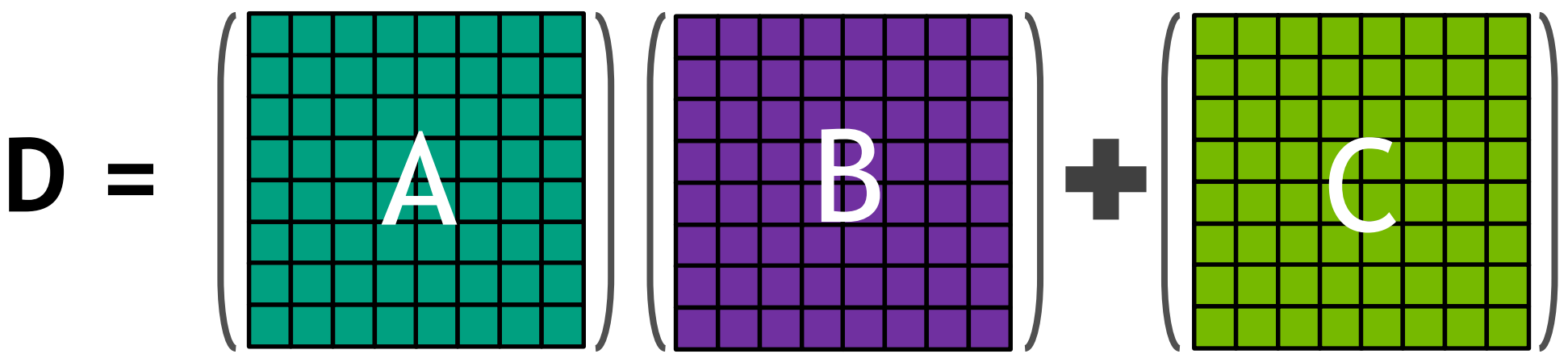
CUDA TENSORコア プログラミング

WMMA: 行列乗算

```
wmma::mma_sync(Dmat, Amat, Bmat, Cmat);
```

mma_sync

- Tensorコアを使用して、行列乗算を実行



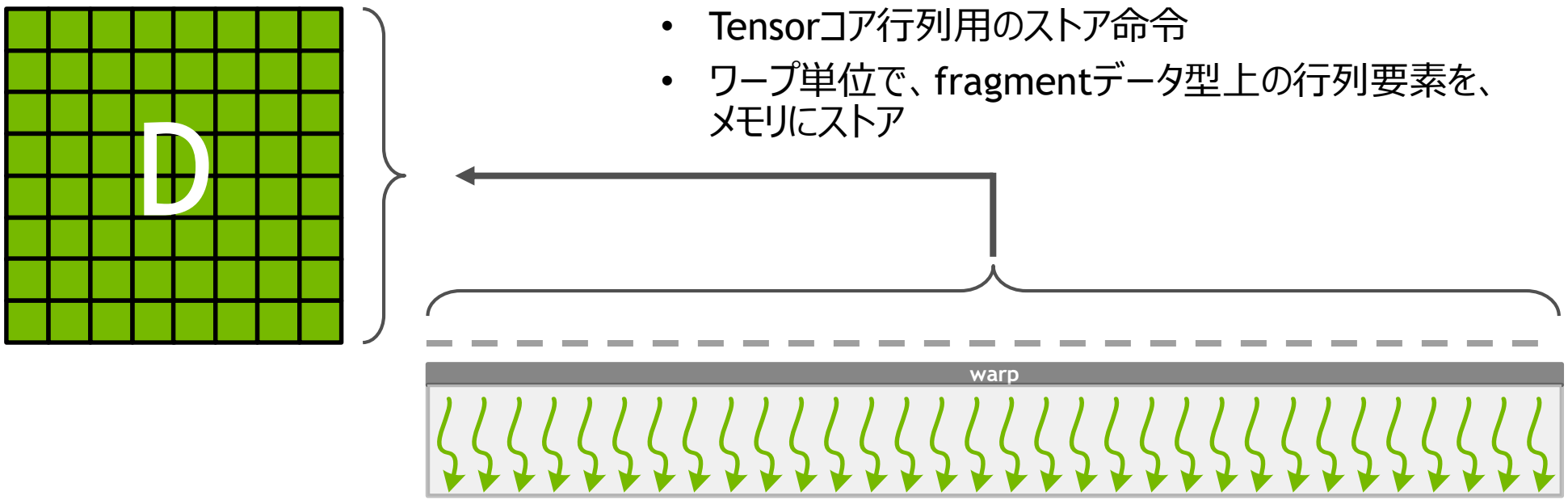
CUDA TENSORコア プログラミング

WMMA: ストア命令

```
wmma::store_matrix_sync(d, Dmat, stride);
```

load_store_sync

- Tensorコア行列用のストア命令
- ワープ単位で、fragmentデータ型上の行列要素を、メモリにストア



TENSORコアの使い方

NVIDIA cuBLAS, cuDNN, TensorRT

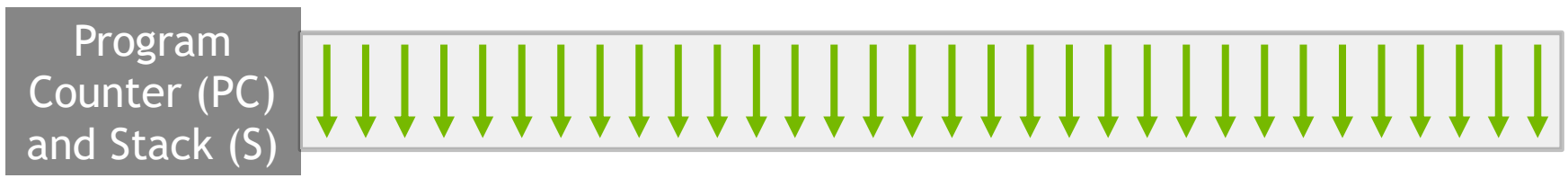
Volta向けに最適化された
フレームワーク・ライブラリ

```
__device__ void tensor_op_16_16_16(
    float *d, half *a, half *b, float *c)
{
  wmma::fragment<matrix_a, …> Amat;
  wmma::fragment<matrix_b, …> Bmat;
  wmma::fragment<matrix_c, …> Cmat;

  wmma::load_matrix_sync(Amat, a, 16);
  wmma::load_matrix_sync(Bmat, b, 16);
  wmma::fill_fragment(Cmat, 0.0f);

  wmma::mma_sync(Cmat, Amat, Bmat, Cmat);

  wmma::store_matrix_sync(d, Cmat, 16,
                          wmma::row_major);
}
```

CUDA C++
Warpレベル行列演算テンプレート

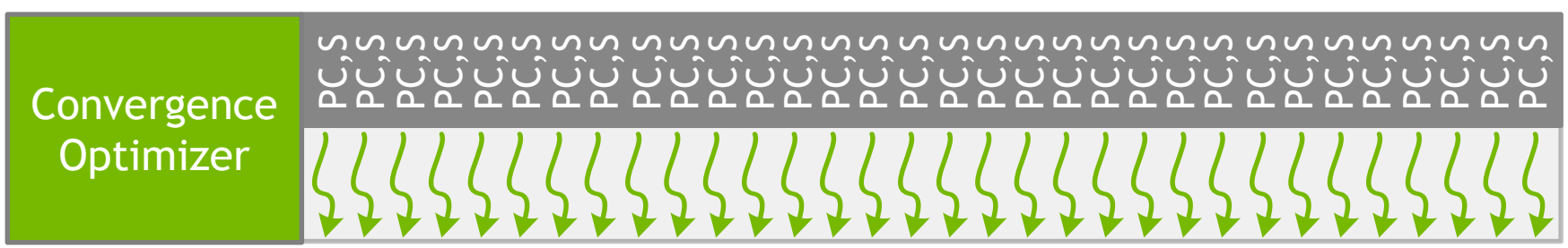
VOLTA INDEPENDENT THREAD SHCEDULING

Pascalまで



Warp(32スレッド)毎に、PCは1つ

Volta



スレッド毎にPCが存在、個別にスケジューリングが可能

WARP同期用ビルドイン関数

CUDA 9で導入

スレッド同期

__syncwarp

アクティブなスレッド(PCの同じスレッド)の取得

__activemask

スレッド間のデータ交換

__all_sync, __any_sync, __uni_sync, __ballot_sync

__shfl_sync, shfl_up_sync, shfl_down_sync, __shfl_xor_sync

__match_any_sync, __match_all_sync

(*) 従来の __shfl, __ballot, __any, __all は、CUDA 9でdeprecated

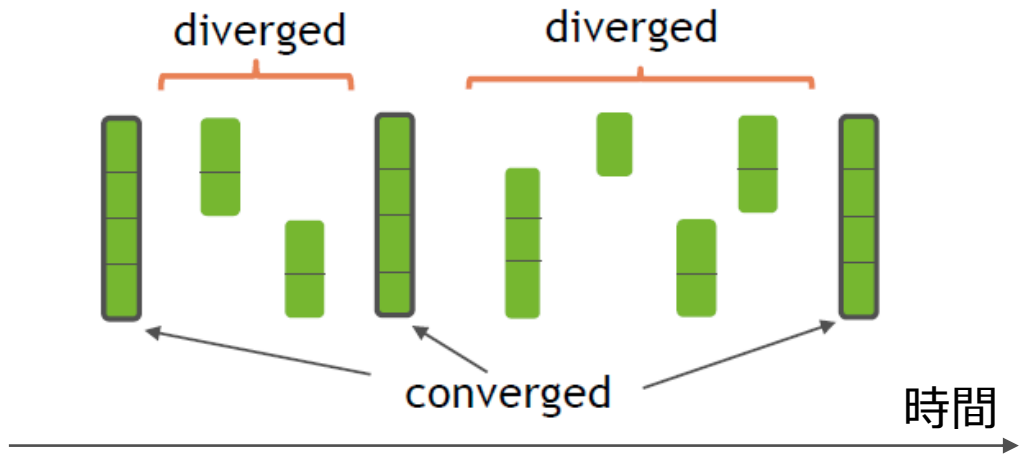
WARP同期プログラミング

想定通りにWarp(32スレッド)が同期する保証はない

```
if (threadIdx.x < 4) {
    A;
} else {
    B;
}

/* 32スレッド同期実行 */

if (threadIdx.x < 8) {
    X;
} else {
    Y;
}
```

ここでWarp内の32スレッドが同期している保証はない

分岐のあるプログラムでは、Warpは分離・集合して命令実行



暗黙のWarp同期を前提としたプログラミングは危険

- 特にVoltaから (Volta以前も、安全ではなかった…)

暗黙的なWARP同期プログラミング

Warpに関して、以下のことを仮定している

1. スレッドは再集合する

```
if (threadIdx.x < 16)
        A;
else
        B;
assert(__activemask() = 0xffffffff);
```

2. スレッドはロックステップ実行する

```
if (__activemask() == 0xffffffff) {
        assert(__activemask() = 0xffffffff);
}
```

暗黙的なWARP同期プログラミング

Warpに関して、以下のことを仮定している

1. スレッドは再集合する

```
if (threadIdx.x < 16)
        A;
else
        B;
assert(__activemask() = 0xffffffff);
```

2. スレッドはロックステップ実行する

```
if (__activemask() == 0xffffffff) {
        assert(__activemask() = 0xffffffff);
}
```

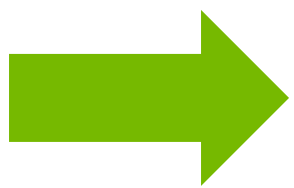
どちらも、Trueになる保証はない

明示的なWarp同期が必要

そのため、CUDA 9でWarp同期のBuild-in関数を追加・改変

例: 共有メモリを使用した ワープ内スレッド間REDUCTION

```
shmem[tid] = v;
v += shmem[tid+16];
shmem[tid] = v;
v += shmem[tid+8
shmem[tid] = v;
v += shmem[tid+4
shmem[tid] = v
v += shmem[tid+2
shmem[tid] = v;
v += shmem[tid+1
shmem[tid] = v;
```



```
shmem[tid] = v;          __syncwarp();
v += shmem[tid+16];      __syncwarp();
shmem[tid] = v;          __syncwarp();
v += shmem[tid+8];       __syncwarp();
shmem[tid] = v;          __syncwarp();
v += shmem[tid+4];       __syncwarp();
shmem[tid] = v;          __syncwarp();
v += shmem[tid+2];       __syncwarp();
shmem[tid] = v;          __syncwarp();
v += shmem[tid+1];       __syncwarp();
shmem[tid] = v;
```

これも、安全ではない

ライブラリの改善

CUDA 9: ライブラリの改善

VOLTA対応

Tensorコアの活用

cuBLAS: Voltaに最適化したGEMMs

全ライブラリ: すぐにVoltaを性能を発揮

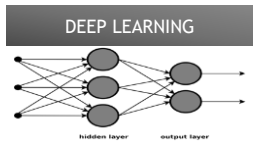
TENSOR CORE

スピード

cuBLAS: RNNs向けGEMM最適化

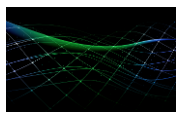
NPP: 画像処理の高速化

cuFFT: 様々なサイズのFFT最適化

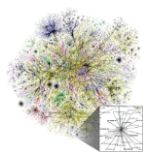


新アルゴリズム

cuSOLVER: マルチGPU向け密行列・疎行列ソルバー、密行列固有値解析



nvGRAPH: 幅優先探索(BFS)、クラスタリング、Triangle-Counting、グラフ挿入・抽出



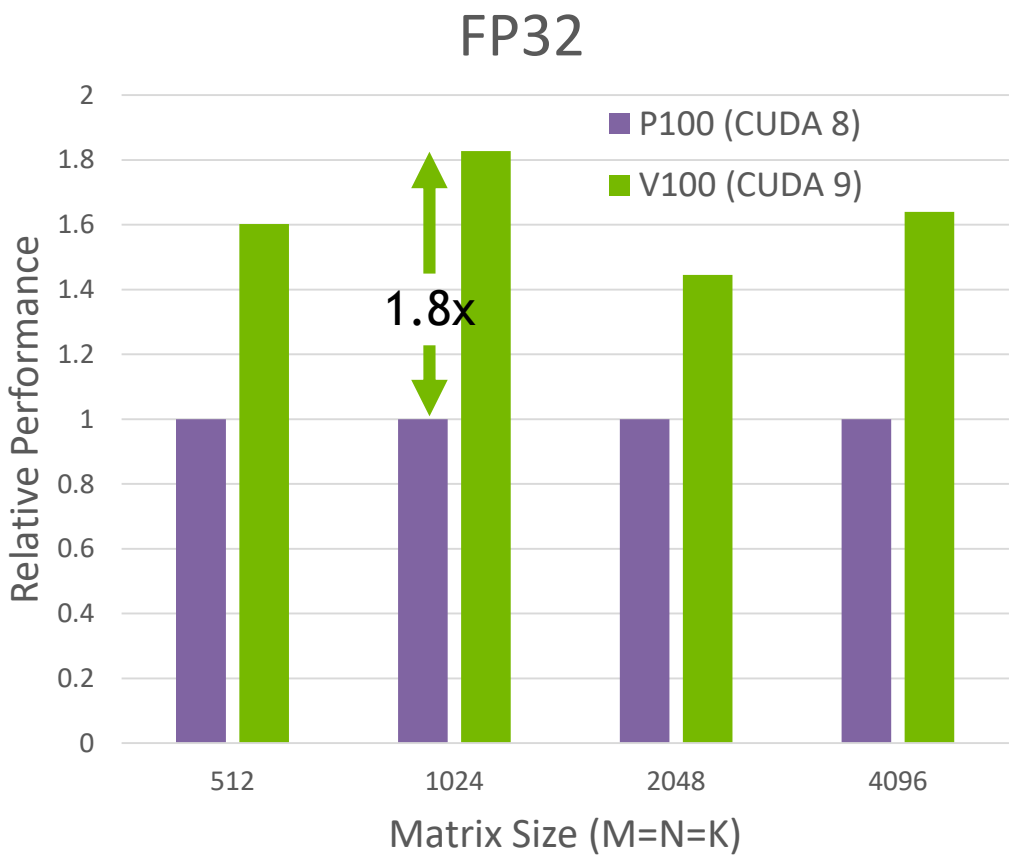
インストール

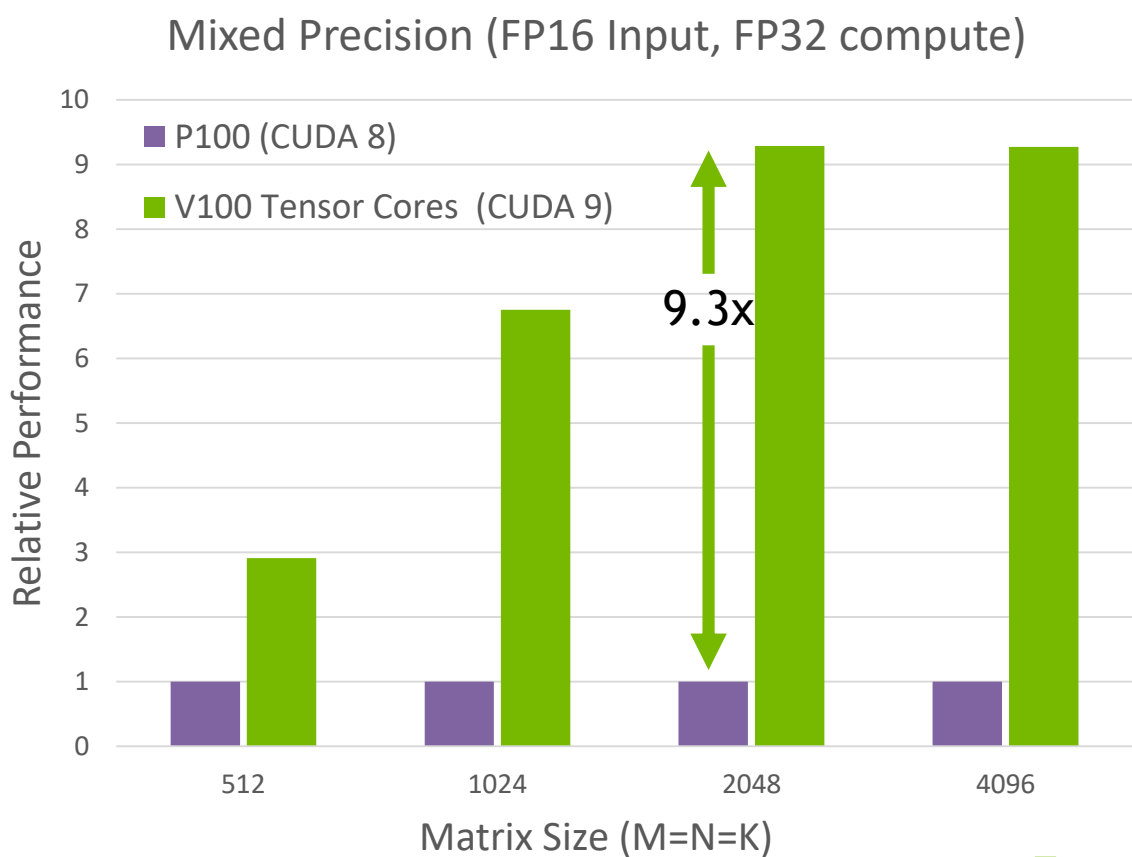
CUDAライブラリだけのパッケージ (without CUDA driver, runtime, etc.)

NPP: モジュラー化

cuBLAS: GEMMS性能改善

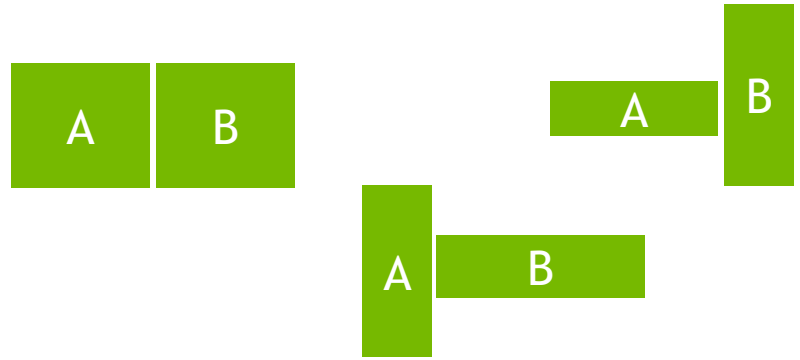
Volta Tensorコア + CUDA 9





cuBLAS: cublasGemmEx()

アルゴリズム選択が可能 (CUDA 8から)




```
cublasStatus_t cublasGemmEx(cublasHandle_t handle,
                            cublasOperation_t transa,
                            cublasOperation_t transb,
                            int m,
                            int n,
                            int k,
                            const void    *alpha,
                            const void     *A,
                            cudaDataType_t Atype,
                            int lda,
                            const void     *B,
                            cudaDataType_t Btype,
                            int ldb,
                            const void    *beta,
                            void           *C,
                            cudaDataType_t Ctype,
                            int ldc,
                            cudaDataType_t computeType,
                            cublasGemmAlgo_t algo)
```

- 18種類のアルゴリズムから選択可能
  - CUBLAS_GEMM_ALGO[0:17]
- CUBLAS_GEMM_DFALT: 自動選択

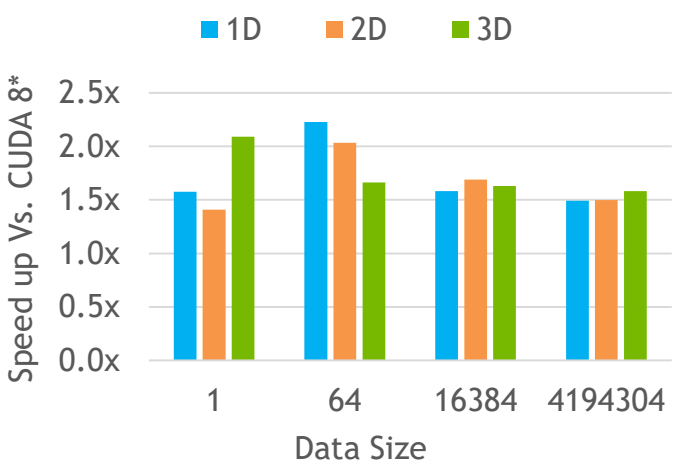
Tensorコア

- 3種類のアルゴリズムから選択可能
  - CUBLAS_GEMM_ALGO[0:2]_TENSOR_OP
- CUBLAS_GEMM_DFALT_TENSOR_OP: 自動選択

CUFFT, NPP

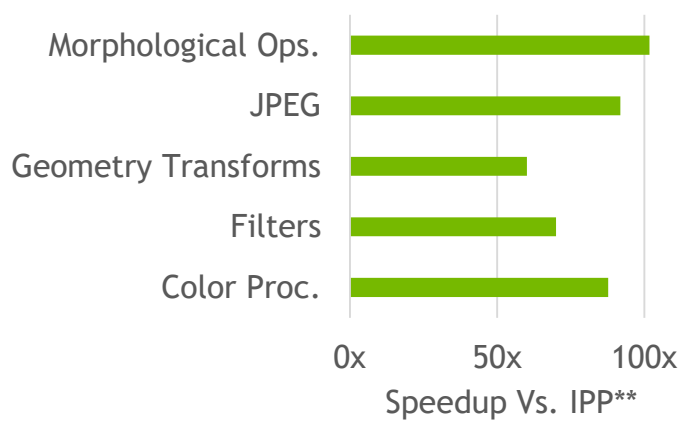
cuFFT

CUDA 8と比べて最大2倍の高速化
1D, 2D, 3D



NPP

IPPと比べて最大100倍の性能
イメージプロセッシング、コンピュータビジョン



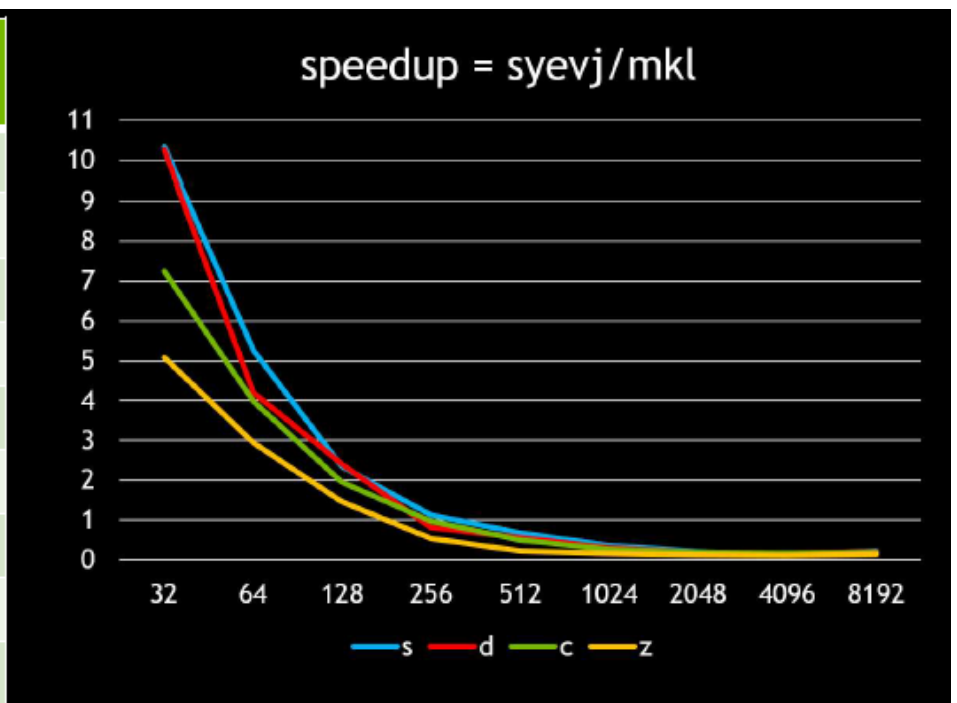
* V100 and CUDA 9 (r384); Intel Xeon Broadwell, dual socket, E5-2698 v4@ 2.6GHz, 3.5GHz Turbo with Ubuntu 14.04.5 x86_64 with 128GB System Memory
* P100 and CUDA 8 (r361); For cublas CUDA 8 (r361): Intel Xeon Haswell, single-socket, 16-core E5-2698 v3@ 2.3GHz, 3.6GHz Turbo with CentOS 7.2 x86-64 with 128GB System Memory
** CPU system running IPP: Intel Xeon Haswell single-socket 16-core E5-2698 v3@ 2.3GHz, 3.6GHz Turbo Ubuntu 14.04.5 x86_64 with 128GB System Memory

cuSOLVER: ヤコビ法ベースの固有値ソルバー

QR法と比べて計算量は増えるが並列性が高い

- 行列サイズ128~256まではMKLより高速

n	QR	Jacobi	MKL
32	0.008	0.04	0.004
64	0.03	0.15	0.04
128	0.11	0.46	0.19
256	0.50	1.05	1.30
512	1.42	2.88	5.19
1024	3.00	4.56	15.26
2048	5.56	5.96	32.66
4096	5.93	7.35	43.66
8192	4.85	9.80	48.35

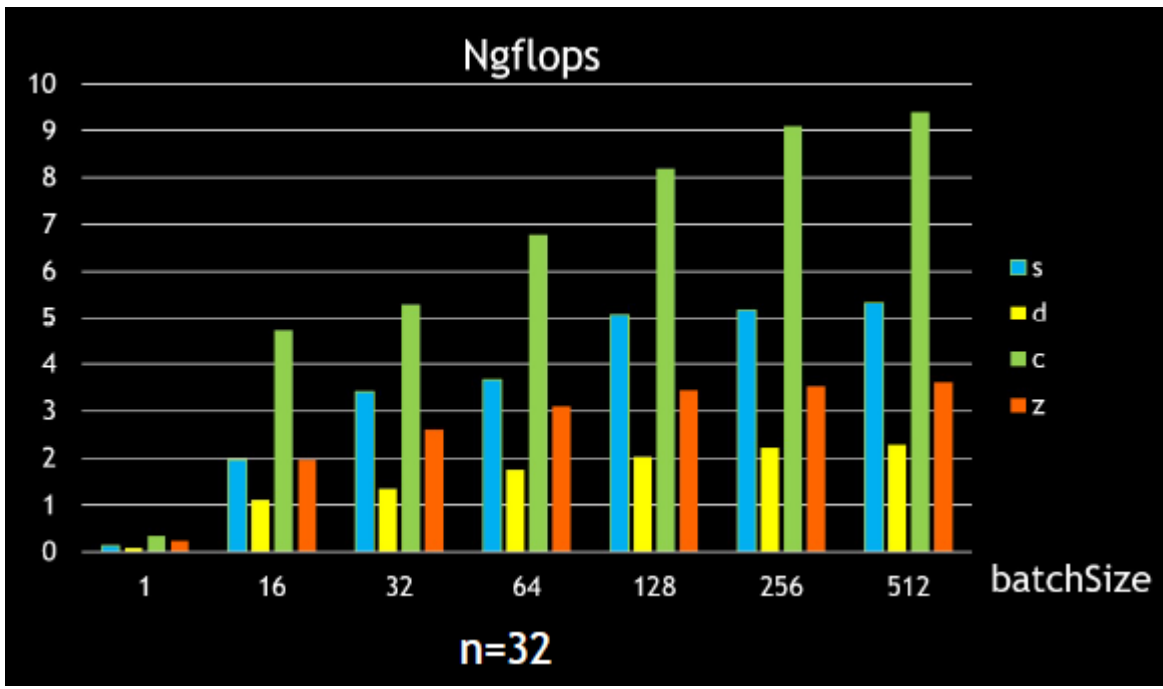


CPU:E5-2690v2, GPU:K40

cuSOLVER: ヤコビ法ベースの固有値ソルバー

QR法と比べて計算量は増えるが並列性が高い

- バッチ実行 (各行列のサイズ:32x32)



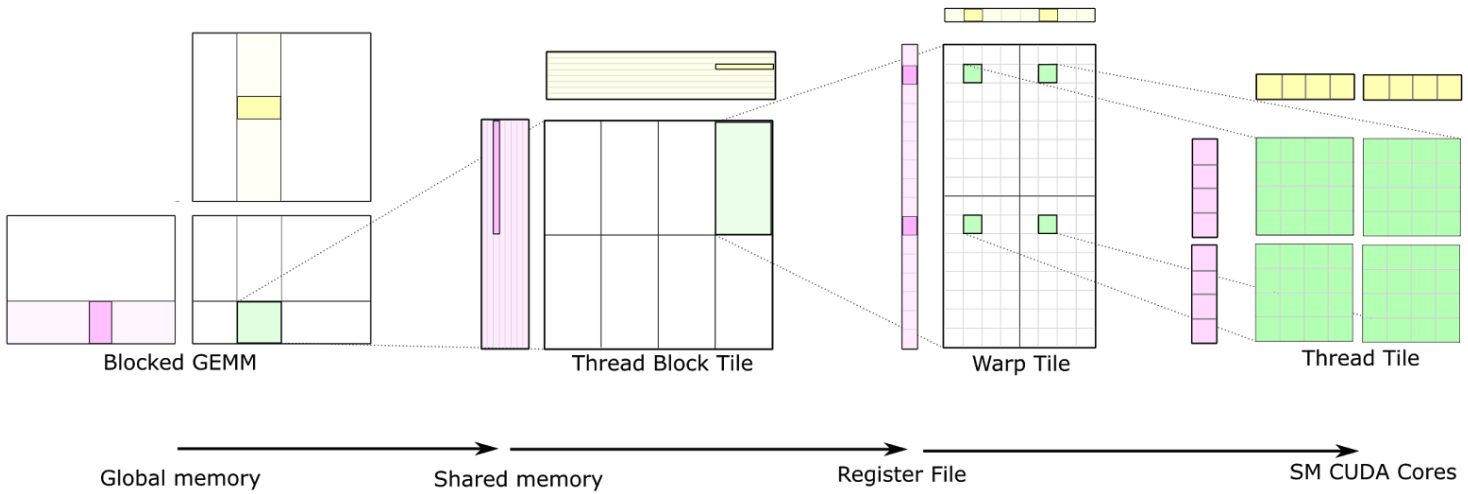
MKL, 16 threads

Data type	MKL Ngflops	Speedup
S	0.73	7.3
D	1.59	1.4
C	3.13	3.0
Z	4.55	0.8

CPU:E5-2690v2, GPU:K40

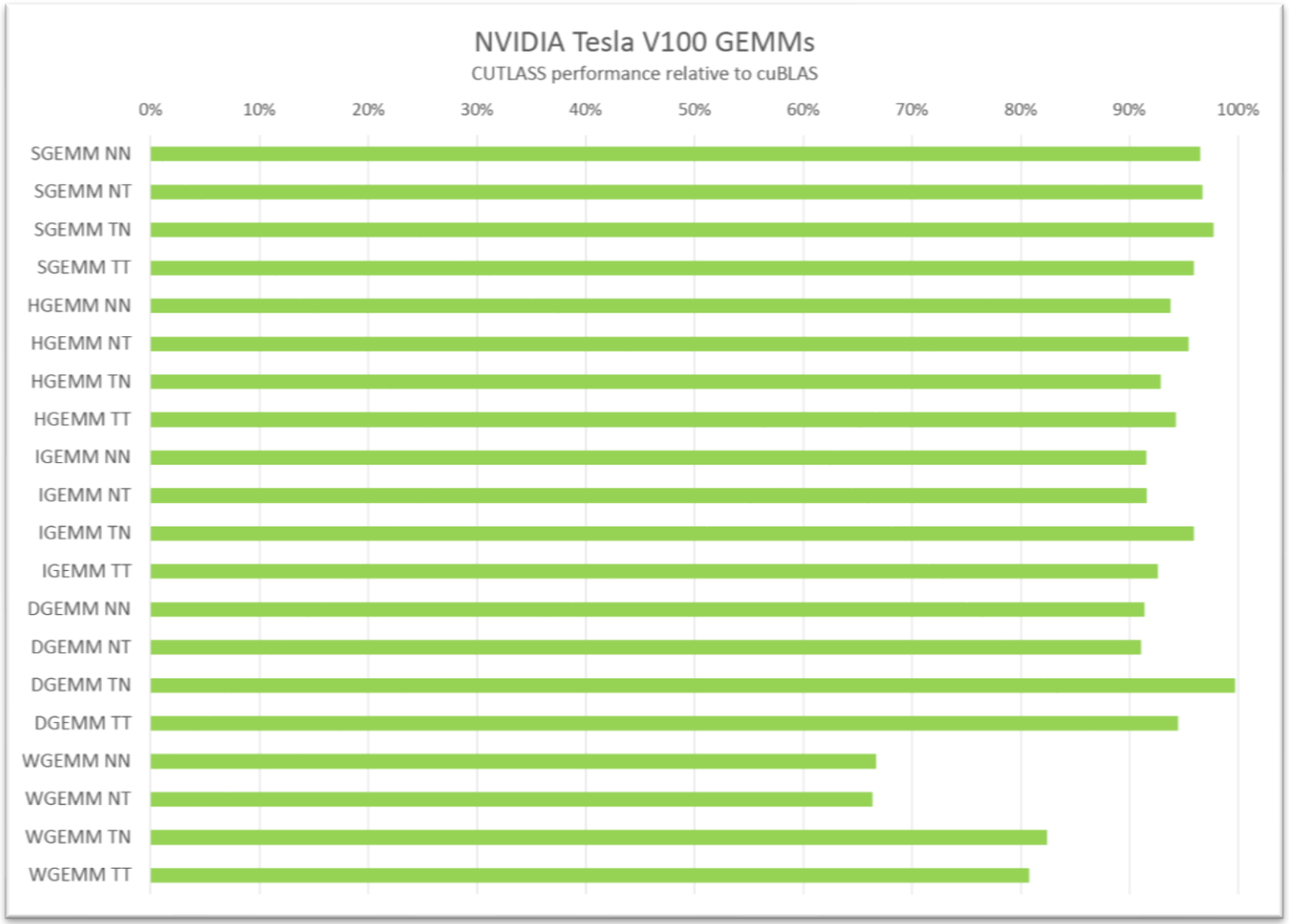
CUTLASS: FAST LINER ALGEBRA IN CUDA C++

https://github.com/NVIDIA/cutlass (version 0.1)



- ユーザのCUDAカーネルから使用できる、高性能な行列積C++テンプレート
  - DLアプリの多くは、行列積の組み合わせ
- CUDAの様々な階層で利用可能
  - デバイスレベル、ブロックレベル、ワープレベル、スレッドレベル

CUTLASSの性能 (対CUBLAS)



cuBLASと遜色ない性能を、CUDA C++レベルで実現

- データ型: FP16, FP32, FP64, INT
- Tensorコア対応
- 行列データレイアウト: NN, NT, TN, TT

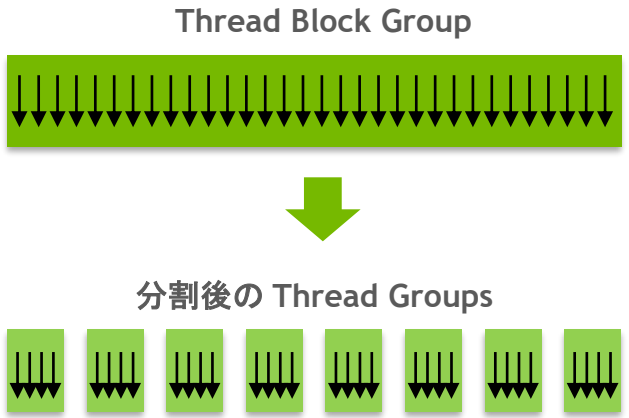
COOPERATIVE GROUPS

COOPERATIVE GROUPS

スケーラブルで柔軟性の高い、スレッド間同期・通信機構

協調動作するスレッドグループの、定義・分割・同期を容易にする

- スケーラブルなグループサイズ: 数スレッド～全スレッド
- 動的なグループの生成・分割が可能
- CUDAとしてサポート
- グループサイズにより適切なハードウェアを選択
- Kepler世代以後のGPUで利用可能



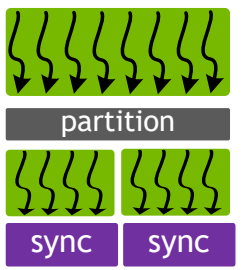
* Note: Multi-Block and Mult-Device Cooperative Groups are only supported on Pascal and above GPUs

多様なスレッド間同期を簡単に

3つのスケール

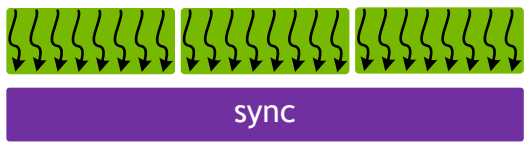
スレッドブロック内

協調動作するスレッドグループを動的に生成し、各グループで同期

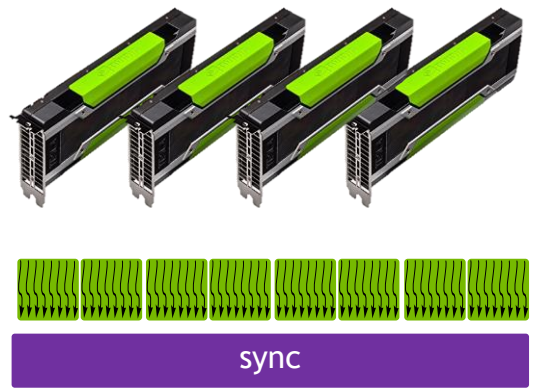


シングルGPU内 (SM間の同期)

スレッドブロック間の同期



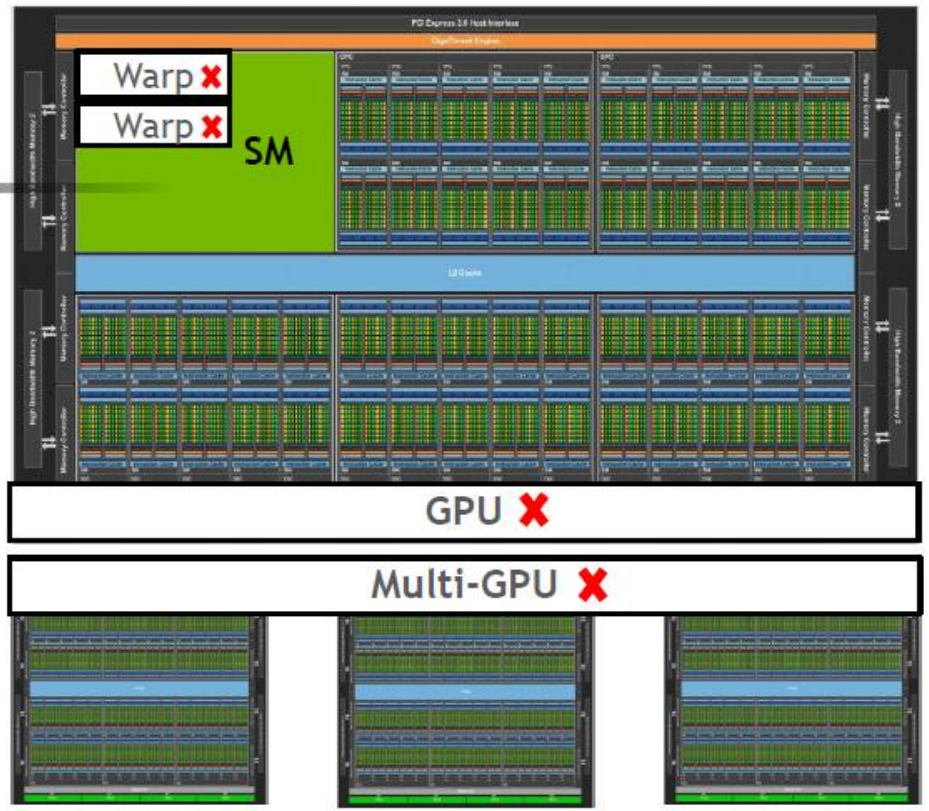
マルチGPU間 (GPU間の同期)



カーネル内でのスレッド同期

CUDA 8 まで

```
__syncthreads(): block level
synchronization barrier in CUDA
```



カーネル内でのスレッド同期

CUDA 9 から

小さいグループ

```
For current coalesced set of threads:
  auto g = coalesced_threads();
For warp-sized group of threads:
 auto block = this_thread_block();
 auto g = tiled_partition<32>(block)
```

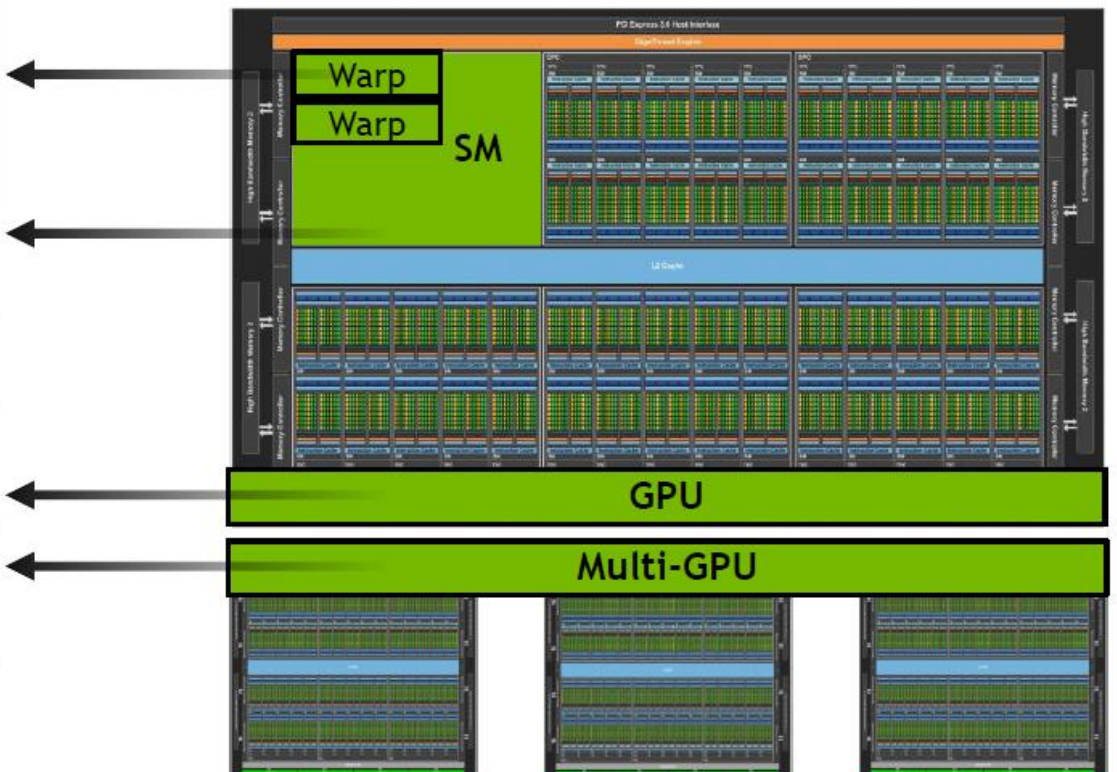
```
For CUDA thread blocks:
  auto g = this_thread_block();
```

スレッドブロック

```
For device-spanning grid:
  auto g = this_grid();
```

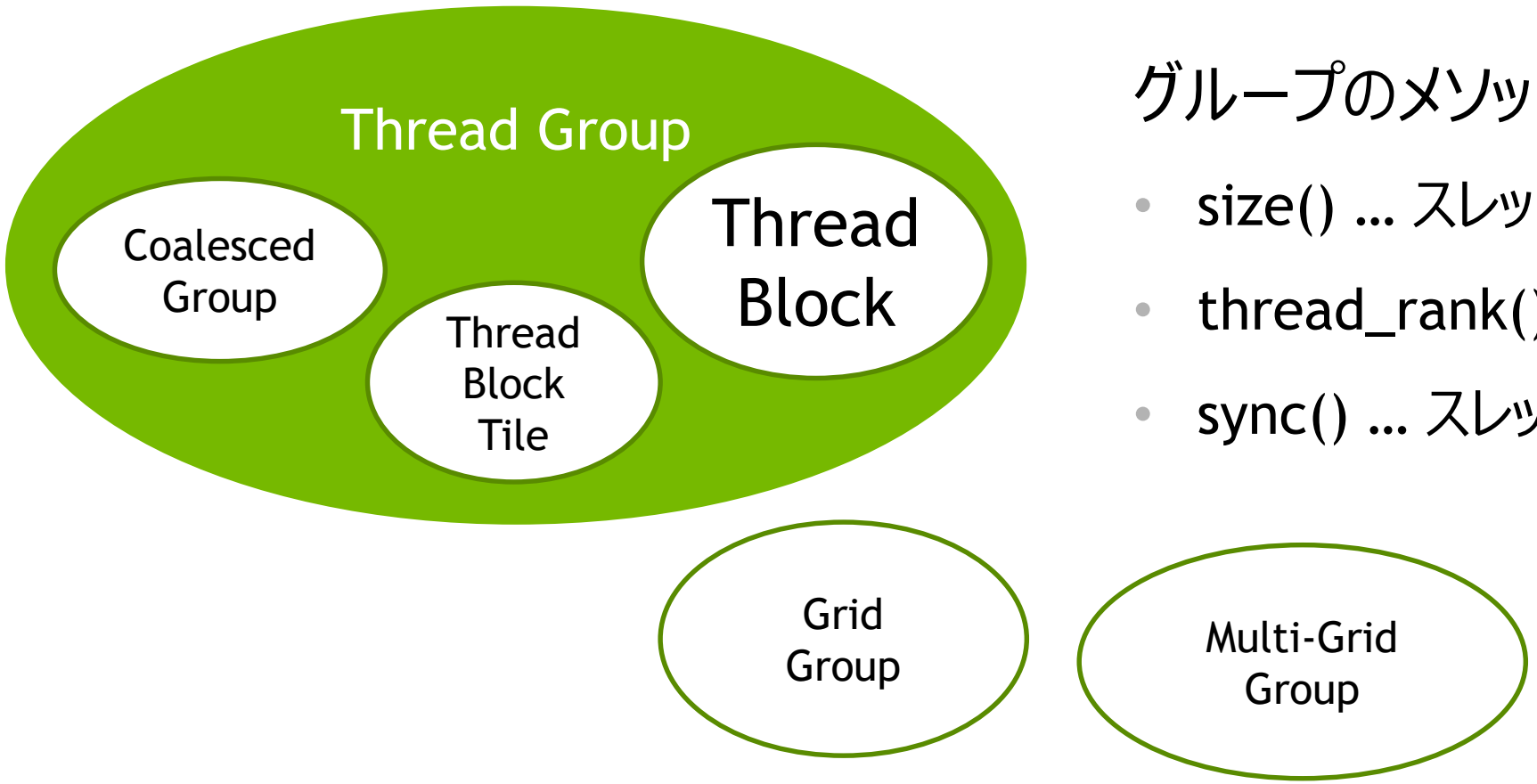
```
For multiple grids spanning GPUs:
  auto g = this_multi_grid();
```

大きいグループ



COOPERATIVEグループ

5種類のグループ



グループのメソッド

- size() … スレッド数
- thread_rank() … スレッドのID
- sync() … スレッド間同期

COOPERATIVEグループ

Thread Blockから、Thread Block Tile(サブグループ)を生成

```
thread_group block = this_thread_block();
```

this_thread_block()は、自Thread Blockに対応

```
block.sync();
```

__synchthreads()と等価

```
thread_group tile32 = tiled_partition(block, 32);
```

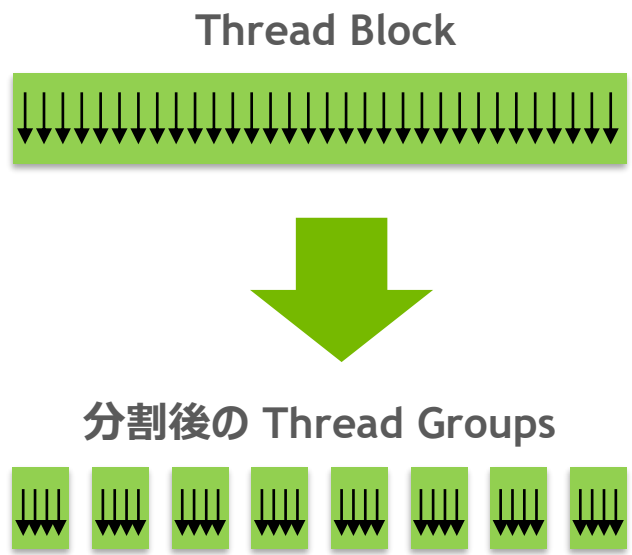
Thread Blockから、32スレッドのグループを作成

```
tile32.sync();
```

サブグループ内の32スレッド間で、同期

```
thread_group tile4 = tiled_partition(tile32, 4);
```

再帰的なサブブループ作成も可能



(*) Tileサイズは32以下、かつ、
2^Nに制限 (CUDA 9.0)

COOPERATIVEグループ

同じデバイス関数を、サイズの異なるグループで共用できる

Thread Block (～1024スレッド)

```
g = this_thread_block();
val = reduce(g, shmem, myVal);
```

Warp (32スレッド)

```
g = tiled_partition(this_thread_block(), 32);
val = reduce(g, shmem, myVal);
```

```
__device__ int reduce(thread_group g, int *shmem, int val) {
  int myRank = g.thread_rank();
  for (int i = g.size()/2; i > 0; i /= 2) {
    shmem[myRank] = val;        g.sync();
    val += shmem[myRank ^ i];   g.sync();
  }
  return val;
}
```

並列reduction
(共有メモリ使用)

THREAD BLOCK TILE

- ワープ内スレッド間通信Build-in関数を使える

```
.shfl()
.shfl_down()
.shfl_up()
.shfl_xor()
.any()
.all()
.ballot()
.match_any()
.match_all()
```

```
template <unsigned size>
__device__ int reduce(thread_block_tile<size> g, int val) {
  for (int i = g.size()/2; i > 0; i /= 2) {
    val += g.shfl_xor(val, i);
  }
  return val;
}
```

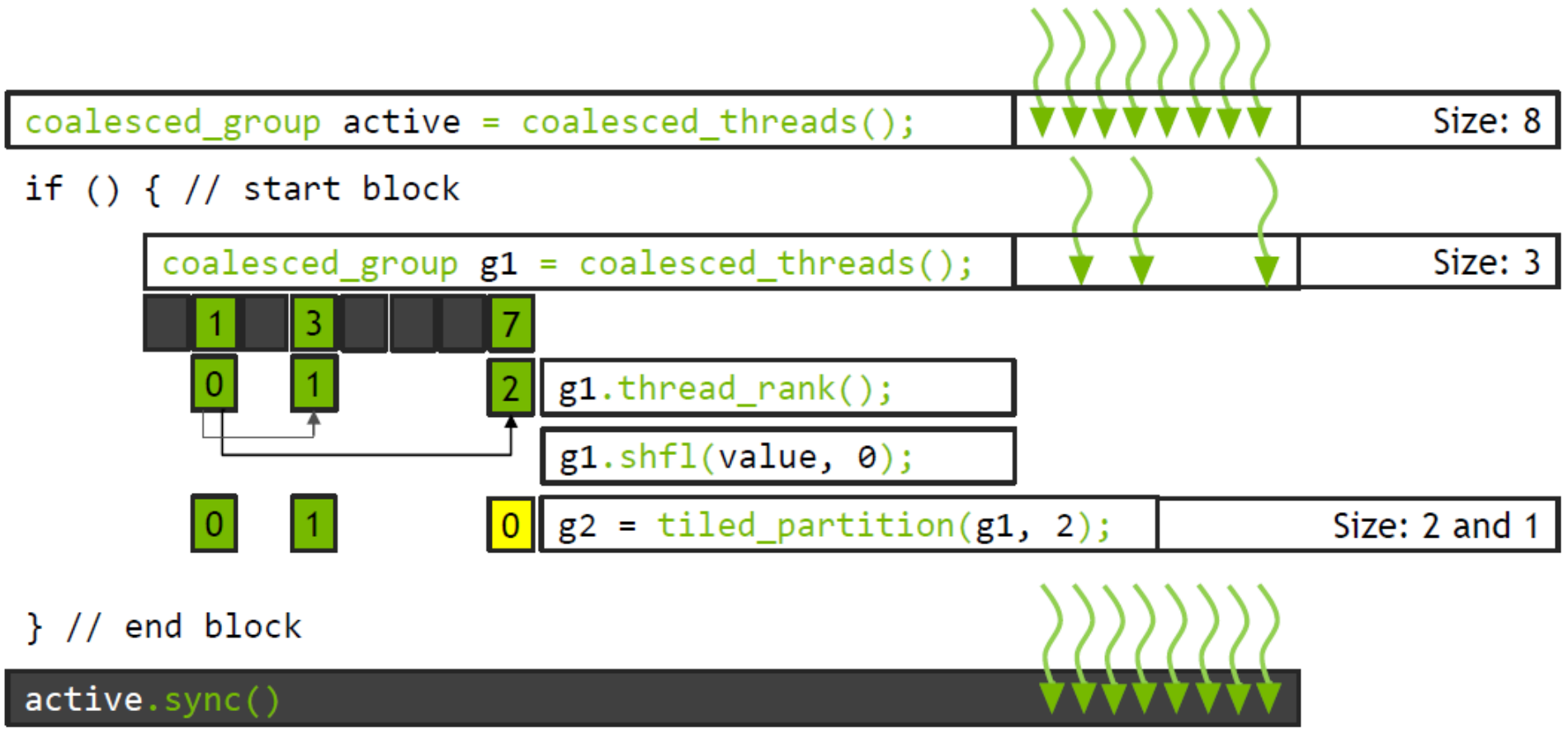
並列reduction
(shfl_xor使用)

- コンパイル時にサイズが分かると高速

```
thread_group_tile<32> tile32 = tiled_partition<32>(this_thread_block());
thread_group_tile<4> tile4 = tiled_partition<4>(this_thread_block());
```

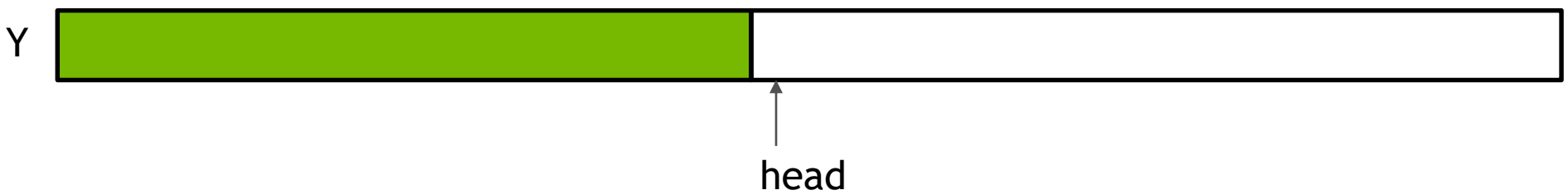
COALESCED GROUP

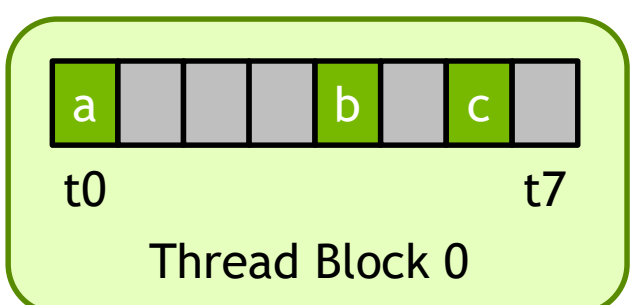
同時に同じパスを実行しているスレッドのグループ

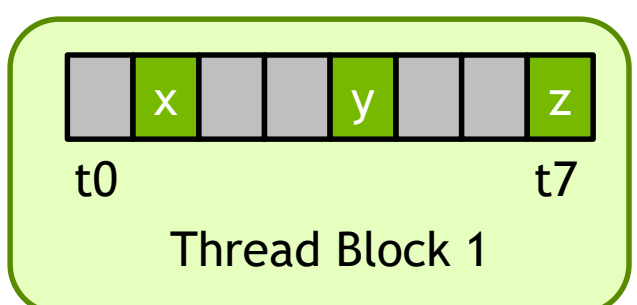


COALESCED GROUP

並列 Array Push (サイズ不定)

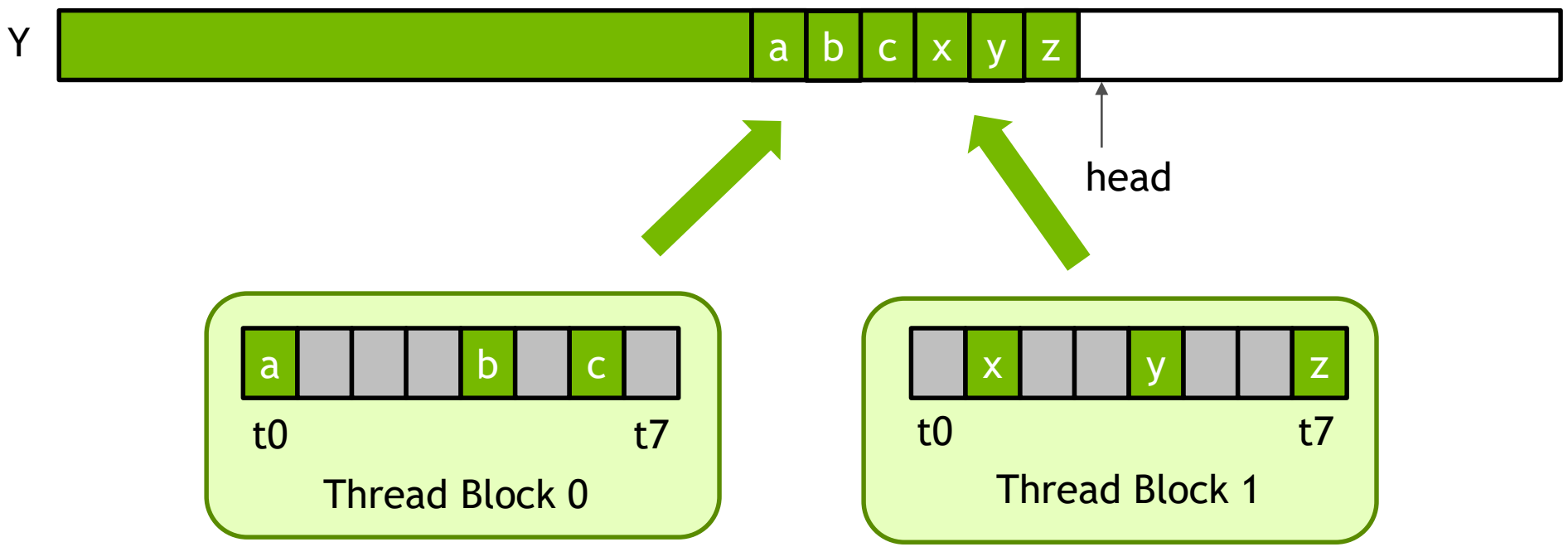






COALESCED GROUP

並列 Array Push (サイズ不定)



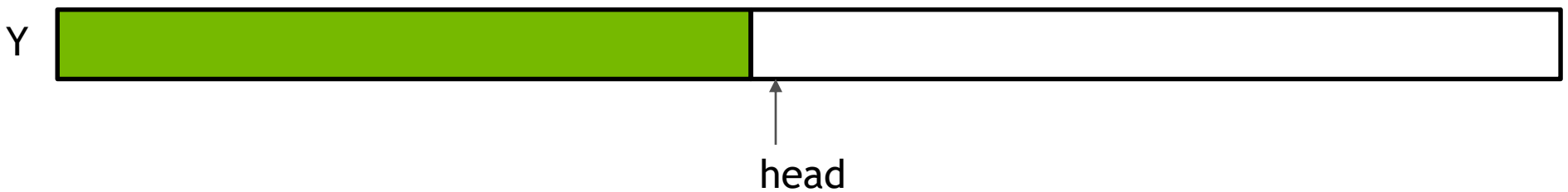
COALESCED GROUP

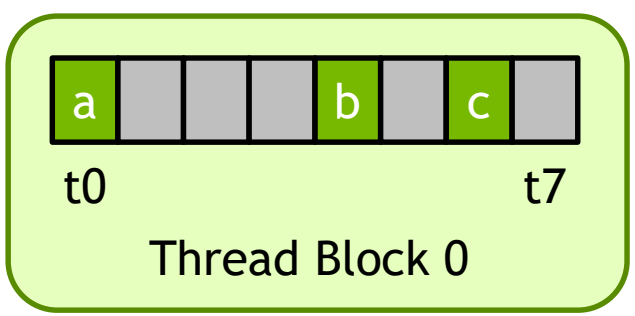
Atomic Aggregation

```
__device__ int atomicAggInc(int *head_ptr)
{
    coalesced_group g = coalesced_threads();
    int old_head;
    if (g.thread_rank() == 0) {
        old_head = atomicAdd(head_ptr, g.size())
    }
    int my_head = g.shfl(old_head, 0) + g.thread_rank();
    return my_head;
}
```

COALESCED GROUP

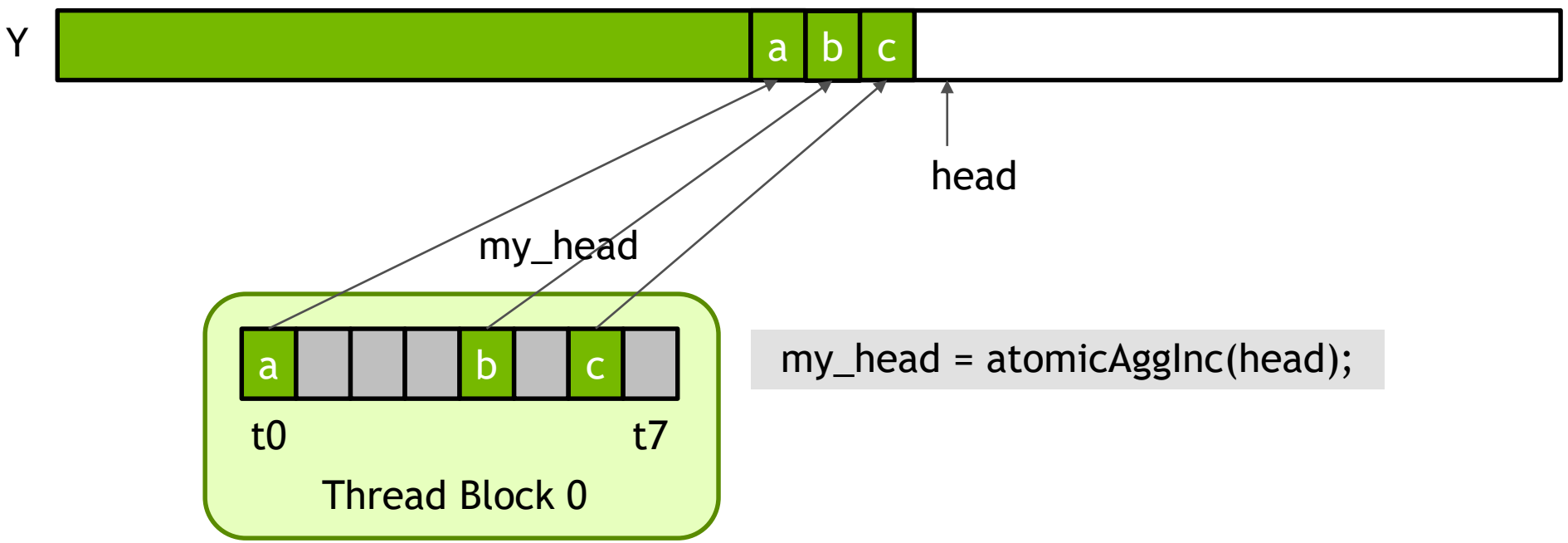
並列 Array Push (サイズ不定)





COALESCED GROUP

並列 Array Push (サイズ不定)




```
my_head = atomicAggInc(head);
```

ATOMIC AGGREGATION

Build-In関数でも実装は可能

Cooperative Groups

```
coalesced_group g = coalesced_threads();


int ret;
if (g.thread_rank() == 0) {
    ret = atomicAdd(ptr, g.size())
}
ret = g.shfl(ret, 0);
return ret + g. thread_rank();
```

Build-In Functions

```
int mask = __activemask();
int rank = __popc(mask & __lanemask_lt());
int leader_lane = __ffs(mask) – 1;
int ret;
If (rank == 0) {
    ret = atomicAdd(p, __popc(mask));
}
ret = __shfl_sync(mask, ret, leader_lane);
return ret + rank;
```

記述しやすいのは、どちらか?

GRID GROUP

グリッド(シングルGPU)内の、全スレッドのグループ

```
__global__ kernel() {
    grid_group grid = this_grid();
    while (...) {
        ...
        grid.sync();
    }
}
```

専用APIでカーネル起動

```
cudaLaunchCooperativeKernel(...);
```

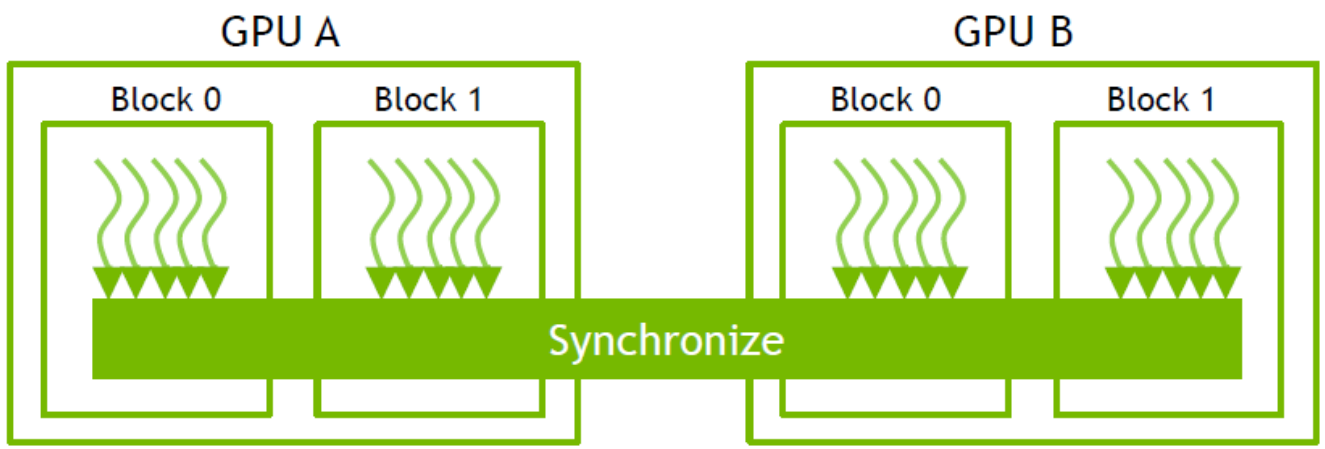
起動したカーネルの全スレッドが、同時にactiveになる必要あり (Persistent Kernel)

```
cudaOccupancyMaxActiveBlocksPerMultiprocessor(&numBlocksPerSm, kernel, numThreads, 0);
```

MULTI GRID GROUP

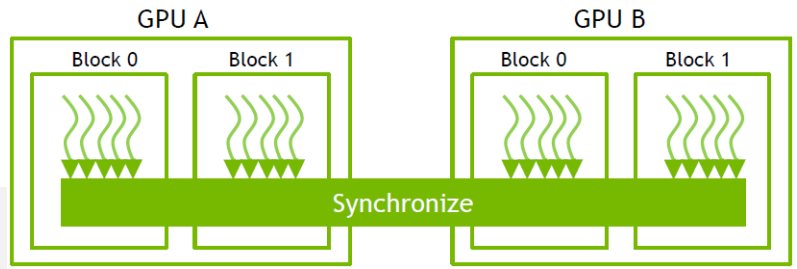
マルチグリッド(マルチGPU)内の、全スレッドのグループ

```
__global__ kernel() {
    multi_grid_group multi_grid = this_multi_grid();
    while (...) {
        ...
        grid.sync();
    }
}
```



MULTI GRID GROUP

マルチグリッド(マルチGPU)内の、全スレッドのグループ



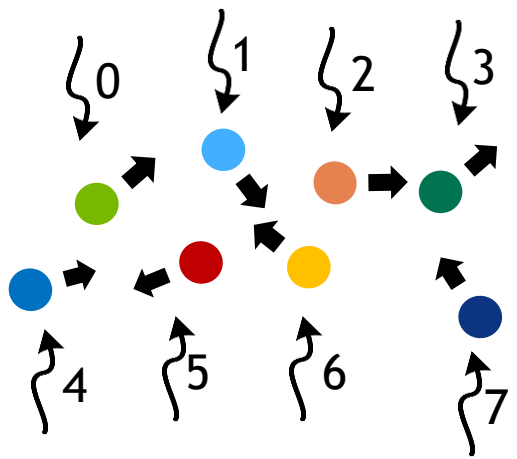
専用APIでカーネル起動

```
struct cudaLaunchParams params[numDevices];
for (int i = 0; i < numDevices; i++) {
   params[i].func = (void*) kernel;
   params[i].gridDim = dim3(...);
   params[i].blockDim = dim3(...);
   params[i].sharedMem = ...;
   params[i].stream = ...;
   params[i].args = ...;
}
cudaLaunchCooperativeKernelMultiDevice(params, numDevices);
```

例: 粒子シミュレーション

Cooperative Groups無し

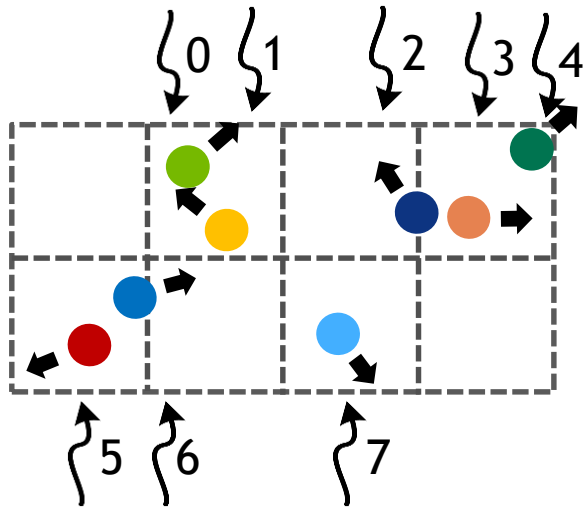
```
// threads update particles in parallel
integrate<<<blocks, threads, 0, stream>>>(particles);
```



例: 粒子シミュレーション

Cooperative Groups無し

```
// threads update particles in parallel
integrate<<<blocks, threads, 0, s>>>(particles);



// Collide each particle with others in neighborhood
collide<<<blocks, threads, 0, s>>>(particles);
```

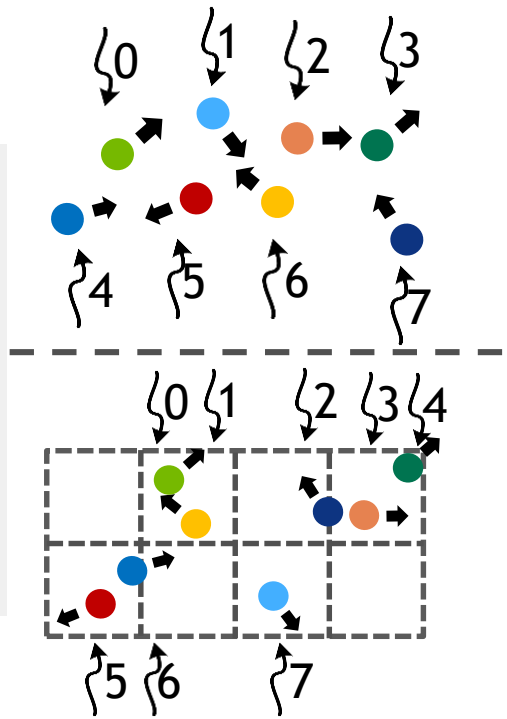


(*) 粒子の位置が移動したら、CUDAスレッドへの粒子のマッピングを変えたほうが、高速に処理できる

例: 粒子シミュレーション

Cooperative Groups無し

```
// threads update particles in parallel
integrate<<<blocks, threads, 0, s>>>(particles);

// ここで暗黙的に同期しているので、マッピング変更が可能

// Collide each particle with others in neighborhood
collide<<<blocks, threads, 0, s>>>(particles);
```

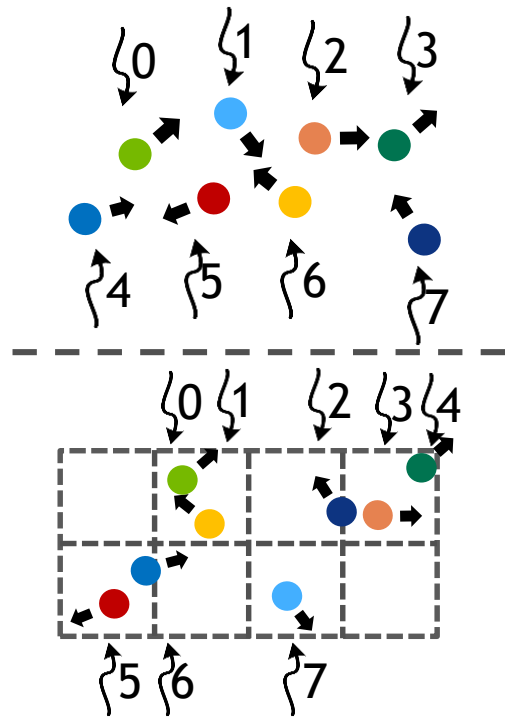


(*) 粒子の位置が移動したら、CUDAスレッドへの粒子のマッピングを変えたほうが、高速に処理できる

GRID GROUPで粒子シミュレーション

2種類の処理を、シングルカーネルで実行

```
__global__ void particleSim(Particle *p, int N) {

  grid_group g = this_grid();

  for (i = g.thread_rank(); i < N; i += g.size())
    integrate(p[i]);

  g.sync() // GPU全体の同期

  for (i = g.thread_rank(); i < N; i += g.size())
    collide(p[i], p, N);
}
```

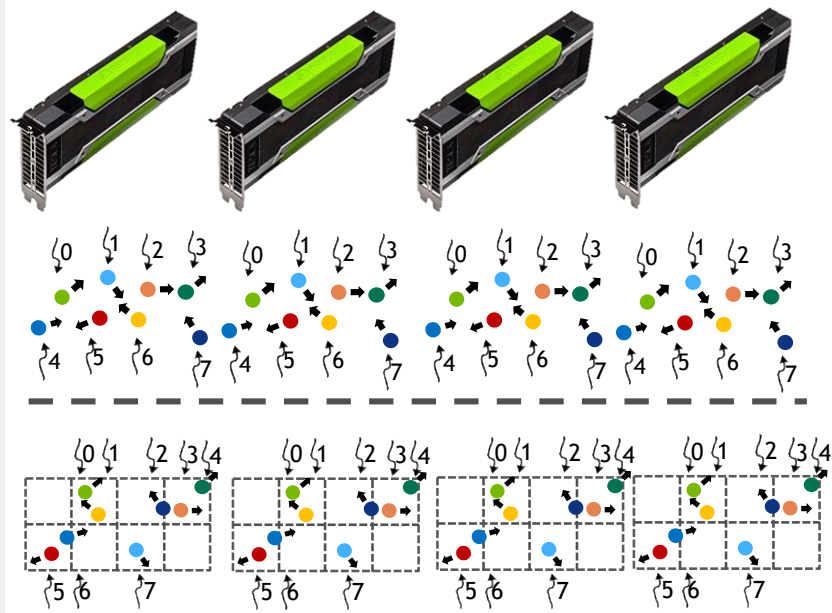


cudaLaunchCooperativeKernel(…)でカーネル起動

MULTI-GRID GROUPで粒子シミュレーション

シングルカーネルで、大規模な問題をマルチGPU実行

```
__global__ void particleSim(Particle *p, int N) {

  multi_grid_group g = this_multi_grid();

  for (i = g.thread_rank(); i < N; i += g.size())
    integrate(p[i]);

  g.sync() // マルチGPUの全てで同期

  for (i = g.thread_rank(); i < N; i += g.size())
    collide(p[i], p, N);
}
```



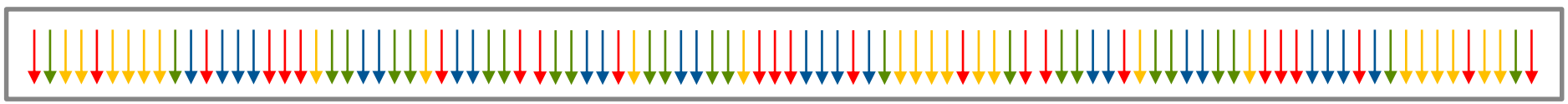
cudaLaunchCooperativeKernelMultiDevice(…)で起動

ロードマップ: COOPERATIVE GROUPS

より柔軟なグループ作成

任意ラベルによる、グループの分割 (Volta限定)

```
// 計算結果が同じスレッドのグループ
int label = foo() % 4;
thread_group block = partition(this_thread_block(), label);
```



(*) ランダムなグループは、SIMT実行効率が低下するので、注意が必要

32より大きなタイル

```
thread_group g = tiled_partition(this_thread_block(), 64);
```

ロードマップ: COOPERATIVE GROUPS

Collectiveアルゴリズムのライブラリ

Reductions, sorting, prefix sum (scan), 等など.

```
// collective key-value sort using all threads in the block
cooperative_groups::sort(this_thread_block(), myValues, myKeys);
```

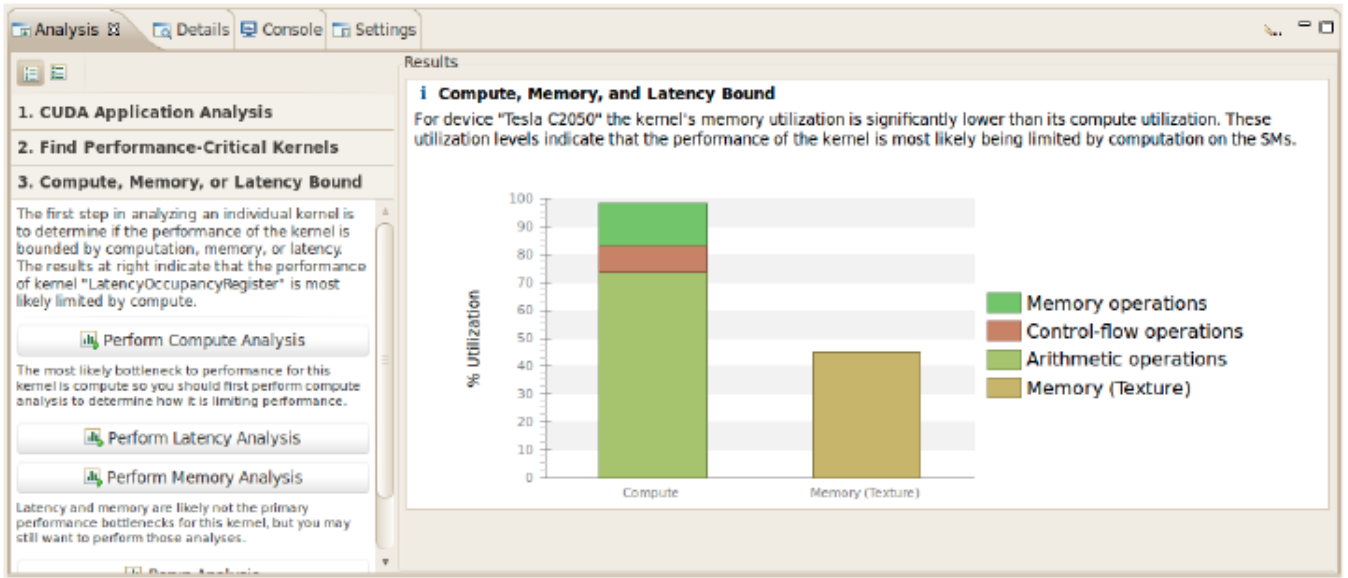
```
// collective scan-based allocate across block
int sz = myAllocationSize(); // amount each thread wants
int offset = cooperative_groups::exclusive_scan(this_thread_block(), sz);
```

開発ツール

多様な開発ツール

VISUAL PROFILER

- Trace CUDA activities
- Profile CUDA kernels
- Correlate performance instrumentation with source code
- Expert-guided performance analysis



NVPROF

- Collect Performance events and metrics

GPU LIBRARY ADVISOR

- Detect CUDA library optimization opportunities

NVDISASM, CUOBJDUMP

CUDA-MEMCHECK

- Detect out-of-bounds memory accesses
- Detect race condition in memory accesses
- Detect uninitialized variable accesses
- Detect incorrect GPU thread synchronization

CUDA-GDB

- Debug CUDA kernels with CLI
- Debug CPU and GPU code
- CPU and GPU core dump support

CUDA-MEMCHECK

Cooperative Groups対応

安全ではないWarp同期プログラミングの検出 (racecheck)

UNSAFE CODE

```
__device__ char reduce(char val) {
  extern __shared__ char smem[];
  const int tid = threadIdx.x;

  #pragma unroll
  for(int i = warpSize/2; i > 0; i /= 2) {
    smem[tid] = val;
    val += smem[tid ^ i];
  }
  return val;
}
```

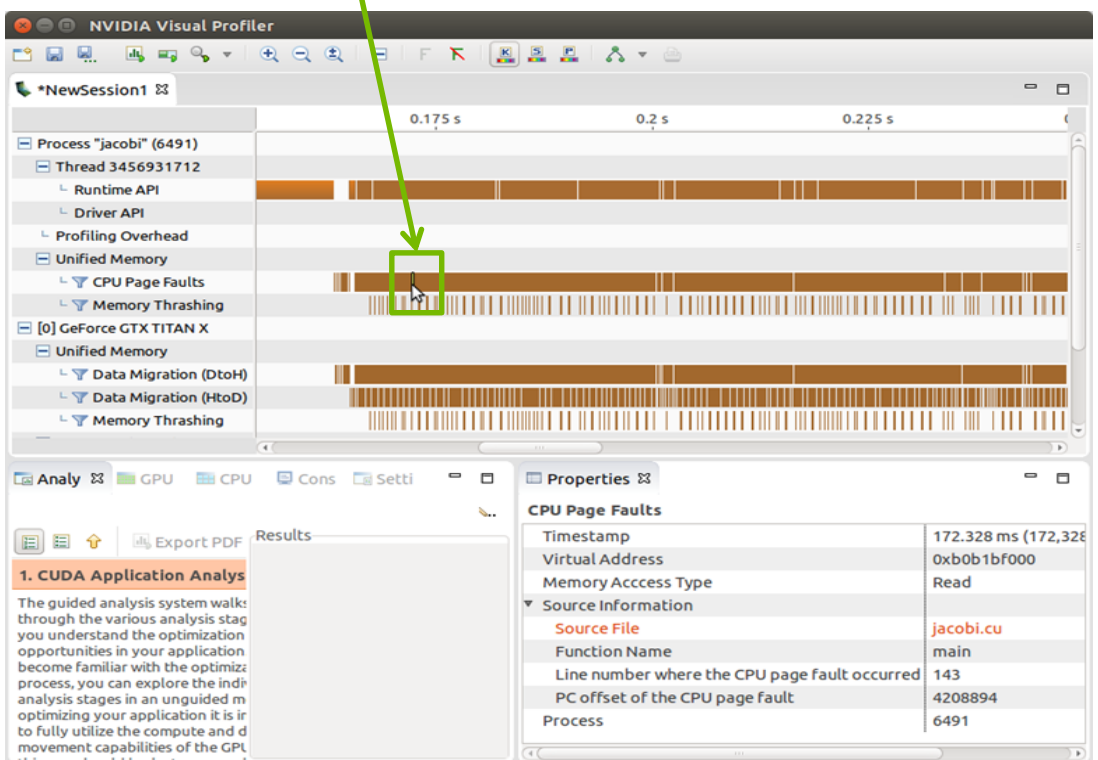
RACECHECK OUTPUT

```
$ cuda-memcheck -tool racecheck --racecheck-report hazard ./a.out

========= CUDA-MEMCHECK
========= WARN:(Warp Level Programming) Potential RAW hazard detected
at __shared__ 0xf in block (0, 0, 0) :

=========     Write Thread (15, 0, 0) at 0x00000e08 in
/home/user/reduction.cu:32:kernel(void)

=========     Read Thread (14, 0, 0) at 0x00000ef0 in
/home/user/reduction.cu:33:kernel(void)
...
```

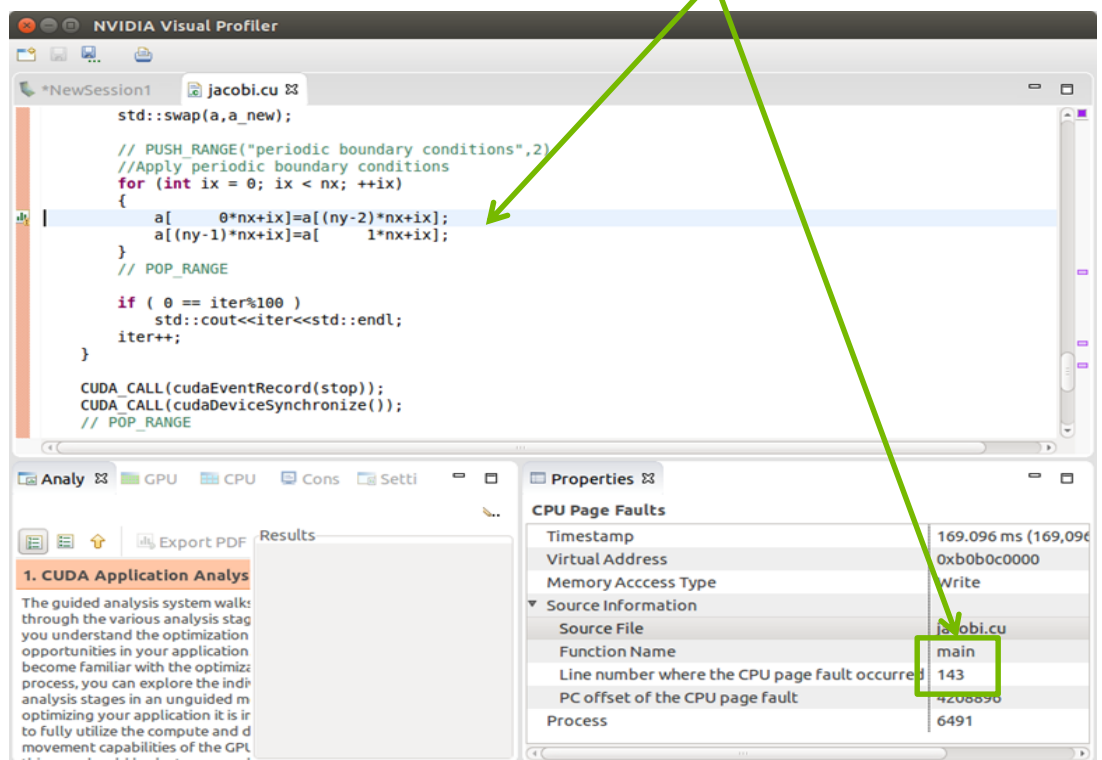
NVVP: UNIFIED MEMORY プロファイリング

CPUページフォールトの発生箇所とソースコードとの対応付け

Page Fault

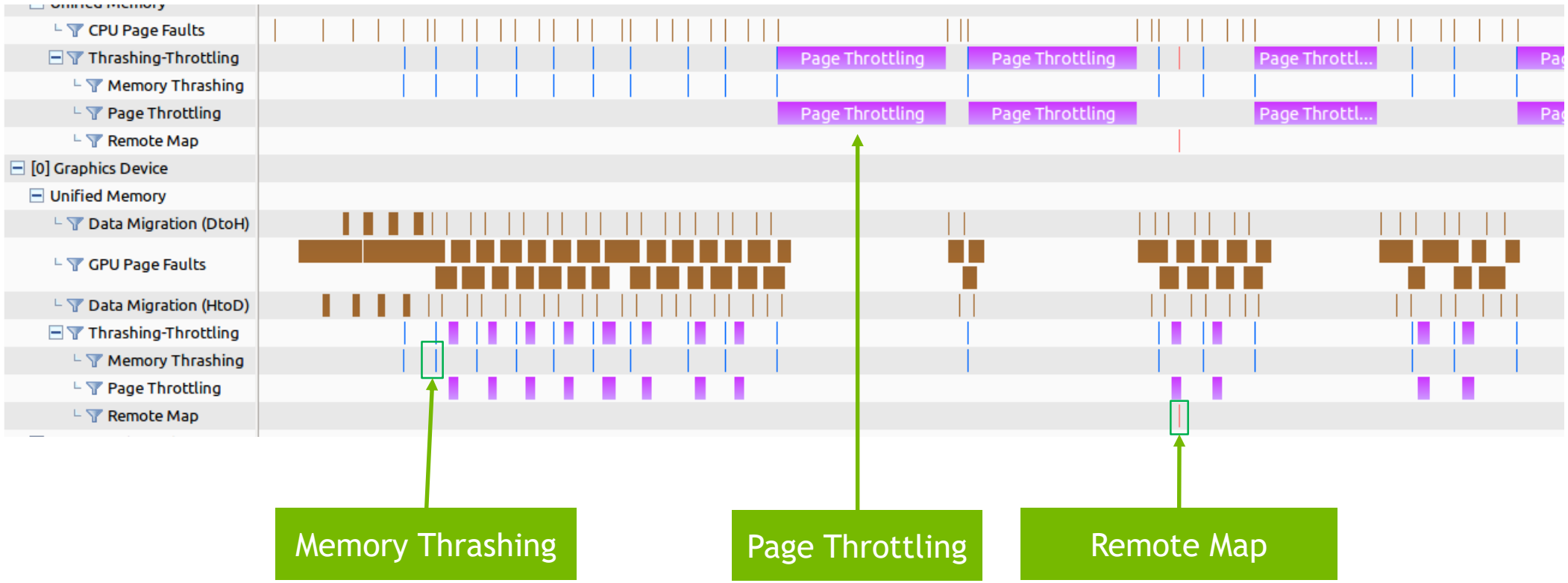


Correlation



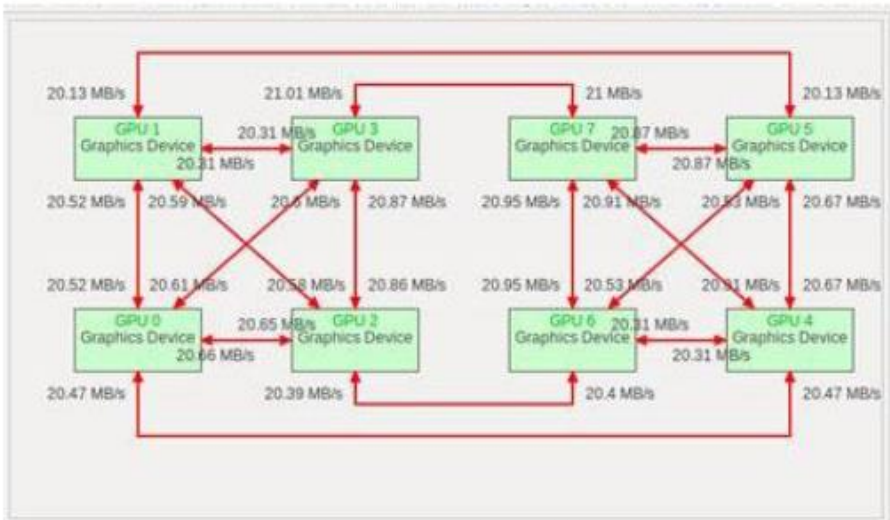
NVVP: UNIFIED MEMORY イベントの追加

仮想メモリ関連の挙動の可視化



NVVP: NVLINKトポロジー

NVLINKの各リンクの利用率



Color legend for NV Link topology links according to bandwidth utilization

- 90 - 100 %
- 80 - 90 %
- 70 - 80 %
- 60 - 70 %
- 50 - 60 %
- 40 - 50 %
- 30 - 40 %
- 20 - 30 %
- 10 - 20 %
- 0 - 10 %

Logical NVLink Properties

Logical NVLink	Peak Bandwidth	Physical NVLinks	Pee Acc
GPU0<->GPU1	40 GB/s	1	Yes
GPU0<->GPU2	40 GB/s	1	Yes
GPU0<->GPU3	40 GB/s	1	Yes
GPU0<->GPU4	40 GB/s	1	Yes
GPU1<->GPU2	40 GB/s	1	Yes
GPU1<->GPU3	40 GB/s	1	Yes
GPU1<->GPU5	40 GB/s	1	Yes
GPU2<->GPU3	40 GB/s	1	Yes
GPU2<->GPU6	40 GB/s	1	Yes
GPU3<->GPU7	40 GB/s	1	Yes
GPU4<->GPU5	40 GB/s	1	Yes
GPU4<->GPU6	40 GB/s	1	Yes
GPU4<->GPU7	40 GB/s	1	Yes
GPU5<->GPU6	40 GB/s	1	Yes
GPU5<->GPU7	40 GB/s	1	Yes
GPU6<->GPU7	40 GB/s	1	Yes

Logical NVLink Throughput

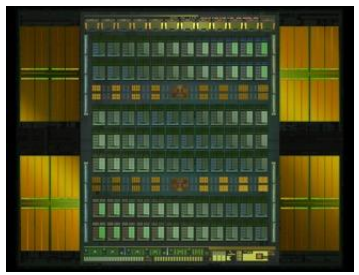
Logical NVLink	Avg Throughput	Max Th
GPU0-->GPU1	20.518 MB/s	16
GPU0<--GPU1	20.524 MB/s	16
GPU0-->GPU2	20.652 MB/s	16
GPU0<--GPU2	20.659 MB/s	16
GPU0-->GPU3	20.602 MB/s	13
GPU0<--GPU3	20.607 MB/s	13
GPU0-->GPU4	20.469 MB/s	1
GPU0<--GPU4	20.467 MB/s	15
GPU1-->GPU2	20.583 MB/s	16

DGX1

CUDA 9の概要

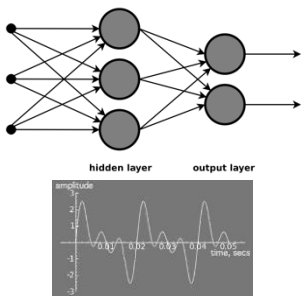
VOLTAに対応

Tesla V100
Voltaアーキテクチャ
Tensorコア
NVLink
Independentスレッドスケジューリング



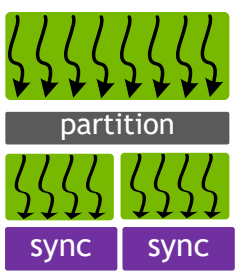
ライブラリの高速化

cuBLAS (主にDL向け)
NPP (画像処理)
cuFFT (信号処理)
cuSolver



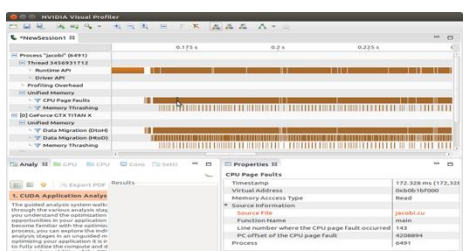
COOPERATIVE GROUPS

柔軟なスレッドグループ
並列アルゴリズムの抽象化
スレッドブロック間の同期(over SM or GPU)

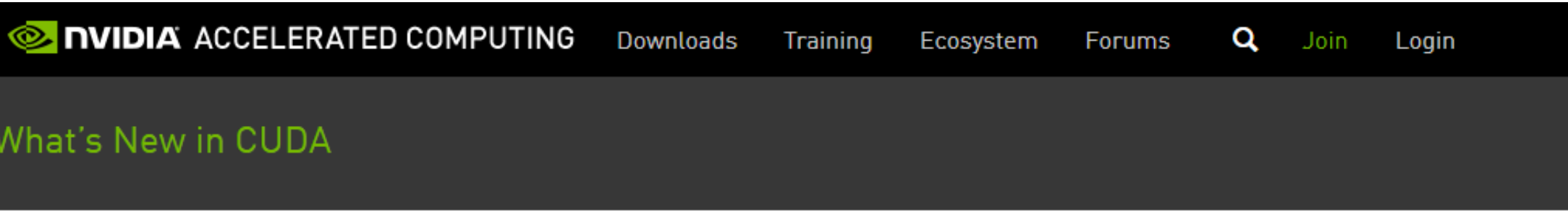


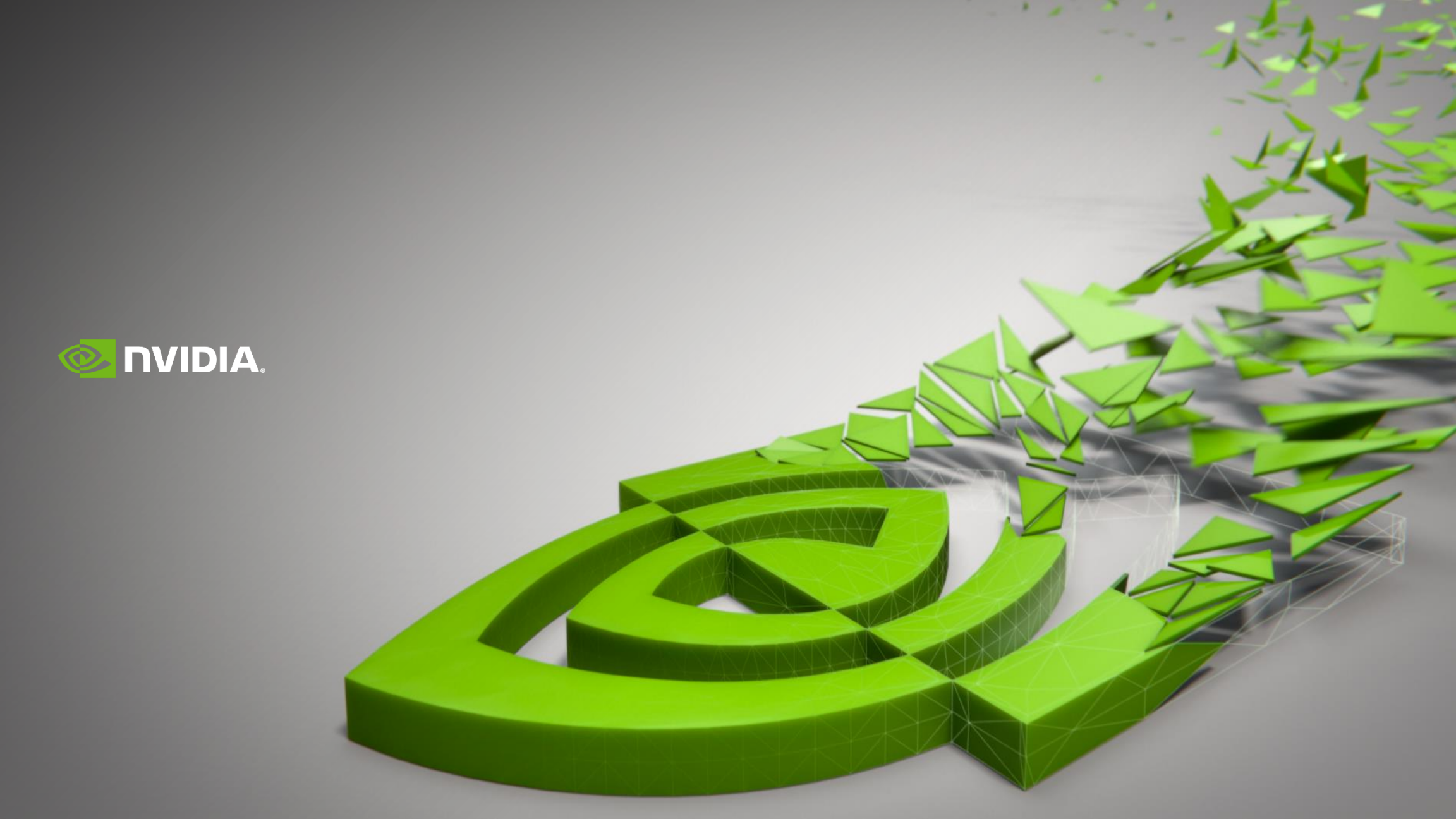
開発ツールの改善

コンパイル時間の短縮
Unified Memoryプロファイル
NVLink可視化
コンパイラサポート



CUDA 9.1


NVIDIA ACCELERATED COMPUTING
Downloads
Training
Ecosystem
Forums
Join
Login
What's New in CUDA
Home > ComputeWorks > CUDA Toolkit > What's New in CUDA
CUDA 9.1 - Coming Soon
CUDA 9.1 brings new algorithms and optimizations that speed up AI and HPC apps on Volta GPUs. With this release you can:
• Develop image augmentation algorithms for deep learning easily with new functions in NVIDIA Performance Primitives
• Run batched neural machine translations and sequence modeling operations on Volta Tensor cores using new APIs in cuBLAS
• Solve large 2D and 3D FFT problems more efficiently on multi-GPU systems with new heuristics in cuFFT
• Launch kernels up to 12x faster with new core optimizations
CUDA 9.1 also includes compiler optimizations, support for new developer tool versions and bug fixes.


NVIDIA