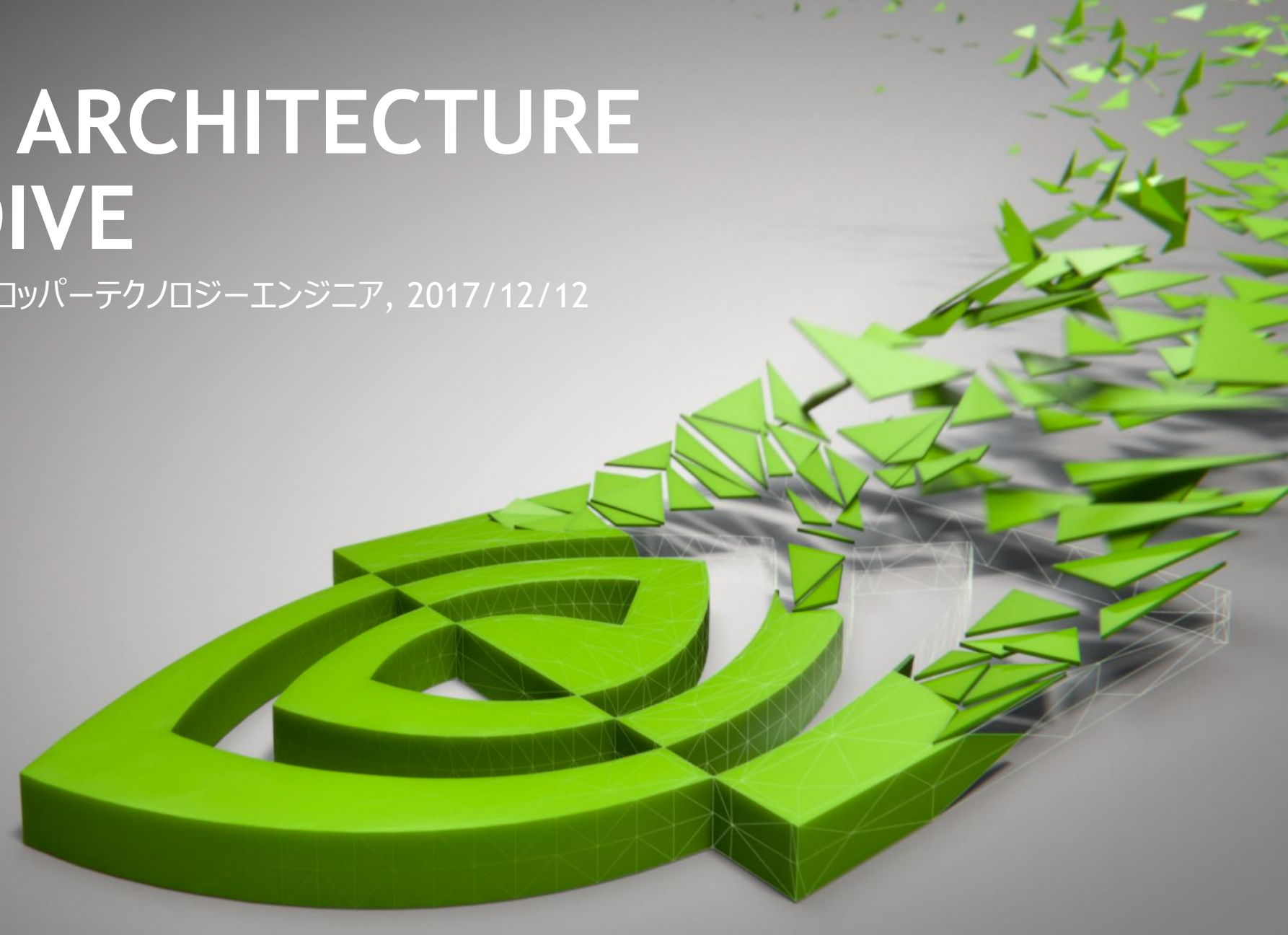
VOLTA ARCHITECTURE DEEP DIVE

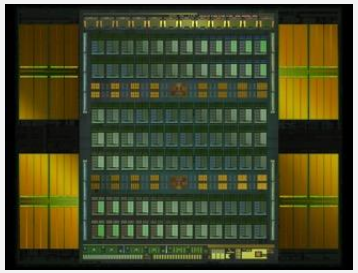
成瀬 彰, シニアデベロッパーテクノロジーエンジニア, 2017/12/12

NVIDIA



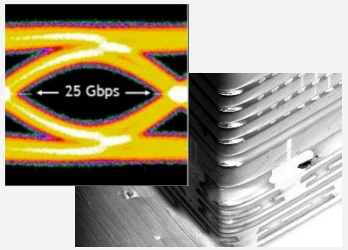
TESLA V100の概要

Volta Architecture



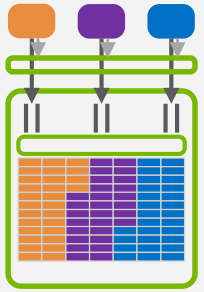
Most Productive GPU

Improved NVLink & HBM2



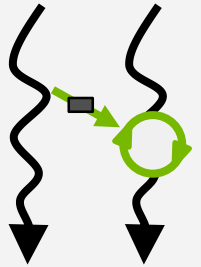
Efficient Bandwidth

Volta MPS



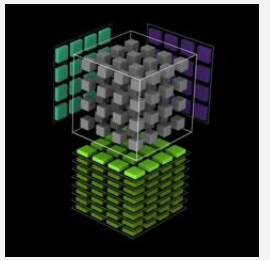
Inference Utilization

Improved SIMT Model



New Algorithms

Tensor Core

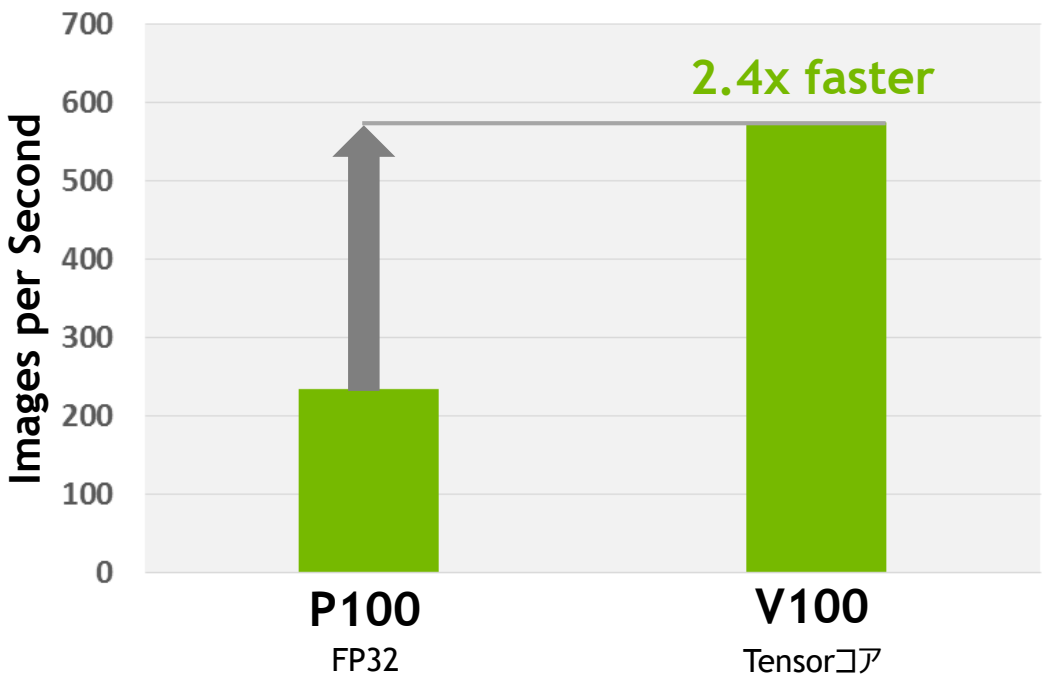


125 Programmable TFLOPS Deep Learning

Deep LearningとHPC、両方に最適なGPU

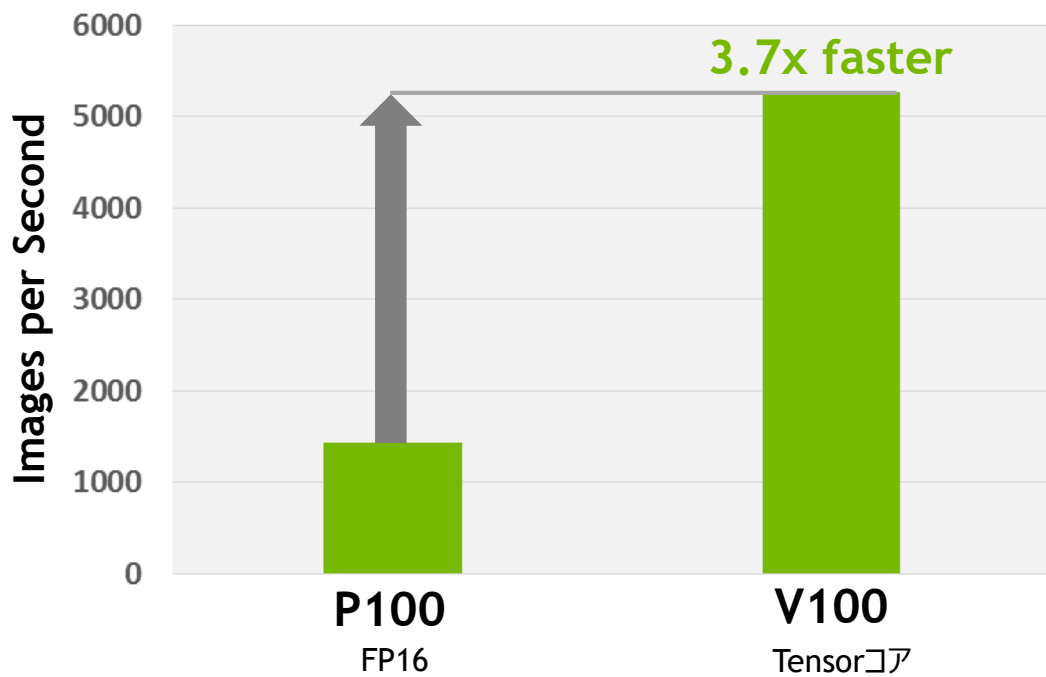
VOLTA
DL性能を大幅に向上

トレーニング



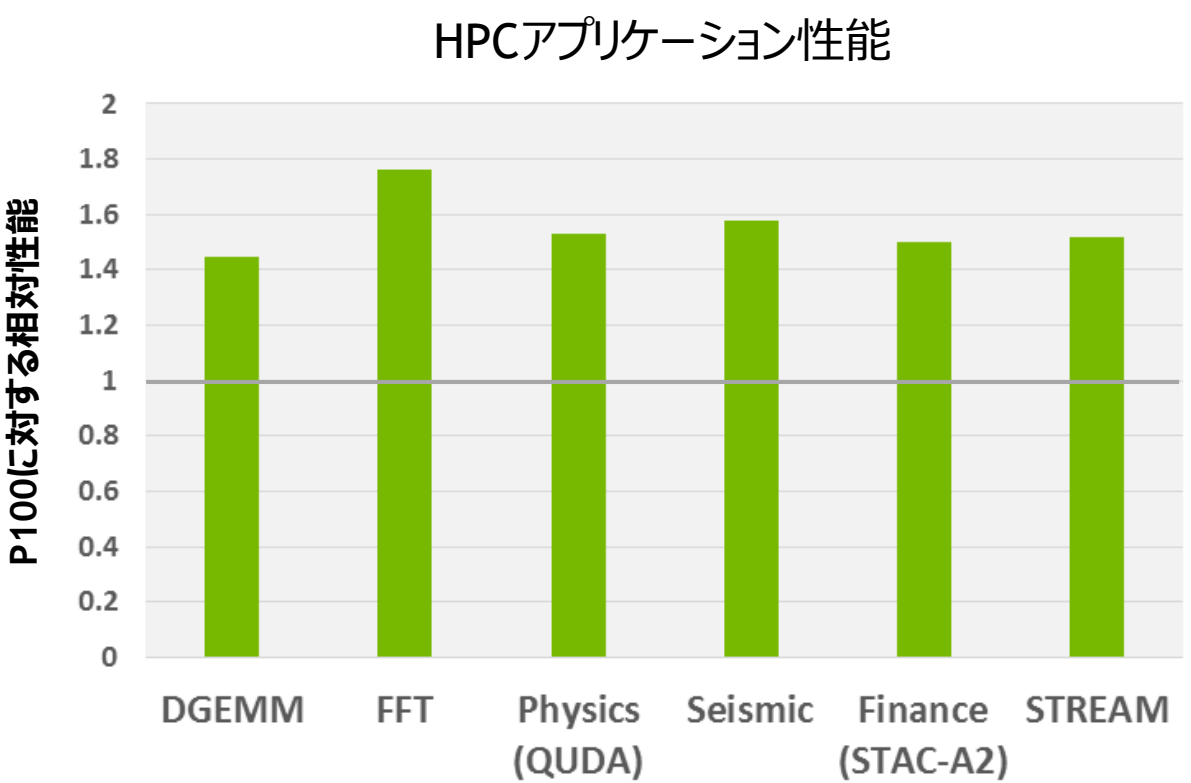
インファレンス

TensorRT - 7ms Latency

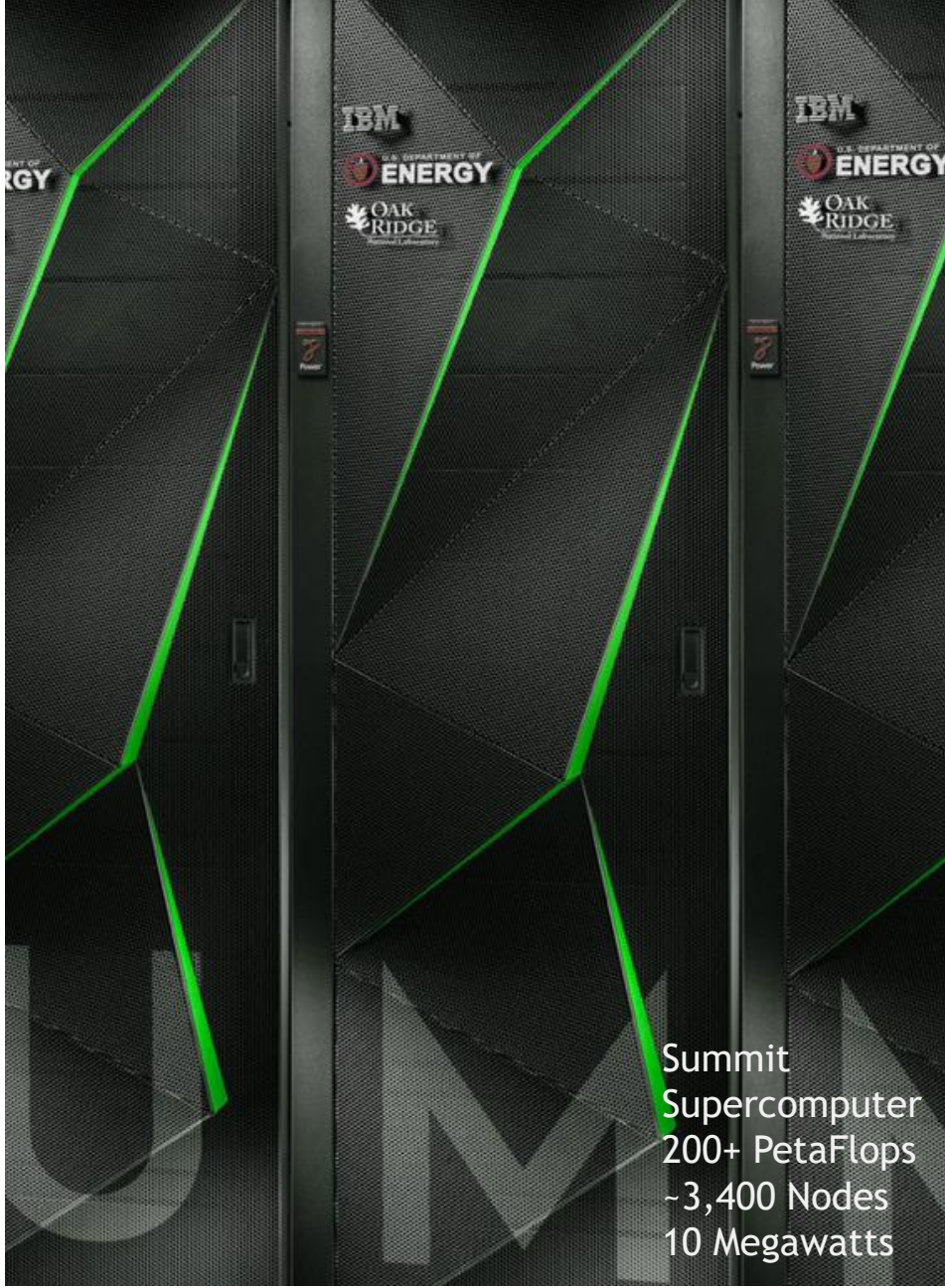


(*) DLモデルはResNet50

VOLTA
HPC性能を大きく向上



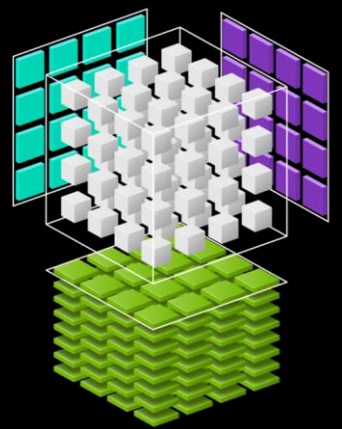
System Config Info: 2X Xeon E5-2690 v4, 2.6GHz, w/ 1X Tesla P100 or V100. V100 measured on pre-production hardware.



VOLTA 米国トップスパコンのエンジン

SUMMIT

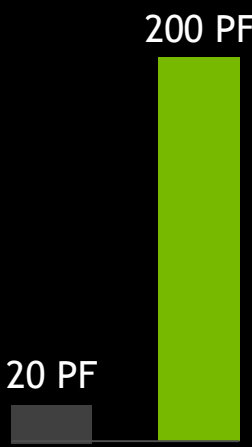
AI Exascale Today



3+EFLOPS

Tensor Ops

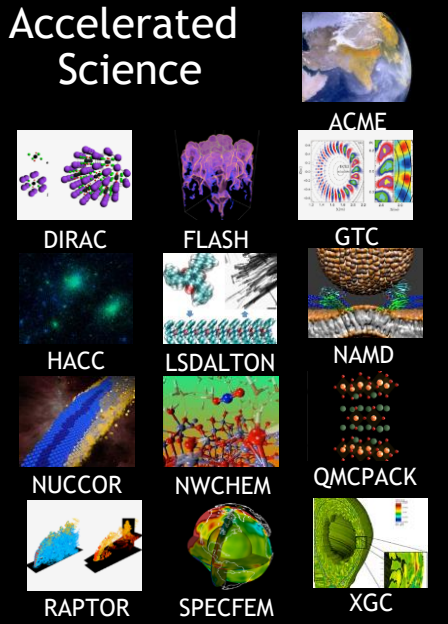
Performance Leadership



10X

Perf Over Titan

Accelerated Science



5-10X

Application Perf Over Titan

TESLA V100

トランジスタ数:21B
815 mm²

80 SM
5120 CUDAコア
640 Tensorコア

HBM2
16 GB, 900 GB/s

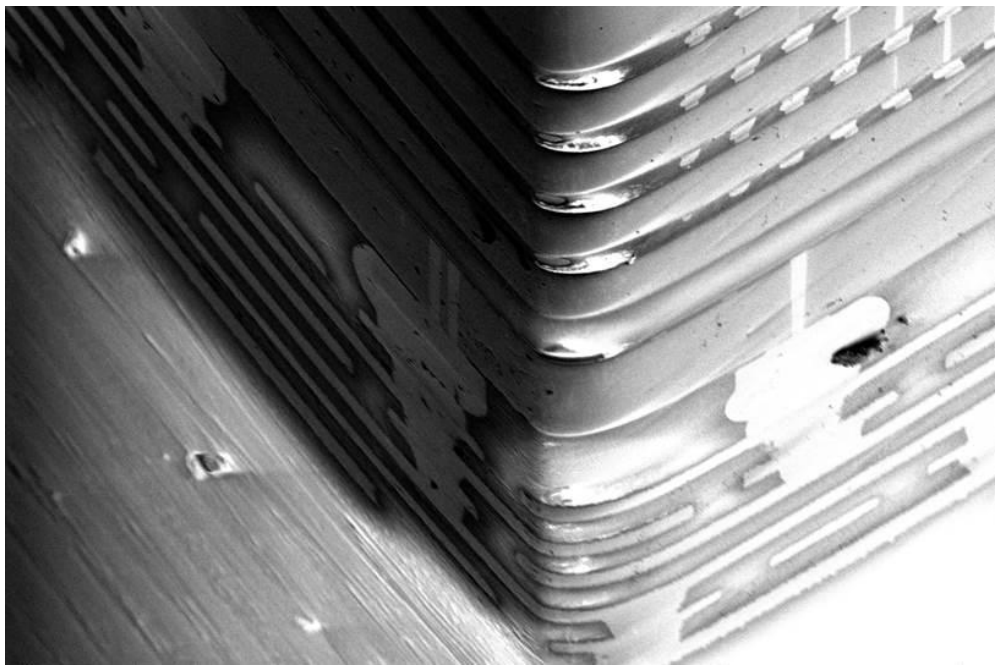
NVLink 300 GB/s

*full GV100 chip contains 84 SMs

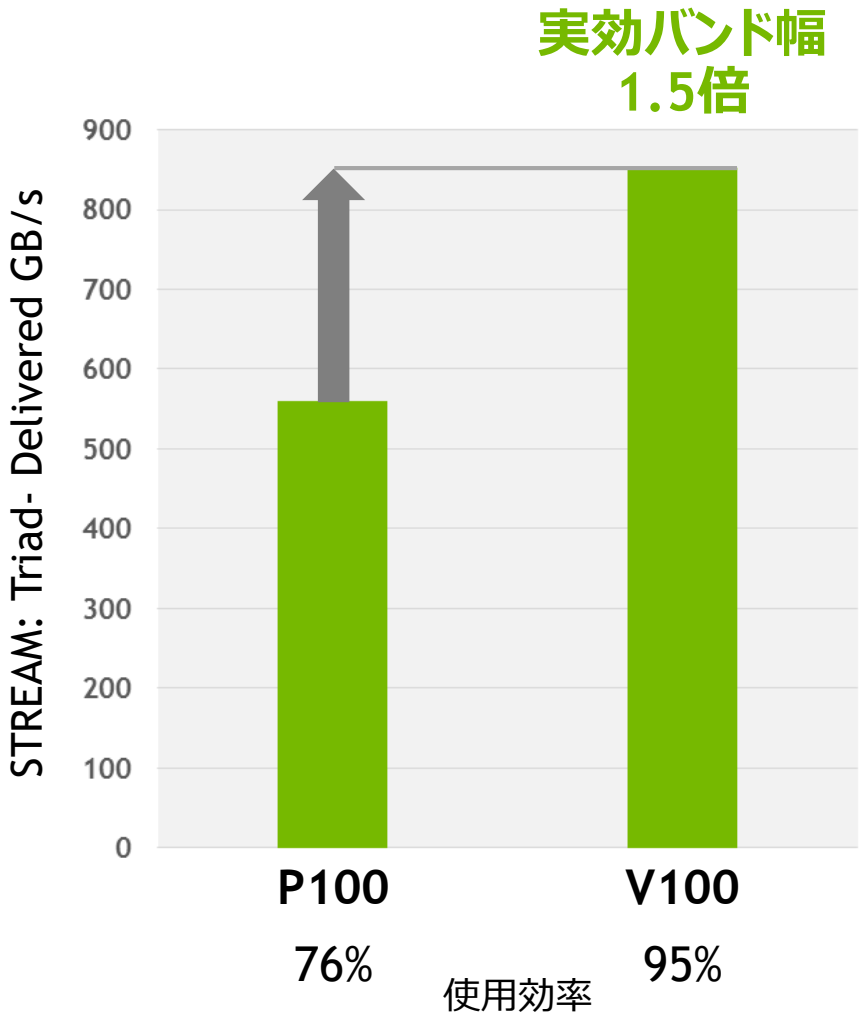
GPUピーク性能比較: P100 vs v100

	P100	V100	性能UP
トレーニング性能	10 TOPS	125 TOPS	12x
インファレンス性能	21 TFLOPS	125 TOPS	6x
FP64/FP32	5/10 TFLOPS	7.8/15.6 TFLOPS	1.5x
HBM2 バンド幅	720 GB/s	900 GB/s	1.2x
NVLink バンド幅	160 GB/s	300 GB/s	1.9x
L2 キャッシュ	4 MB	6 MB	1.5x
L1 キャッシュ	1.3 MB	10 MB	7.7x

HBM2メモリ、使用効率UP



HBM2 stack

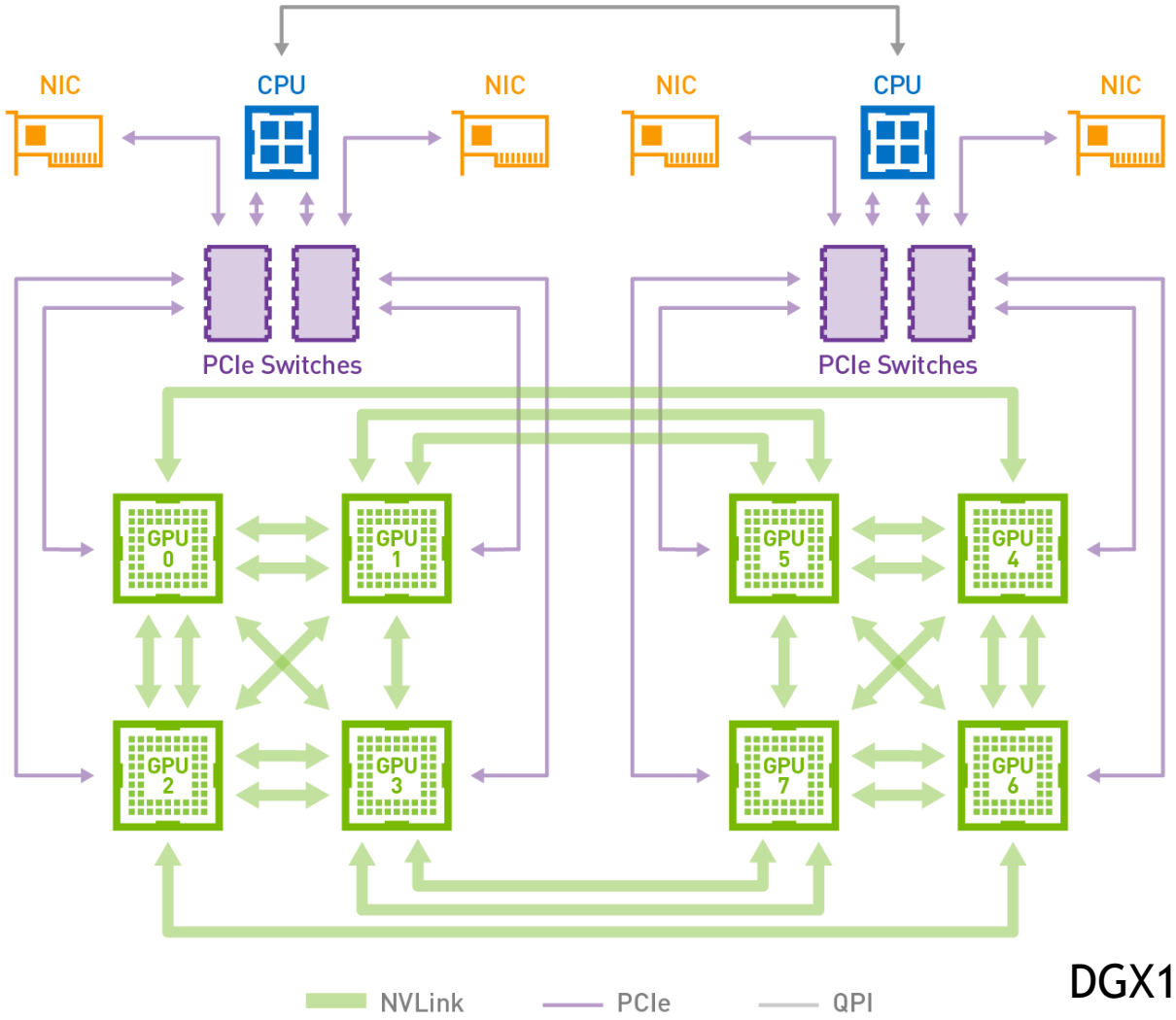


V100 measured on pre-production hardware.

VOLTA NVLINK

	P100	V100
リンク数	4	6
バンド幅 / リンク	40 GB/s	50 GB/s
トータルバンド幅	160 GB/s	300 GB/s

(*) バンド幅は双方向



DGX1V

NEW SM MICROARCHITECTURE

VOLTA GV100 SM

	GV100
FP32ユニット	64
FP64ユニット	32
INT32ユニット	64
Tensorコア	8
レジスタファイル	256 KB
統合L1・共有メモリ	128 KB
Activeスレッド	2048

(*) SMあたり

VOLTA GV100 SM

生産性の向上

命令セットを一新

スケジューラを2倍

命令発行機構をシンプルに

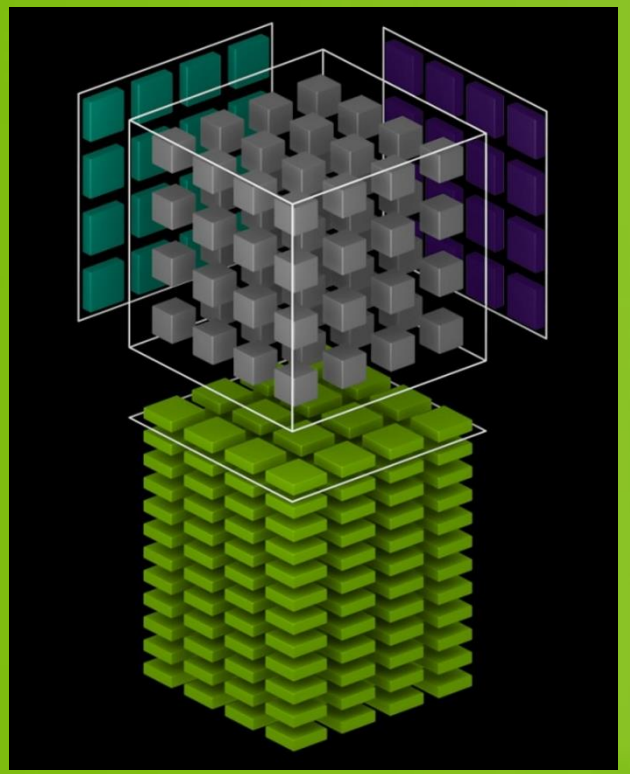
L1キャッシュの大容量・高速化

SIMTモデルの改善

テンソル計算の加速

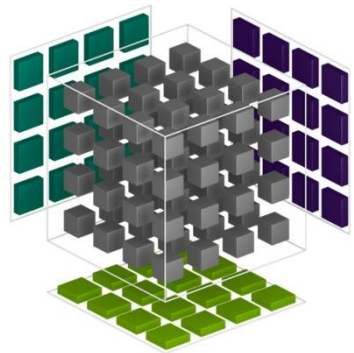
最もプログラミングの簡単なSM

VOLTA TENSORコア



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)

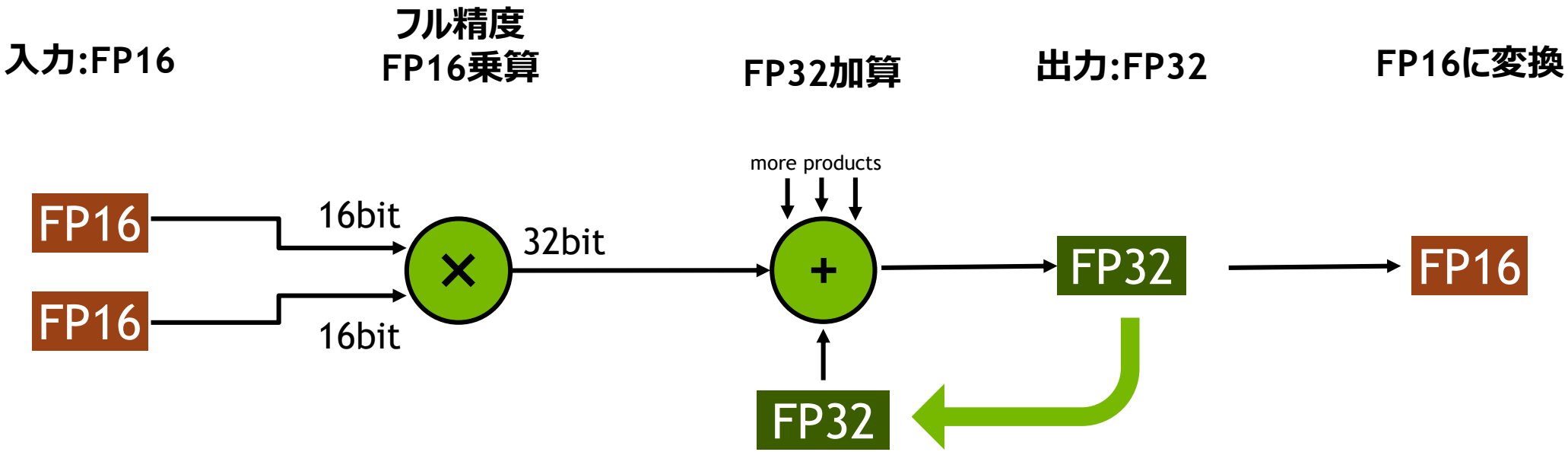
行列のFMA (Fused Multiply-Add)

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

FP16 or FP32　　FP16　　FP16　　FP16 or FP32

$$D = AB + C$$

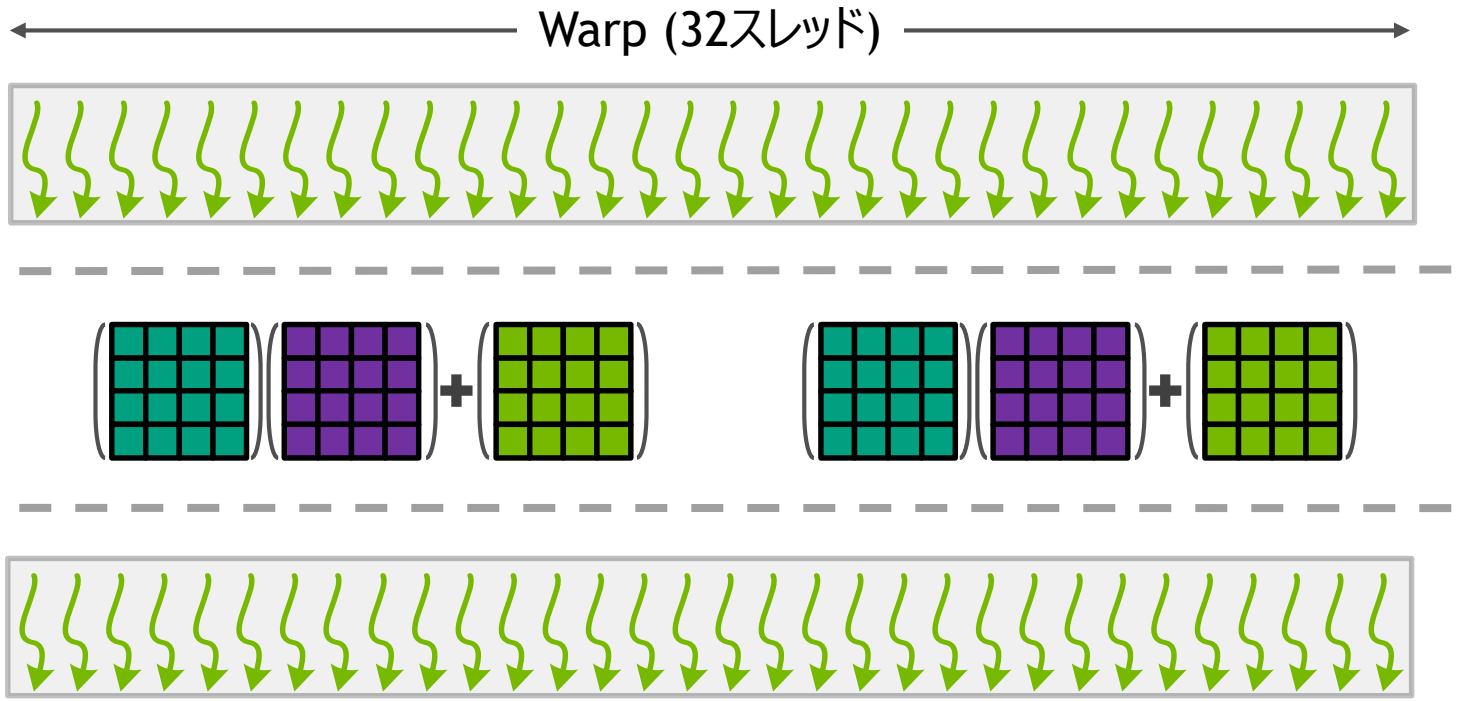
VOLTA TENSOR演算



FP16加算もサポート (インファレンス用)

TENSORコアの使われ方

16x16の行列の積和演算を、Warpレベル(32スレッド)で協調実行



Tensorコアを使用、
16x16行列の行列積を実行

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

CUBLAS: TENSORコアの使い方

cublasGemmExで行列積

```
cublasCreate( &handle );
cublasSetMathMode( handle, CUBLAS_TENSOR_OP_MATH );

algo = CUBLAS_GEMM_DFALT_TENSOR_OP;
cublasGemmEx( handle, transa, transb, m, n, k, alpha,
                      A, CUDA_R_16F, lda,
                      B, CUDA_R_16F, ldb,
                      beta,
                      C, CUDA_R_16F, ldc,
                      CUDA_R_32F, algo );
```

Tensorコア使用モードを選択

Tensorコア用の行列積アルゴリズムの選択

入力行列A,Bのデータ型を指定

出力行列Cのデータ型を指定

計算型を指定 (Tensorコアの場合は、加算の計算精度)

CUBLAS: TENSORコアの使い方

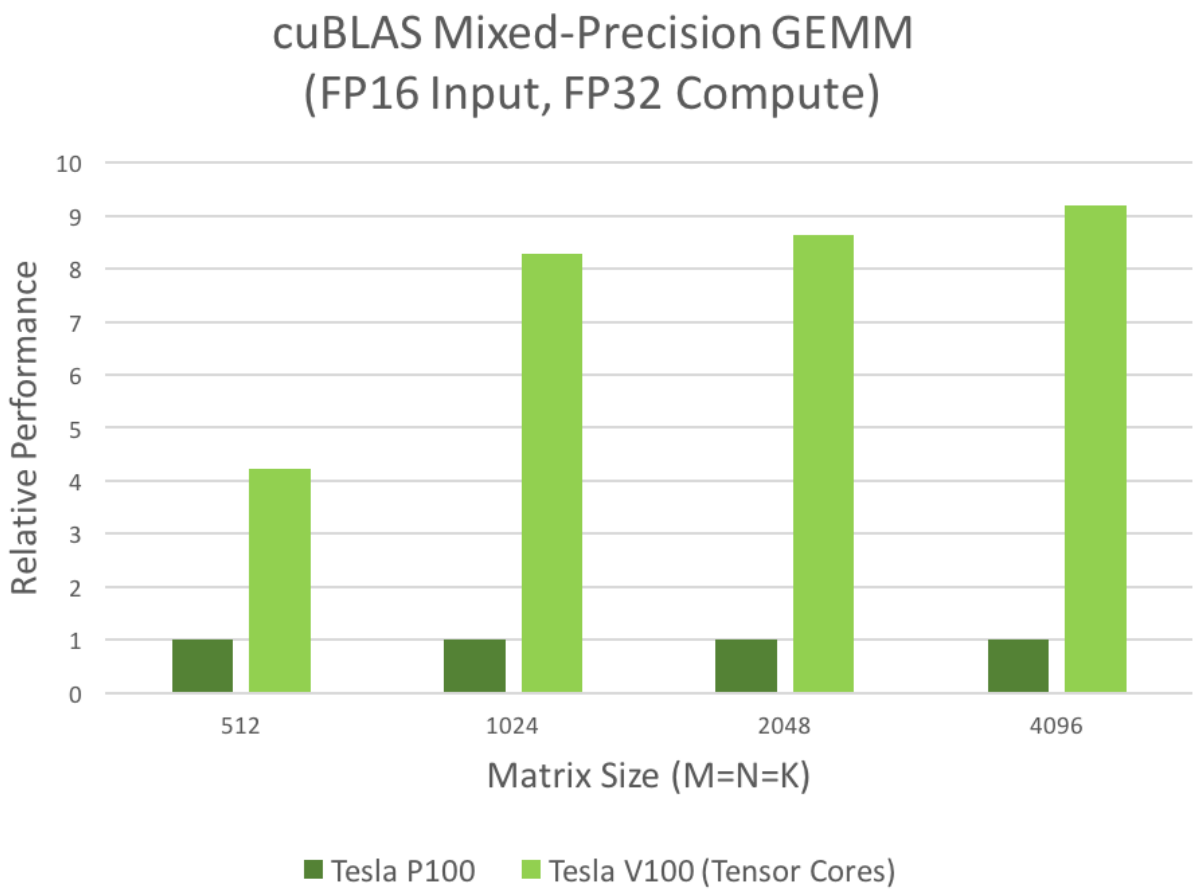
cublasGemmExで行列積

入力行列A,Bのデータ型	出力行列Cのデータ型	加算型	
FP16	FP16	FP32	(標準的な用途?)
FP16	FP16	FP16	FP16で加算 (インファレンス)
FP16	FP32	FP32	FP32で出力
FP32	FP32	FP32	FP32データのまま、Tensorコア使用

他APIでも使用可: cublasSgemmEx, cublasHgemm, cublasHgemmBatched, cublasHgemmStrideBatched

CUBLAS: TENSORコアの実効性能

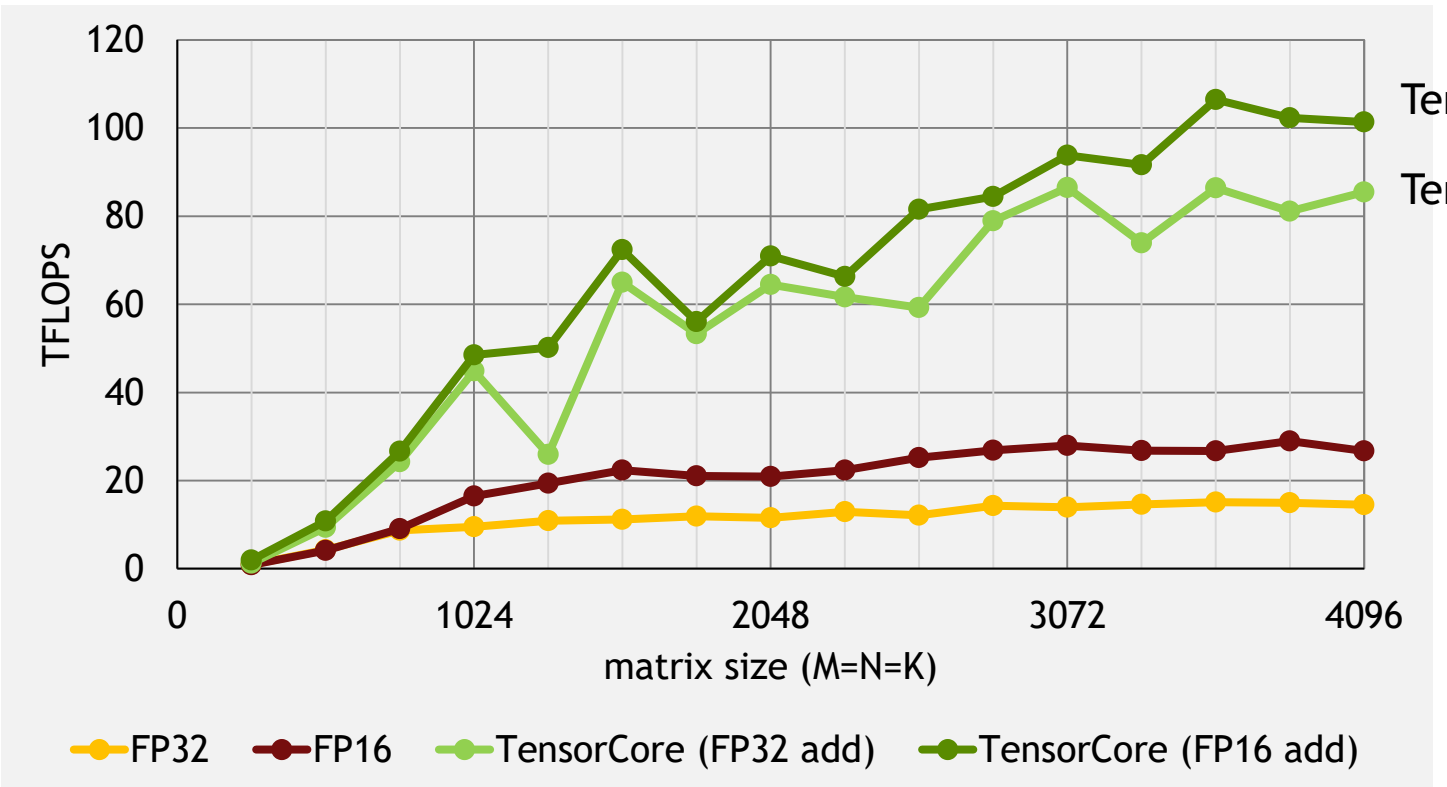
P100 FP32 vs. V100 Tensorコア



最大9倍の
性能向上

CUBLAS: TENSORコアの実効性能

V100同士で比較: FP32 vs. Tensorコア



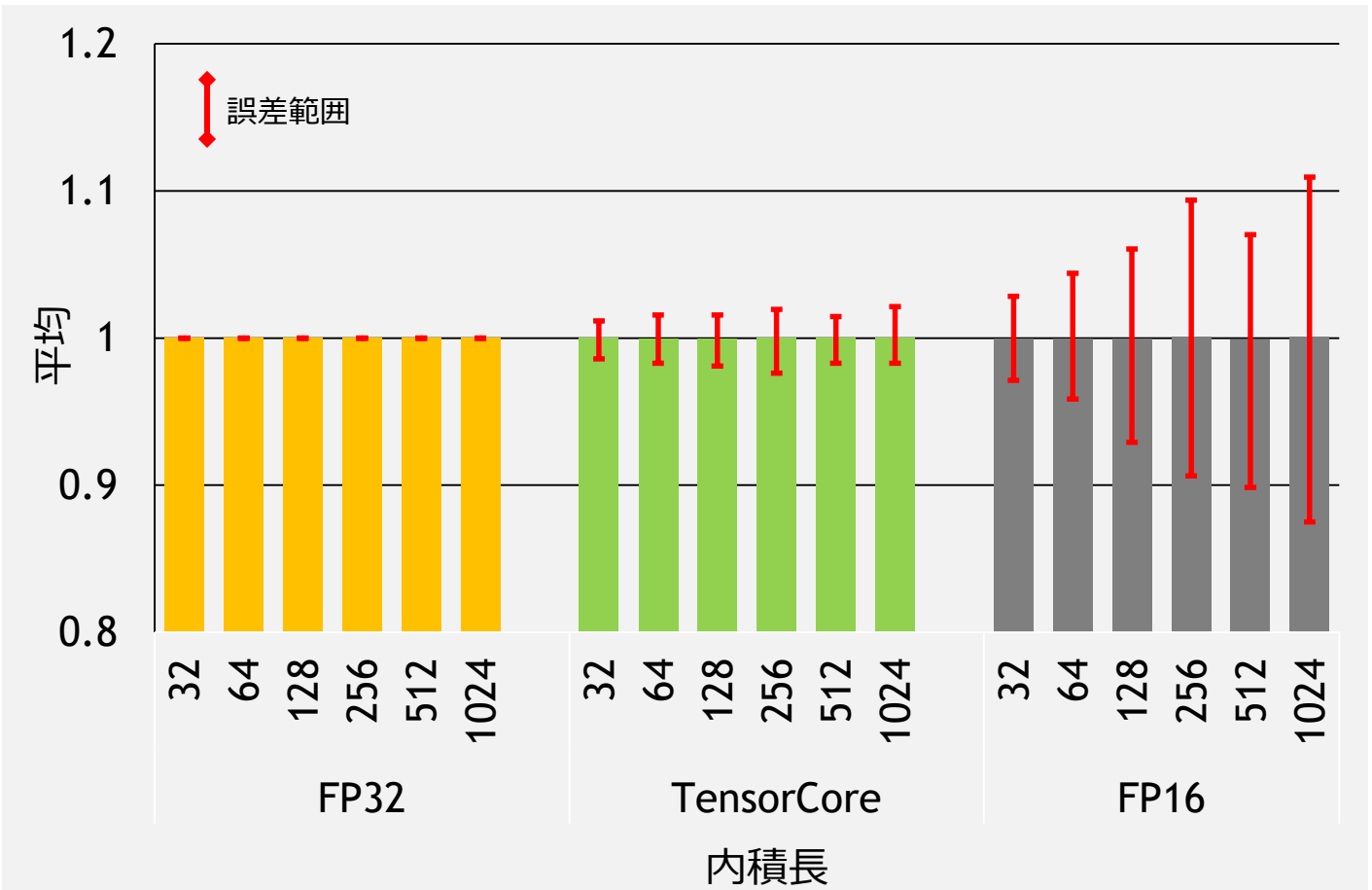
FP32と比べて、
最大で6倍以上の性能UP
(FP32加算の場合)

- CUDA 9.0.176
- cublasGemmEx()使用

TENSORコアの計算精度

FP32の計算結果に近い

アプリケーション依存

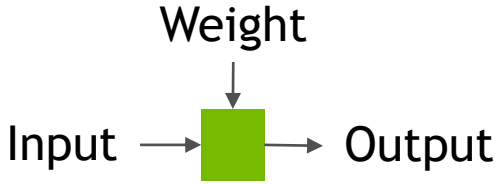


Tensorコアの演算結果は、FP16と比べて、FP32との誤差が小さい

- 行列A: 指数分布 (activation)
- 行列B: 正規分布 (weight)
  (平均0.0, 分散1.0)
- 内積長: 32 – 1024
- 1万サンプル
- 誤差区間: 99%

CUDNN: TENSORコアの使い方

Convolution




```
cudnnCreate( &handle );
cudnnCreateTensorDescriptor( &cudnnIdesc );
cudnnCreateTensorDescriptor( &cudnnOdesc );
cudnnCreateFilterDescriptor( &cudnnFdesc );
cudnnCreateConvolutionDescriptor( &cudnnConvDesc );
…
cudnnSetConvolutionNdDescriptor( cudnnConvDesc, … );
cudnnSetConvolutionMathType( cudnnConvDesc, CUDNN_TENSOR_OP_MATH );
…
algo = CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM;
…
cudnnConvolutionForward( handle, alpha, cudnnIdesc, dev_I,
          cudnnFdesc, dev_F, cudnnConvDesc, algo,
          workspace, workSpaceSize, beta, cudnnOdesc, dev_O );
```

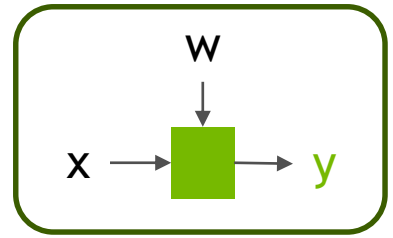
Tensorコア使用モードを選択

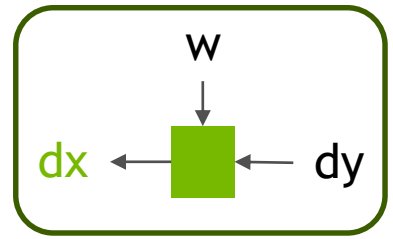
Tensorコア対応のConvolutionアルゴリズム選択

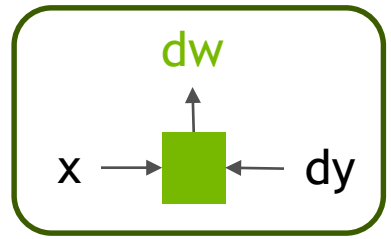
CUDNN: TENSORコアの使い方

Convolution: Tensorコア対応アルゴリズム

- Forward
  - CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM
  - CUDNN_CONVOLUTION_FWD_ALGO_WINOGRAD_NONFUSED
- BackwardData
  - CUDNN_CONVOLUTION_BWD_DATA_ALGO_1
  - CUDNN_CONVOLUTION_BWD_DATA_ALGO_WINOGRAD_NONFUSED
- BackwardFilter
  - CUDNN_CONVOLUTION_BWD_FILTER_ALGO_1
  - CUDNN_CONVOLUTION_BWD_FILTER_ALGO_WINOGRAD_NONFUSED

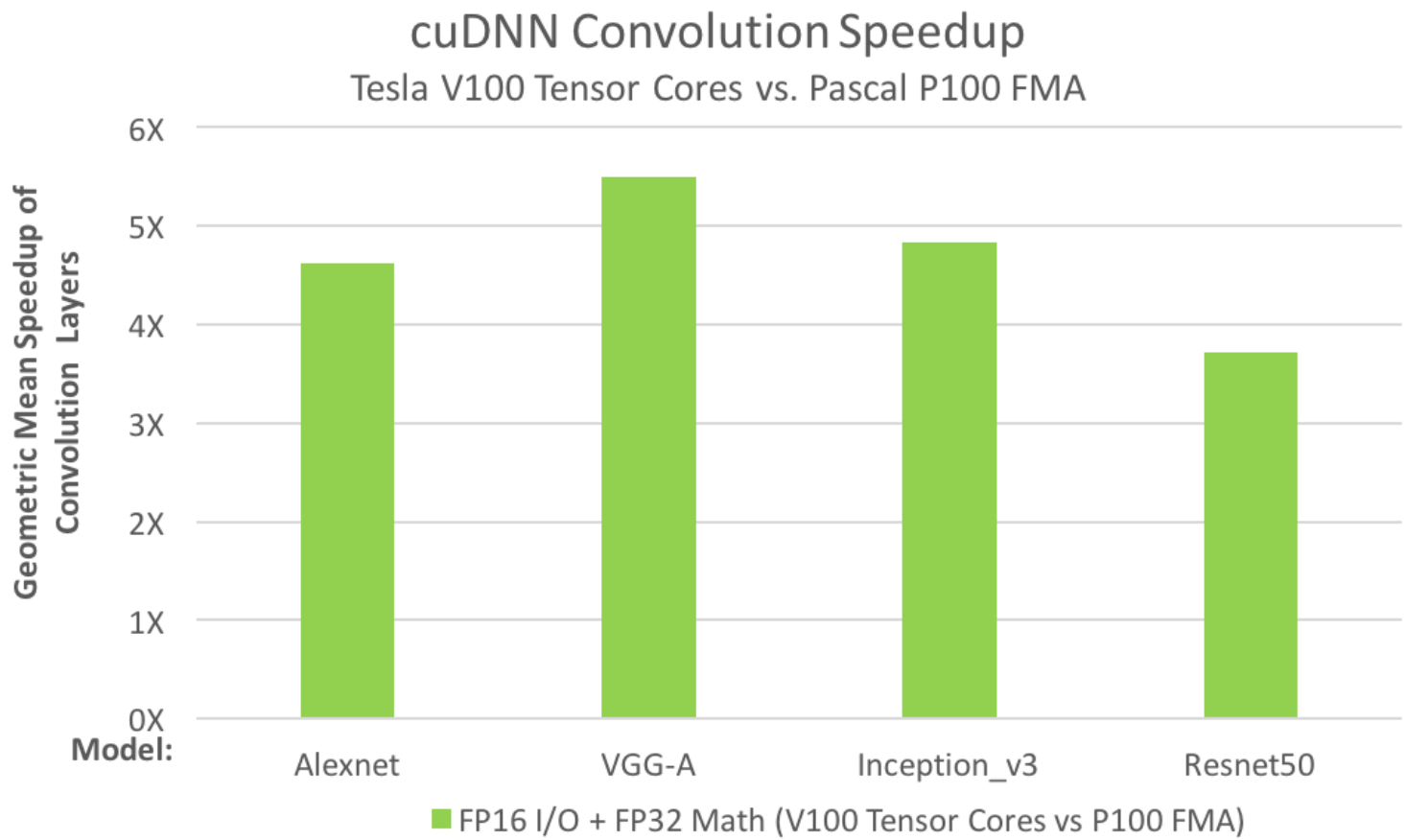






CUDNN: TENSORコアの実効性能

Pascal FP32 vs. V100 Tensorコア



Convolution層の性能比較

INDEPENDENT THREAD SCHEDULING

VOLTA GV100 SM

生産性の向上

命令セットを一新

スケジューラを2倍

命令発行機構をシンプルに

L1キャッシュの大容量・高速化

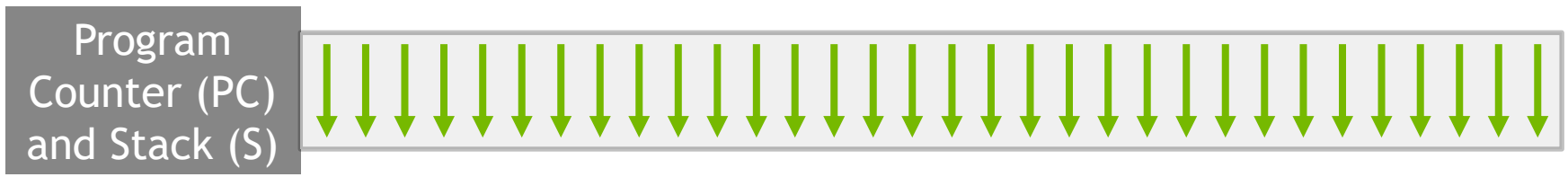
SIMTモデルの改善

テンソル計算の加速

最もプログラミングの簡単なSM

WARPの実装

Pascalまで



Warp(32スレッド)毎に、PCは1つ

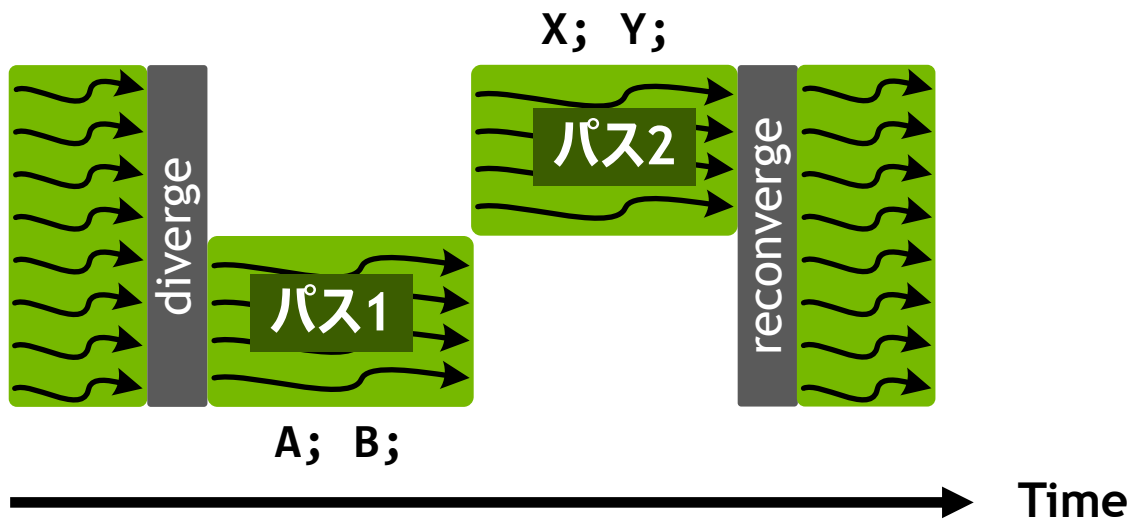
PASCAL: WARP実行モデル

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

パス1

パス2

Warp内で複数パスに分岐した場合、
一方のパスが完了するまで、
もう一方のパスは実行されない

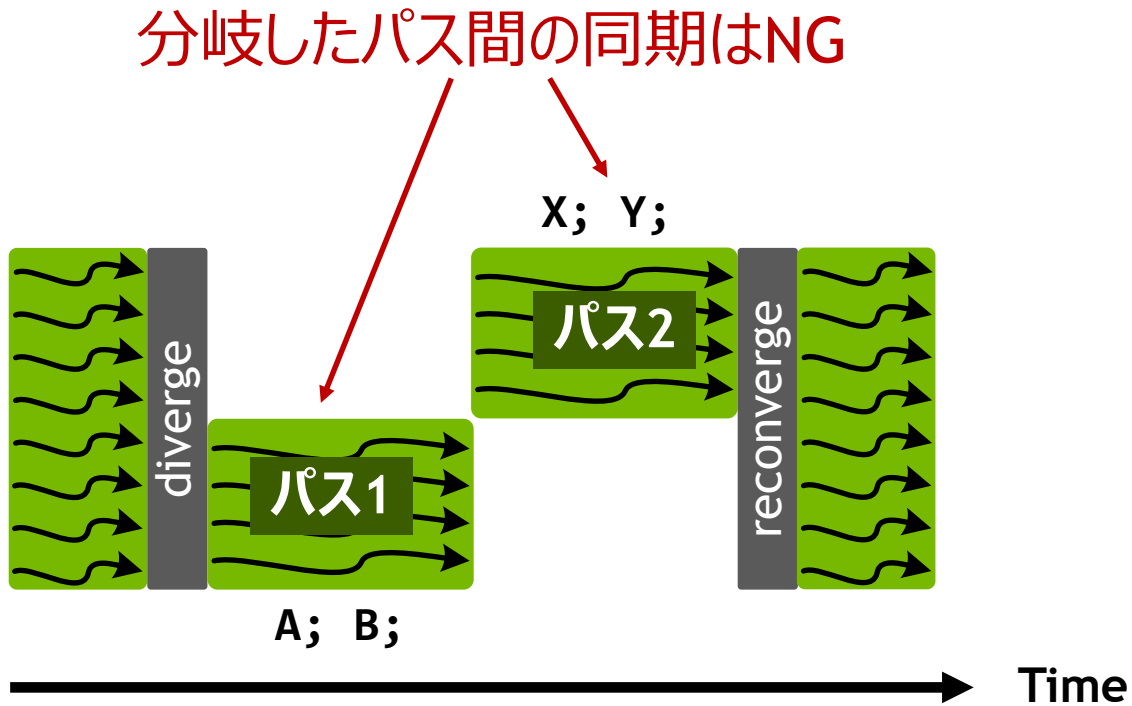


PASCAL: WARP実行モデル

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

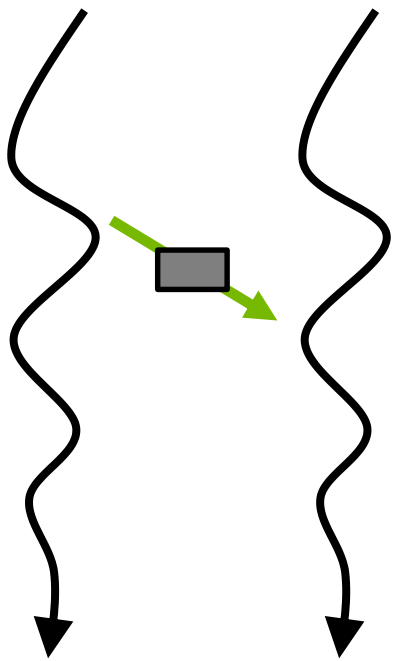
パス1

パス2



スレッド間で通信するプログラム

Pascal

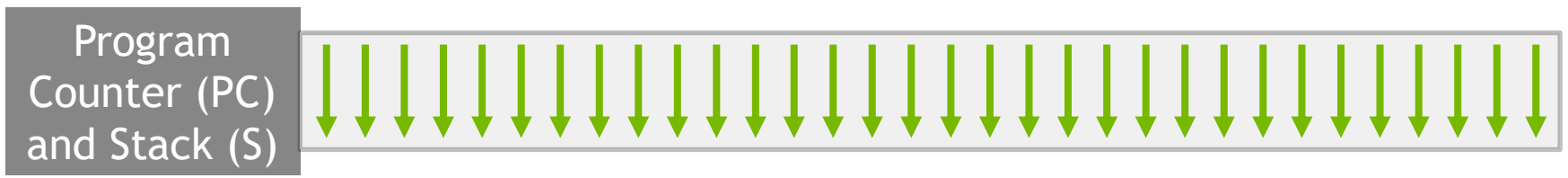


Lock-Freeアルゴリズムであれば**OK**

他スレッドを待つのは**NG**

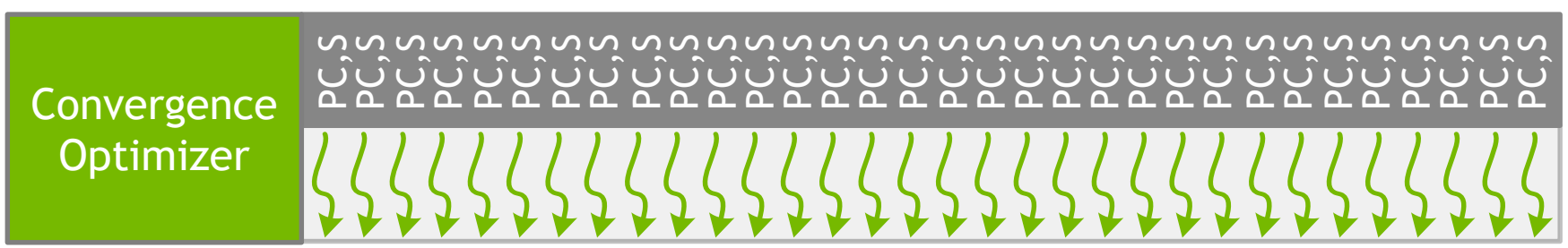
WARPの実装

Pascalまで



Warp(32スレッド)毎に、PCは1つ

Volta



スレッド毎にPCを管理、個別にスケジューリングが可能

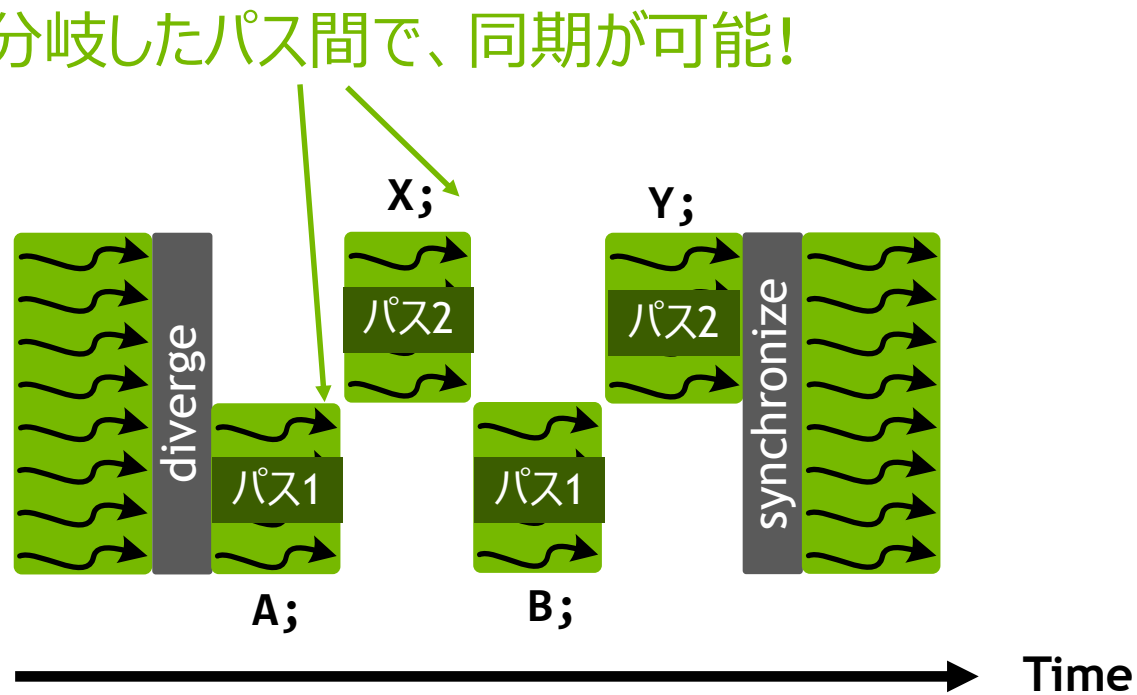
VOLTA: 拡張WARP実行モデル

Thread Independent Scheduling

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

パス1

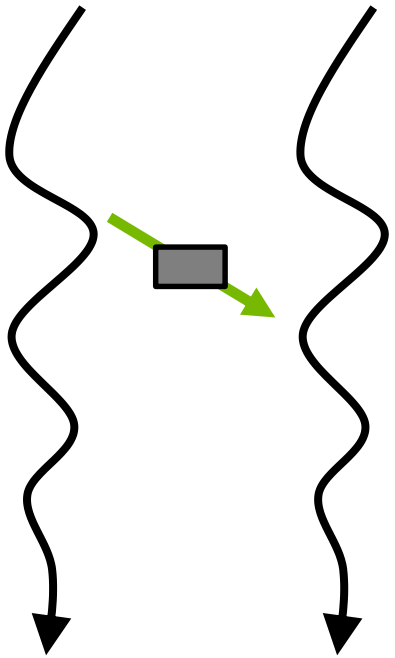
パス2



(注意) 同じワープの別スレッドが、同じサイクルに別インストラクションの実行は出来ない

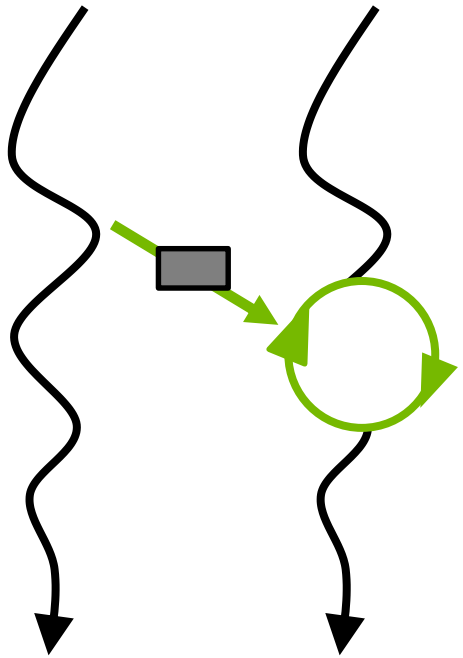
スレッド間で通信するプログラム

Pascal



Lock-Freeアルゴリズムであれば**OK**
他スレッドを待つのは**NG**

Volta



Starvation Freeアルゴリズムであれば**OK**
他スレッドを待っても**OK**

STARVATION FREEアルゴリズムの例

双方向リンクリスト

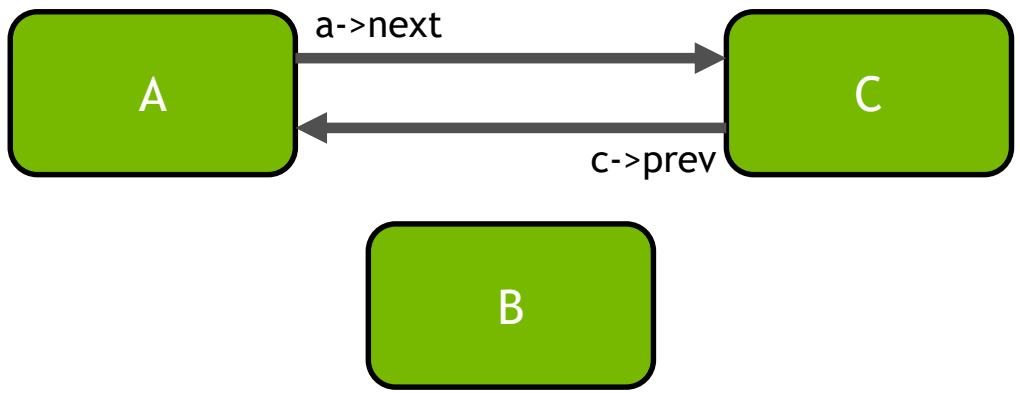
```
__device__ void insert_after(Node *a, Node *b)
{
    Node *c;
    lock(a); lock(a->next);
    c = a->next;

    a->next = b;
    b->prev = a;

    b->next = c;
    c->prev = b;

    unlock(c); unlock(a);
}
```

Doubly-Linked List with Fine Grained Lock



STARVATION FREEアルゴリズムの例

双方向リンクリスト

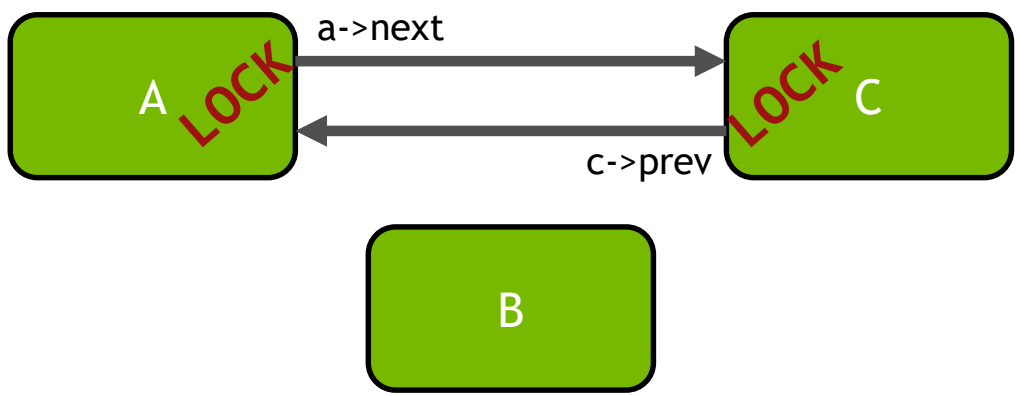
```
__device__ void insert_after(Node *a, Node *b)
{
    Node *c;
    lock(a); lock(a->next);
    c = a->next;

    a->next = b;
    b->prev = a;

    b->next = c;
    c->prev = b;

    unlock(c); unlock(a);
}
```

Doubly-Linked List with Fine Grained Lock



***Not shown: lock() implementation**

STARVATION FREEアルゴリズムの例

双方向リンクリスト

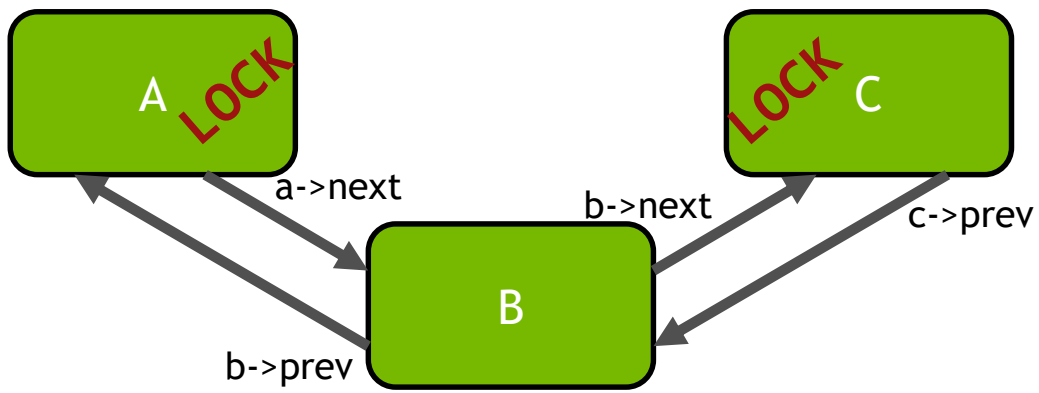
```
__device__ void insert_after(Node *a, Node *b)
{
    Node *c;
    lock(a); lock(a->next);
    c = a->next;

    a->next = b;
    b->prev = a;

    b->next = c;
    c->prev = b;

    unlock(c); unlock(a);
}
```

Doubly-Linked List with Fine Grained Lock



STARVATION FREEアルゴリズムの例

双方向リンクリスト

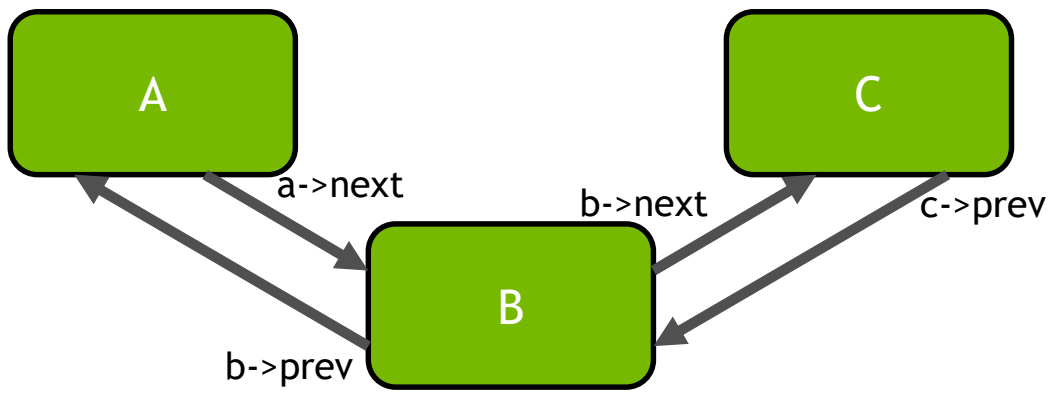
```
__device__ void insert_after(Node *a, Node *b)
{
    Node *c;
    lock(a); lock(a->next);
    c = a->next;

    a->next = b;
    b->prev = a;

    b->next = c;
    c->prev = b;

    unlock(c); unlock(a);
}
```

Doubly-Linked List with Fine Grained Lock



Pascalで、このプログラムを実行するのは危険
アルゴリズムをLock-freeに変える必要がある

STARVATION FREEアルゴリズムの例

双方向リンクリスト

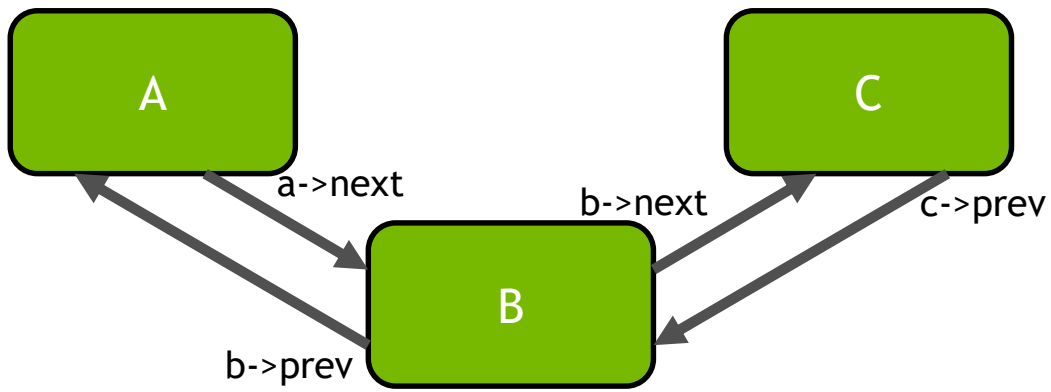
```
__device__ void insert_after(Node *a, Node *b)
{
    Node *c;
    lock(a); lock(a->next);
    c = a->next;

    a->next = b;
    b->prev = a;

    b->next = c;
    c->prev = b;

    unlock(c); unlock(a);
}
```

Doubly-Linked List with Fine Grained Lock



Voltaは最大16万スレッドを同時起動できるので、
あるスレッドがlock獲得で停滞しても、
他のスレッドが処理を進められる

VOLTA: 拡張SIMTモデル

	CPU	GPU (Pascal)	GPU (Volta)
データ並列	SIMD	SIMT	SIMT
スレッド並列 (タスク並列)	MIMD	SIMT (lock-free)	SIMT

Pascalまで

- スレッド並列のプログラムは、アルゴリズムをlock-freeに変更する必要

Volta

- アルゴリズム変更なく(or 少なく)、GPUで実行可能に

L1 CACHE AND SHARED MEMORY

VOLTA GV100 SM

生産性の向上

命令セットを一新

スケジューラを2倍

命令発行機構をシンプルに

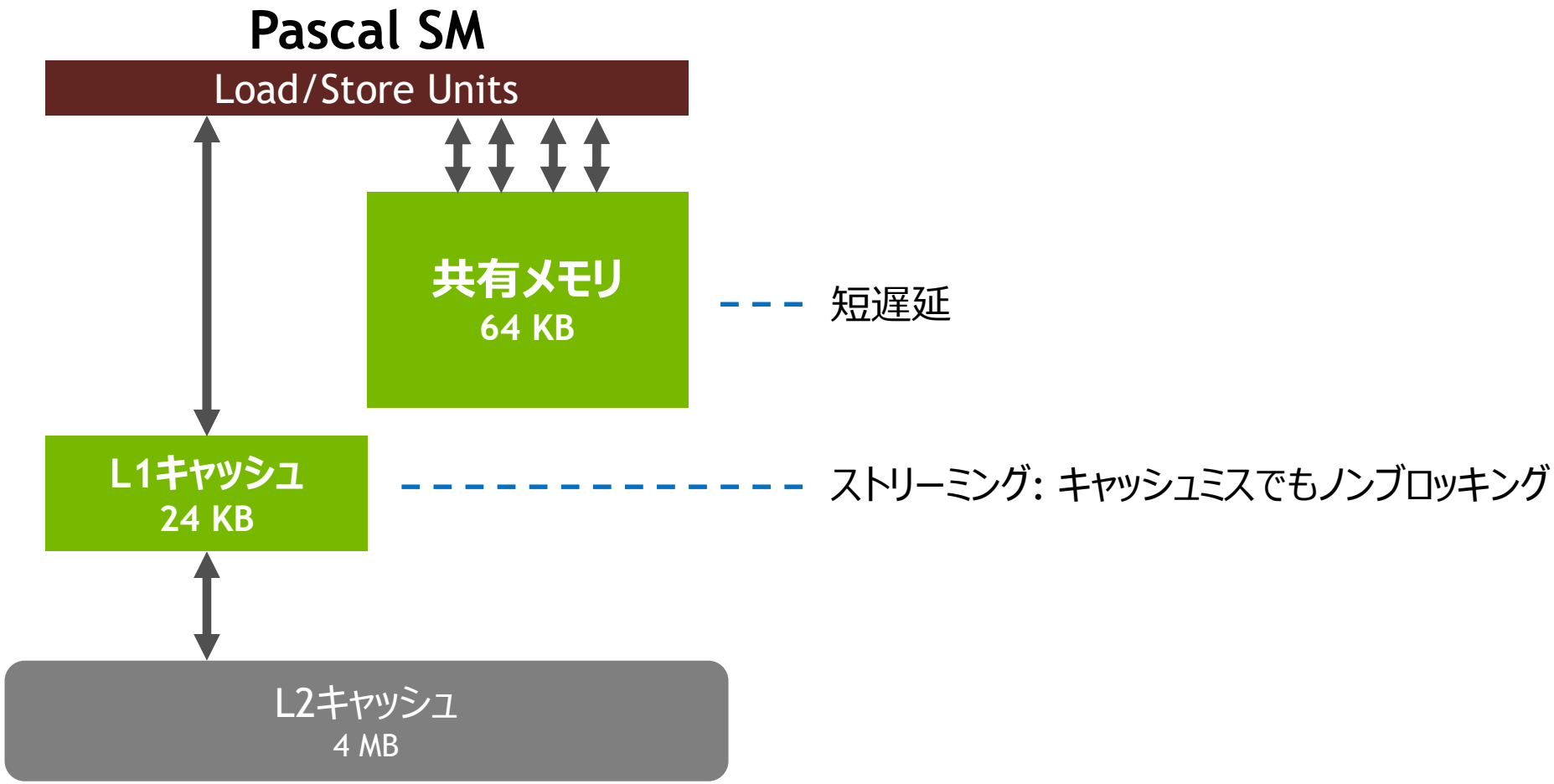
L1キャッシュの大容量・高速化

SIMTモデルの改善

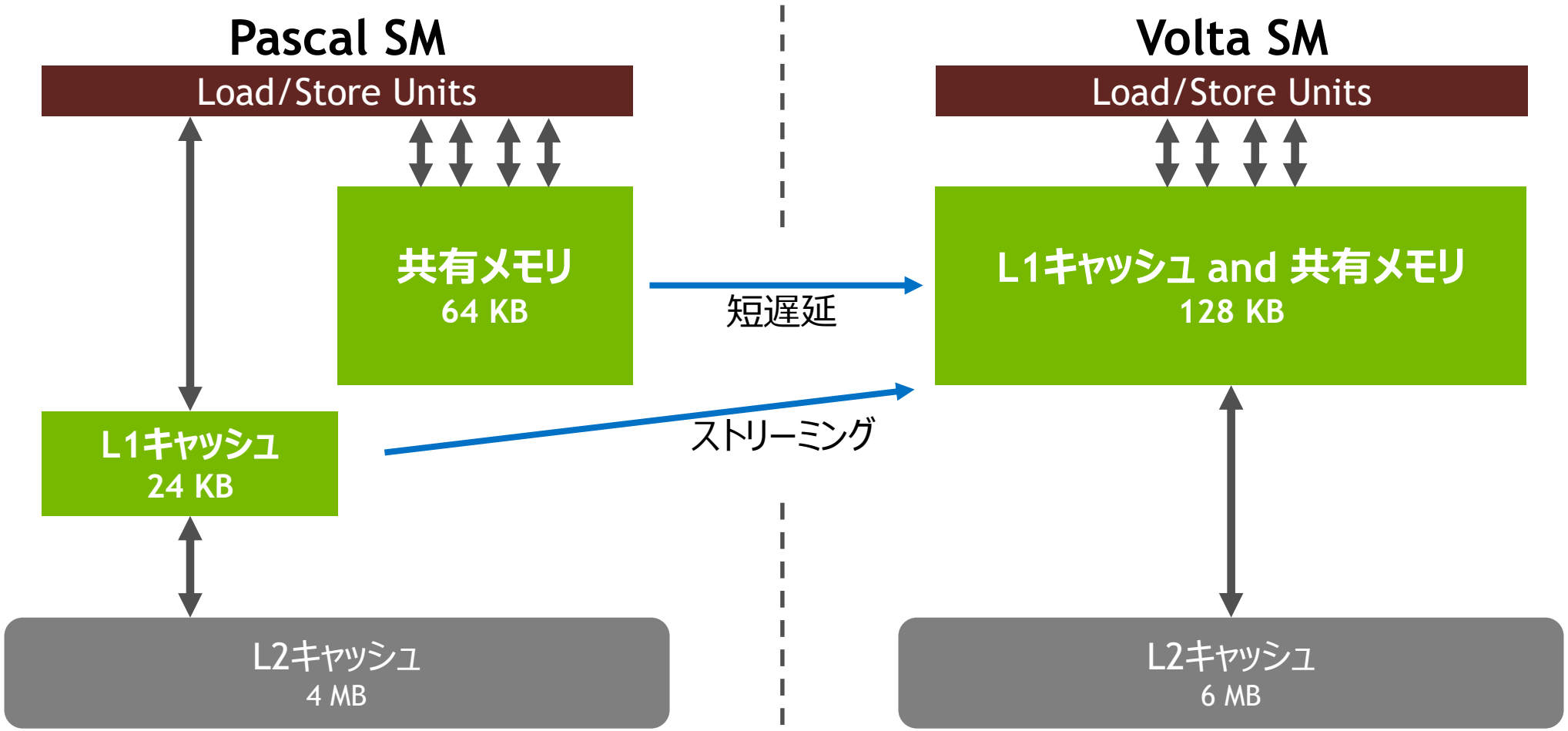
テンソル計算の加速

最もプログラミングの簡単なSM

PASCALのL1キャッシュと共有メモリ



VOLTA: L1キャッシュと共有メモリの統合



VOLTA: L1キャッシュと共有メモリの統合

Volta: ストリーミングL1キャッシュ

ノンブロッキング

短い遅延

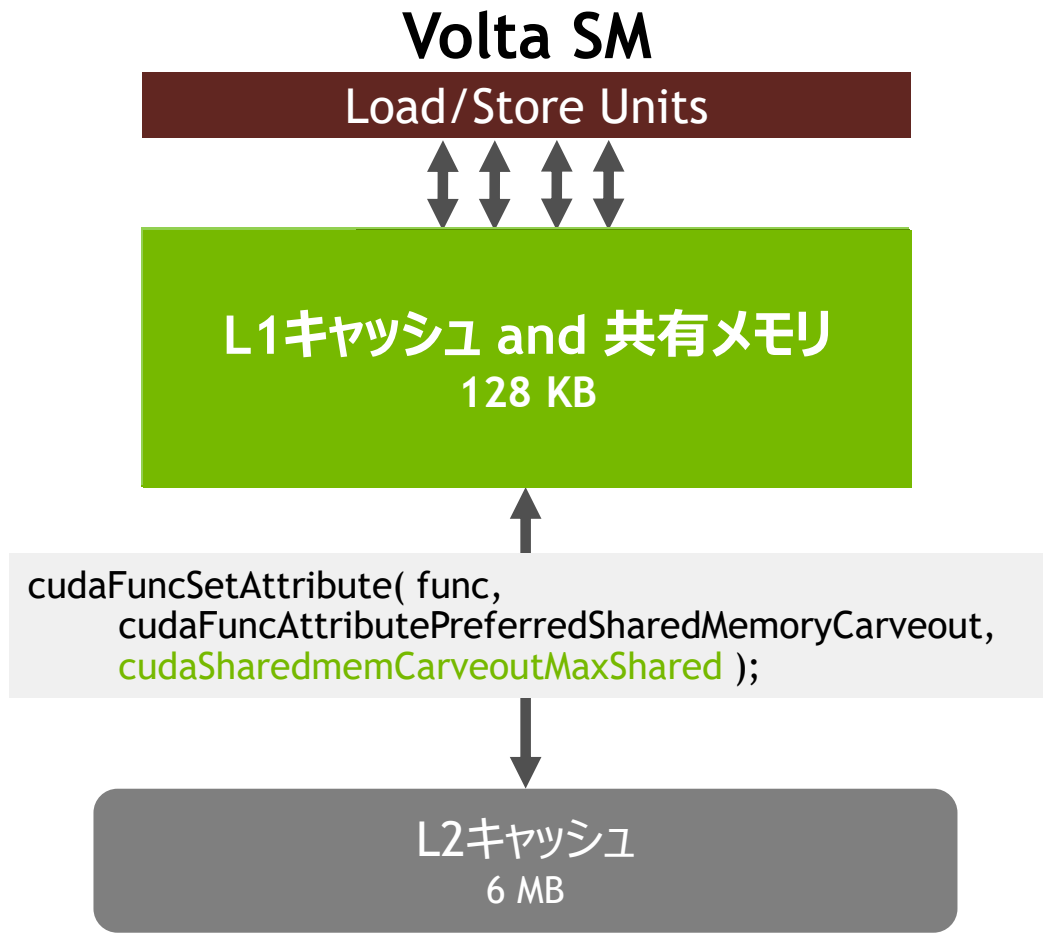
4倍以上のバンド幅

5倍以上の容量

Volta: 共有メモリ

L1キャッシュとストレージを共用

最大96KBまで設定可能 (カーネル毎)



L1キャッシュで、共有メモリ使用時相応の性能を

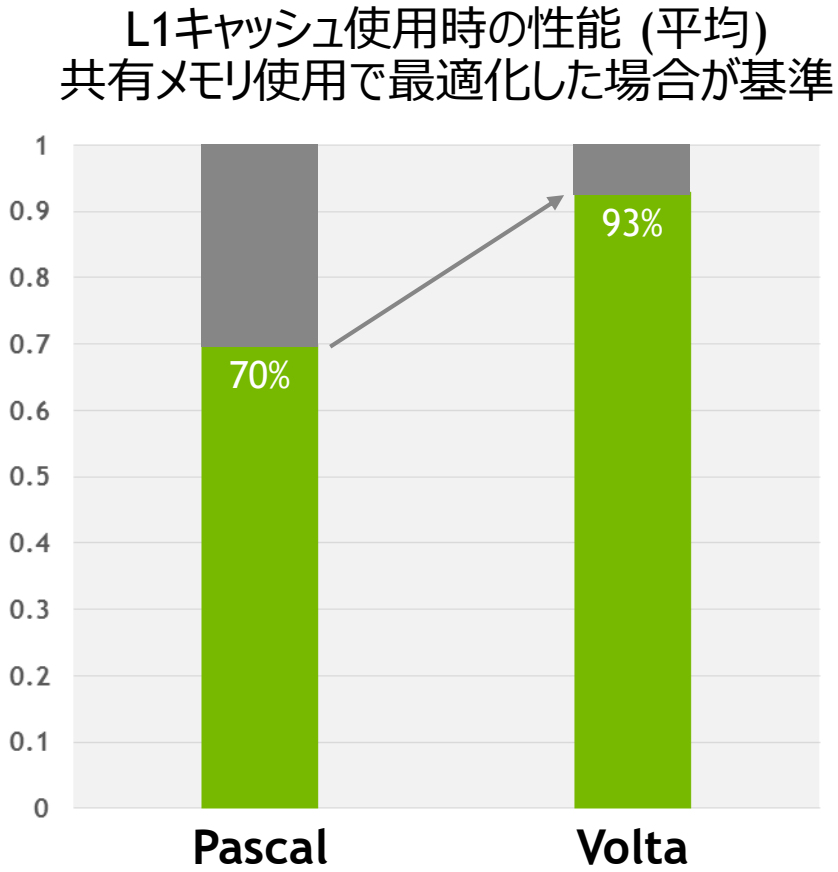
Volta L1キャッシュ

キャッシュ

- 簡単に使える (ソースコード変更不要)
- 90%以上のケースで同等の性能

共有メモリ

- スレッド間の協調が必要なとき
- Atomicsが高速
- 安定した性能



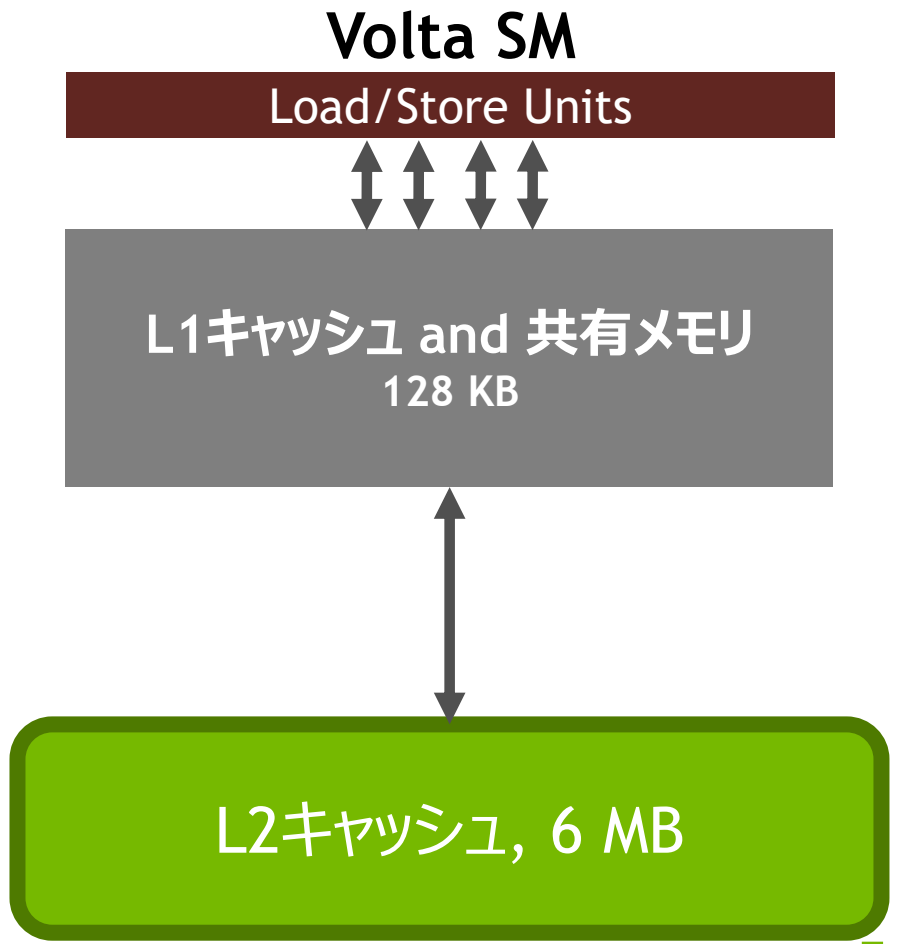
VOLTA: L2キャッシュの改善

Volta: ストリーミングL1キャッシュ

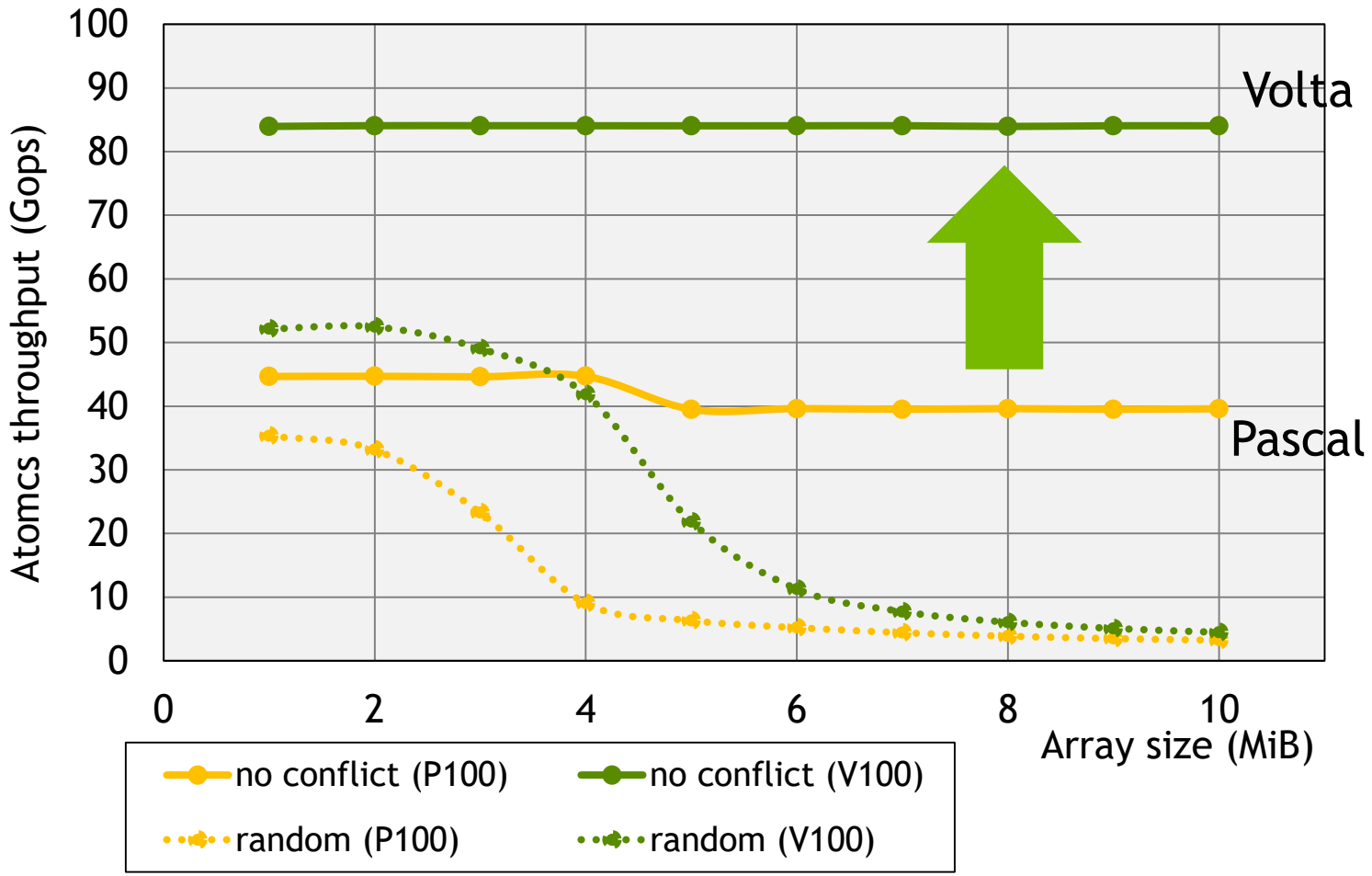
ノンブロッキング
短い遅延
4倍以上のバンド幅
5倍以上の容量

Volta: 共有メモリ

L1キャッシュとストレージを共用
最大96KBまで設定可能 (カーネル毎)



L2 ATOMICS性能の改善



最大2倍のスループット向上

- AtomicAdd(FP32)
- 256M threads
- アクセスパターン:
  規則的, ランダム

SCHEDULER

VOLTA GV100 SM

生産性の向上

命令セットを一新

スケジューラを2倍

命令発行機構をシンプルに

L1キャッシュの大容量・高速化

SIMTモデルの改善

テンソル計算の加速

最もプログラミングの簡単なSM

SM: PASCALとVOLTAの相違

- ワープスケジューラ: 2 → 4
- FP32ユニット/スケジューラ: 32 → 16

Pascal

Volta

PASCALのスケジューラ

Pascal

GP100:

- 1個のスケジューラに、2個のディスパッチャー
- 各ディスパッチャーが、16CUDAコアを担当

- スケジューラは、1サイクルに1回、Warpを選択、どちらかのディスパッチャーに渡す
- 各ディスパッチャーは、2サイクルに1回、16CUDAコアに命令を投入
- 投入された命令は、2サイクル使って、32スレッドの処理を実行
  - 32 スレッド = 16 CUDAコア x 2サイクル

VOLTAのスケジューラ

GV100:

- 1個のスケジューラに、1個のディスパッチャー
- 各ディスパッチャーが、16個のFP32ユニットとINTユニットを担当

- スケジューラは、1サイクルに1回、Warpを選択、ディスパッチャーに渡す
- ディスパッチャーは、1サイクルに1回、16個のFP32 or INTユニットに、命令を投入
- 投入された命令は、2サイクル使って、32スレッドの処理を実行

VOLTAのスケジューラ

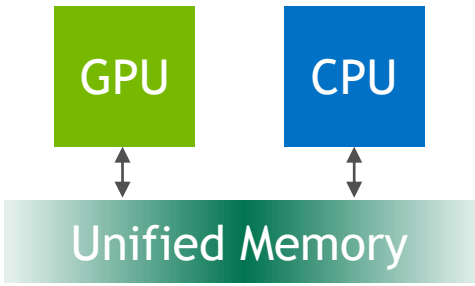
GV100:

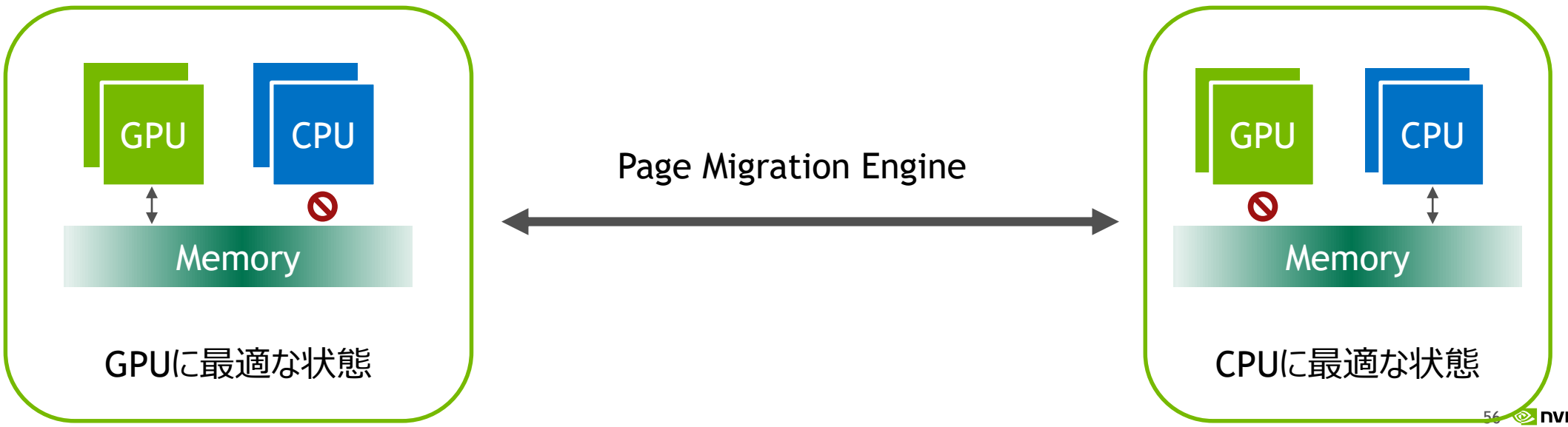
- 1個のスケジューラに、1個のディスパッチャー
- 各ディスパッチャーが、16個のFP32ユニットとINTユニットを担当

FP32とINTの同時実行が可能

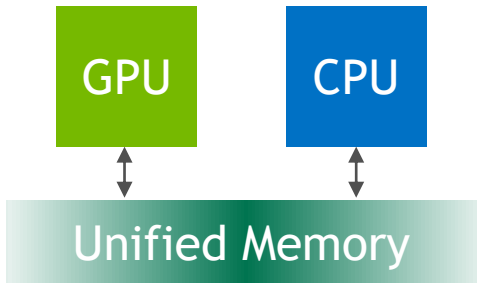
UNIFIED MEMORY

PASCALのユニファイド・メモリ

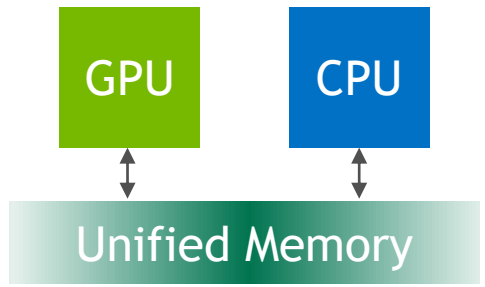




VOLTAのユニファイド・メモリ (CPUとPCIで接続)



VOLTAのユニファイド・メモリ (CPUとNVLINKで接続)



ユニファイド・メモリの状況

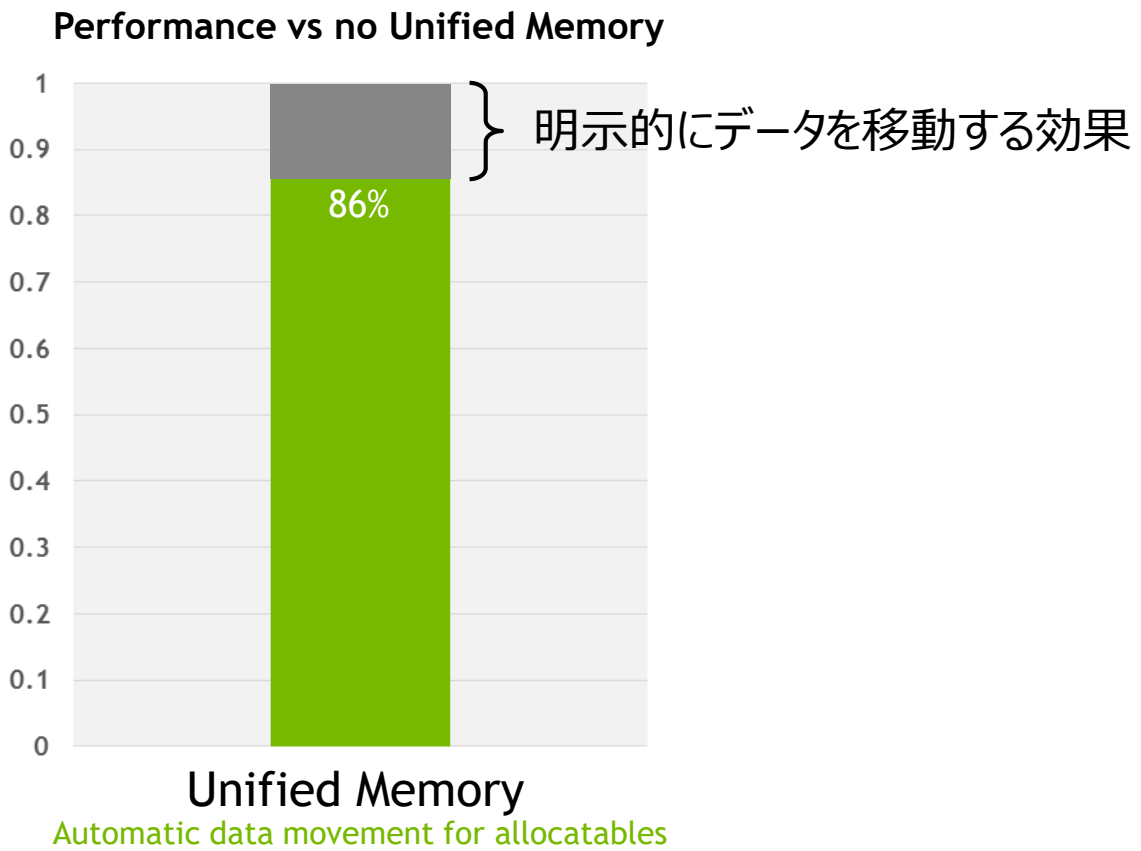
少ない労力で、高い性能を

OpenACC on P100

- PGIのOpenACCコンパイラは、ユニファイド・メモリをサポート (コンパイラ・オプション)

SPEC ACCELベンチマーク、15個の平均性能 (データ移動を手動で最適化した場合との比較)

- PCIe: 86%
- NVLINK: 91%



PGI 17.1 Compilers OpenACC SPEC ACCEL™ 1.1 performance measured March, 2017. SPEC® and the benchmark name SPEC ACCEL™ are registered trademarks of the Standard Performance Evaluation Corporation.

ロードマップ: UNIFIED SYSTEM ALLOCATOR

標準のmalloc()で、ユニファイド・メモリが使えるようになる

CUDAコード with System Allocator

```
void sortfile(FILE *fp, int N) {
  char *data;

  // Allocate memory using any standard allocator
  data = (char *) malloc(N * sizeof(char));

  fread(data, 1, N, fp);

  sort<<<...>>>(data,N,1,compare);

  use_data(data);

  // Free the allocated memory
  free(data);
}
```

OSサポートが必要

- HMM Linux Kernel Module
- Linux kernel 4.14にマージ

CPUとGPU間のデータ移動は、透過的に行われる (ユニファイド・メモリと同様)

CUDA MULTI-PROCESS SCHEDULING

GPU上のマルチ・プロセスのスケジューリング

背景

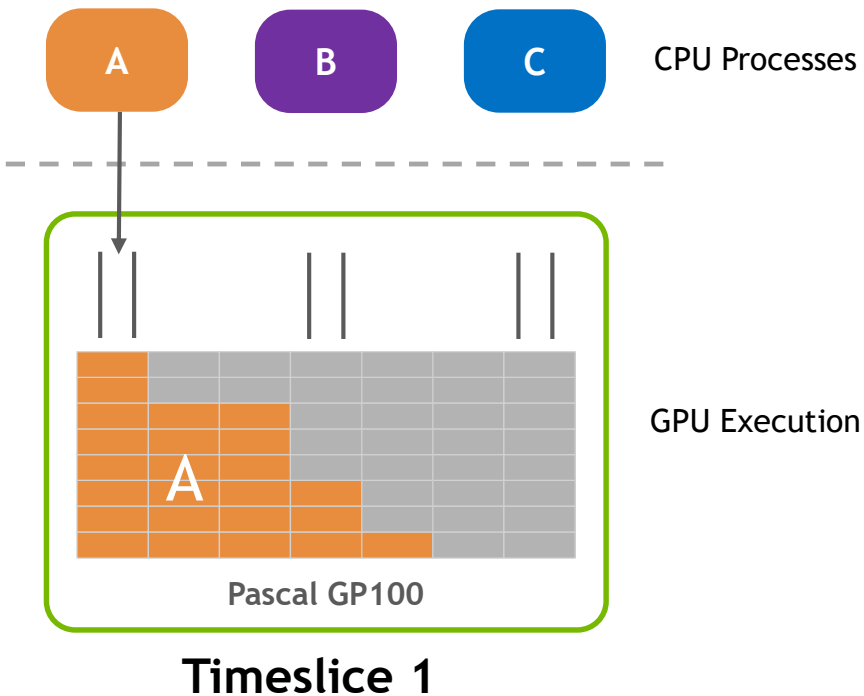
時分割スケジューリング

- GPU使用時間帯を、プロセスに配分
- あるタイミングで、GPUを使用しているプロセスは、一つ
- 各プロセスの排他実行を重視

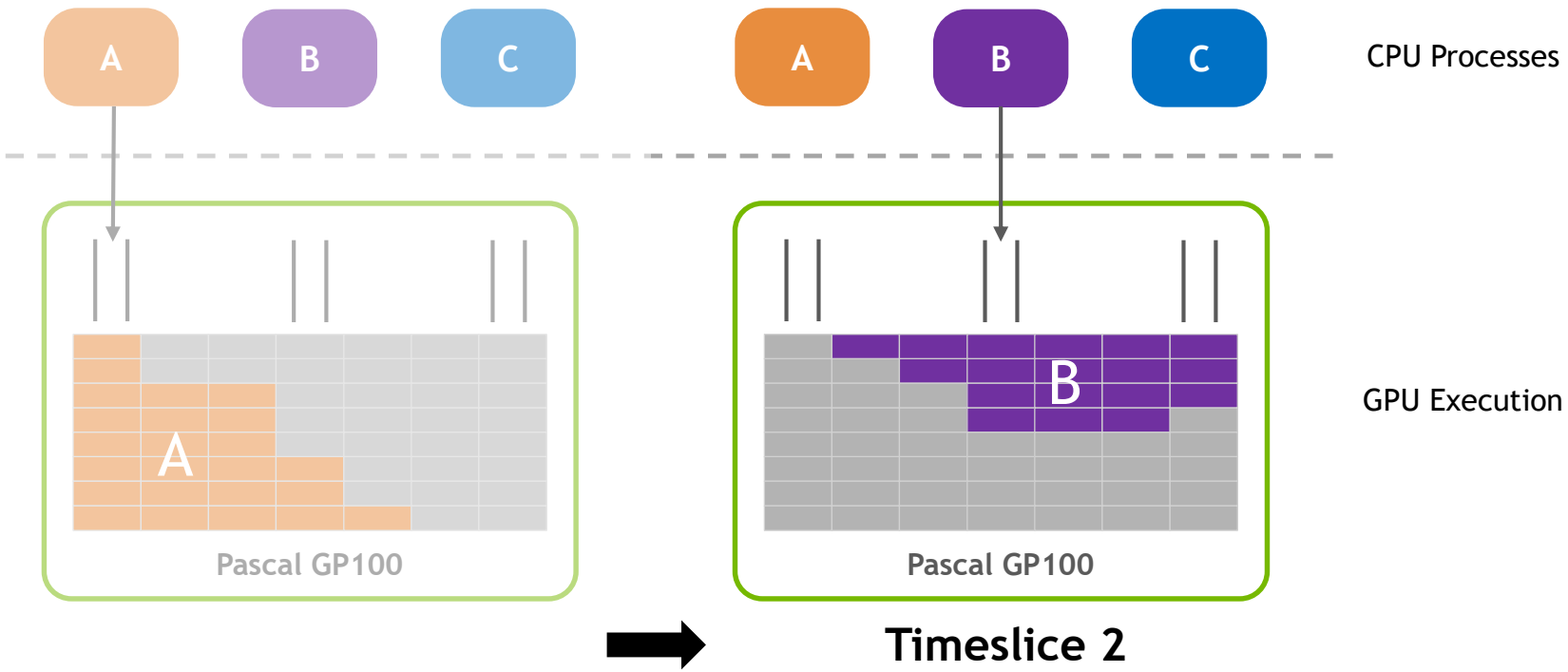
マルチ・プロセス サービス

- 同じ時間帯に、複数プロセスの同時GPU使用を許す
- 全プロセスで考えたときのスループットを重視

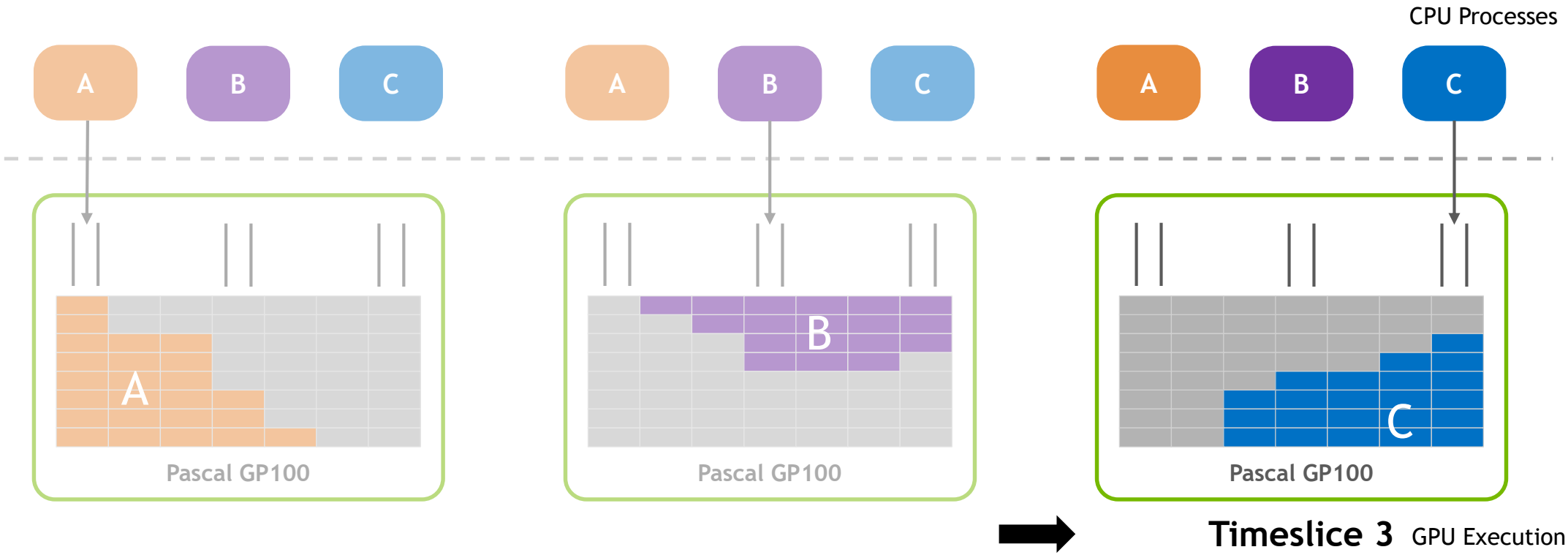
時分割スケジューリング



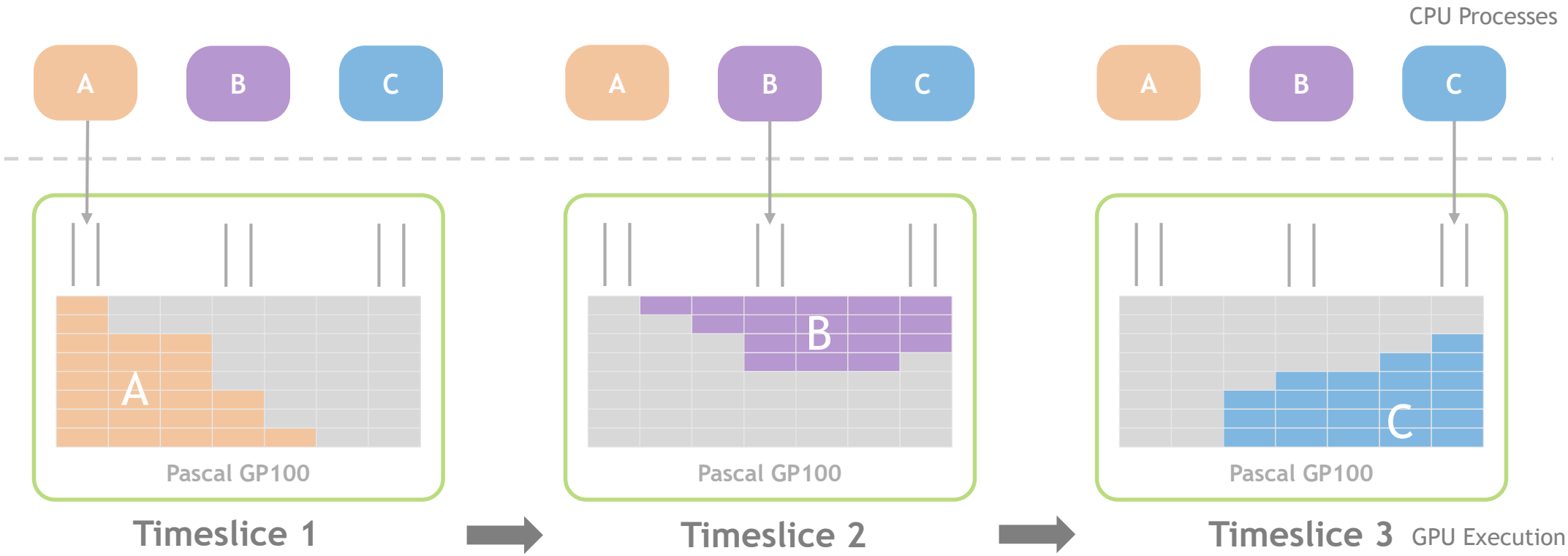
時分割スケジューリング



時分割スケジューリング



時分割スケジューリング



各プロセスのGPU利用率が低ければ、当然、GPU利用率は低いまま

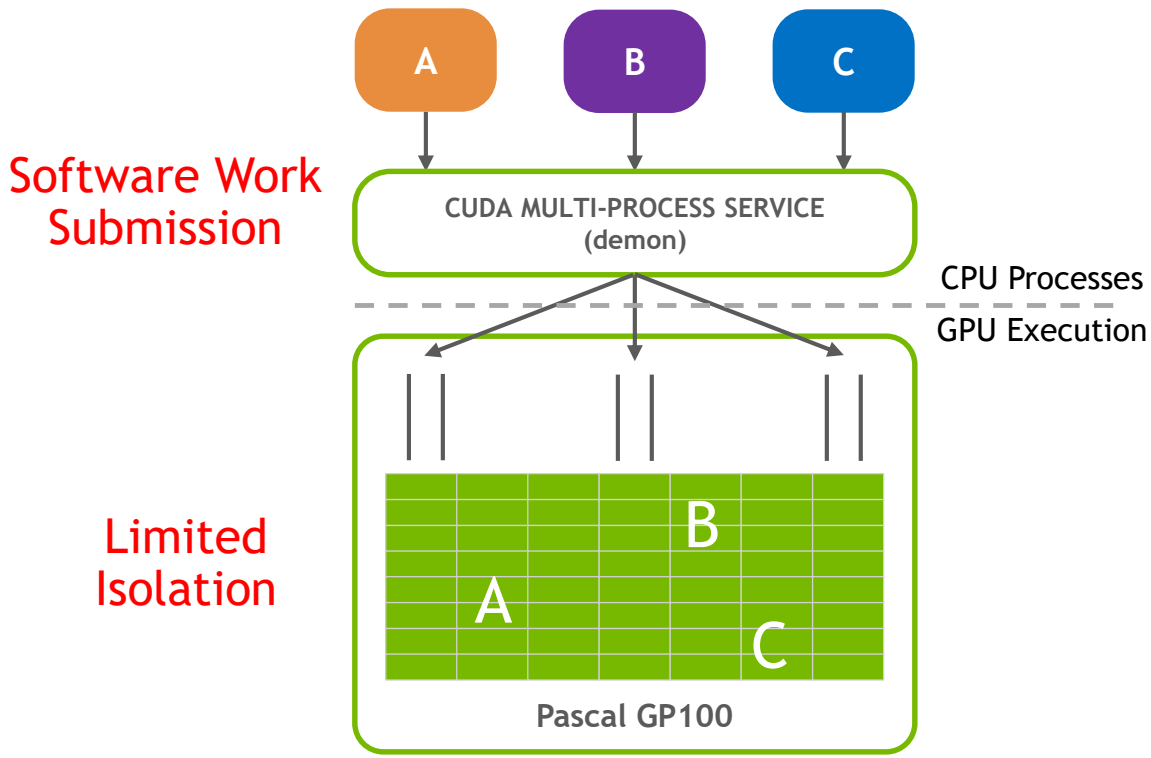
PASCAL: マルチ・プロセス サービス (MPS)

CUDA MPS

- 各プロセスのGPU使用率は低くても、同時にGPUリソースを使用することで、トータルでGPU使用率を高めることができる

Defaultではオフ

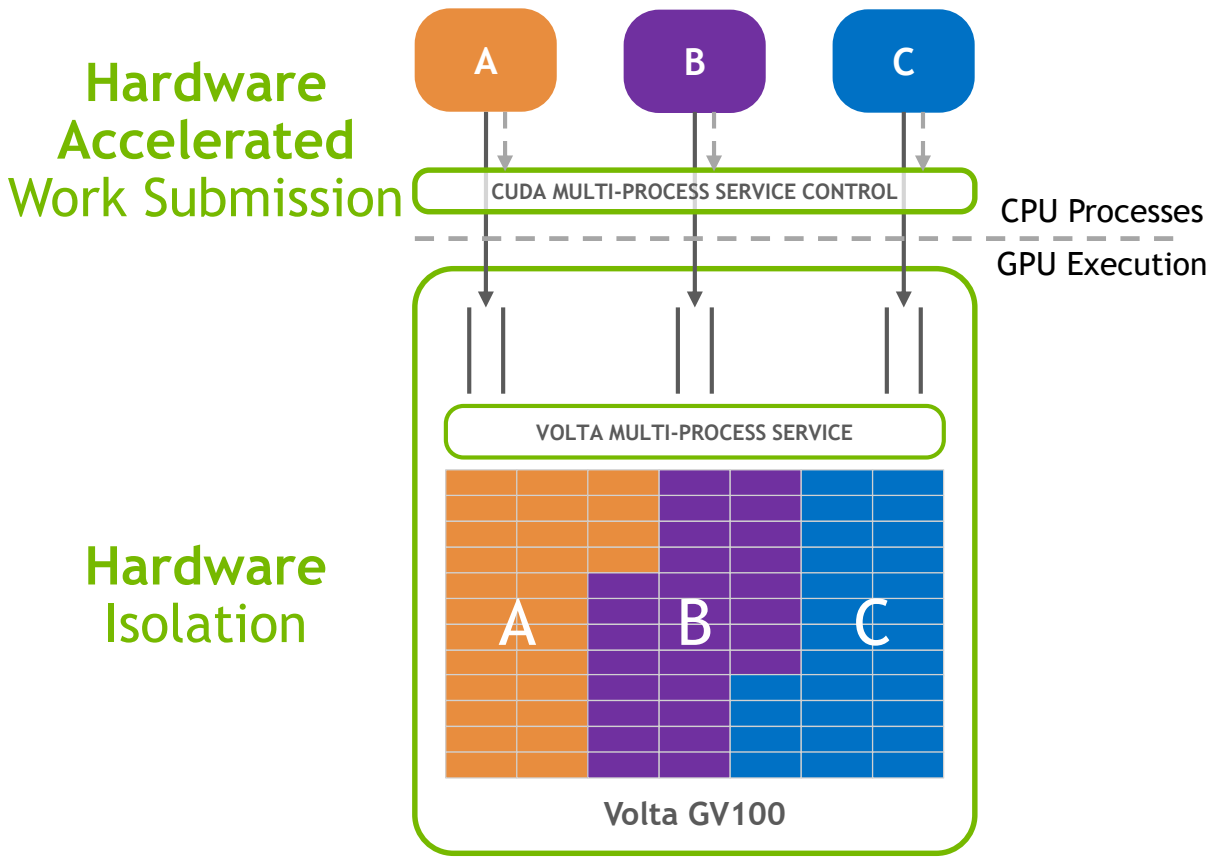
- メモリ保護に制限 (他プロセスのメモリを壊す可能性)



VOLTA: マルチ・プロセス サービス (MPS)

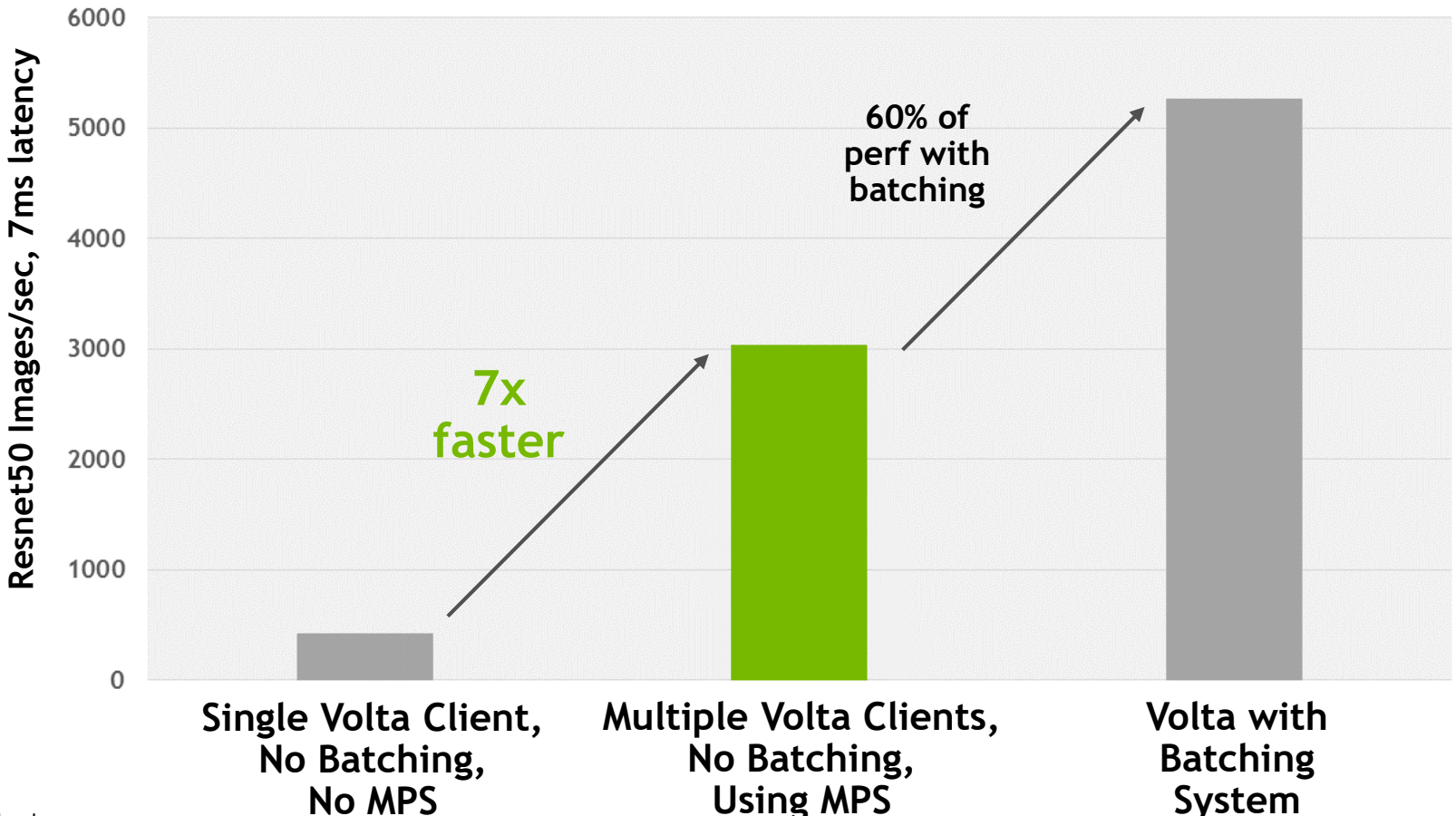
VoltaでMPS改善:

- ハードウェアでメモリ保護 (安全)
- カーネル起動遅延の短縮
- カーネル起動スループットの改善
- スケジューラー分割によるQoS向上
  (性能安定)
- 対応プロセス数の増加
  (Pascal:16 → Volta:48)



VOLTA MPS: インファレンス事例

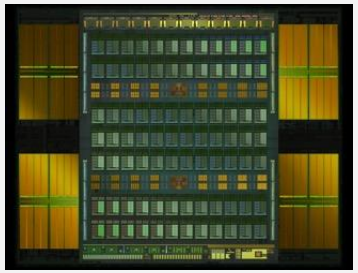
大Batchサイズを使えないケースでも、MPSでスループット向上



V100 measured on pre-production hardware.

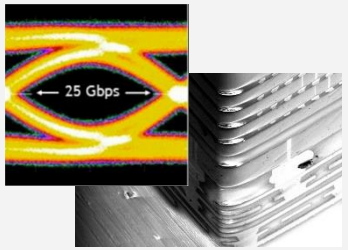
TESLA V100の概要

Volta Architecture



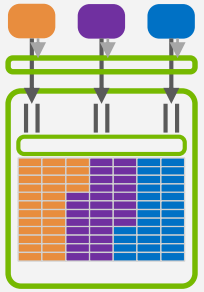
Most Productive GPU

Improved NVLink & HBM2



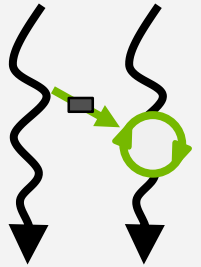
Efficient Bandwidth

Volta MPS



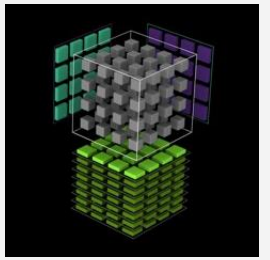
Inference Utilization

Improved SIMT Model



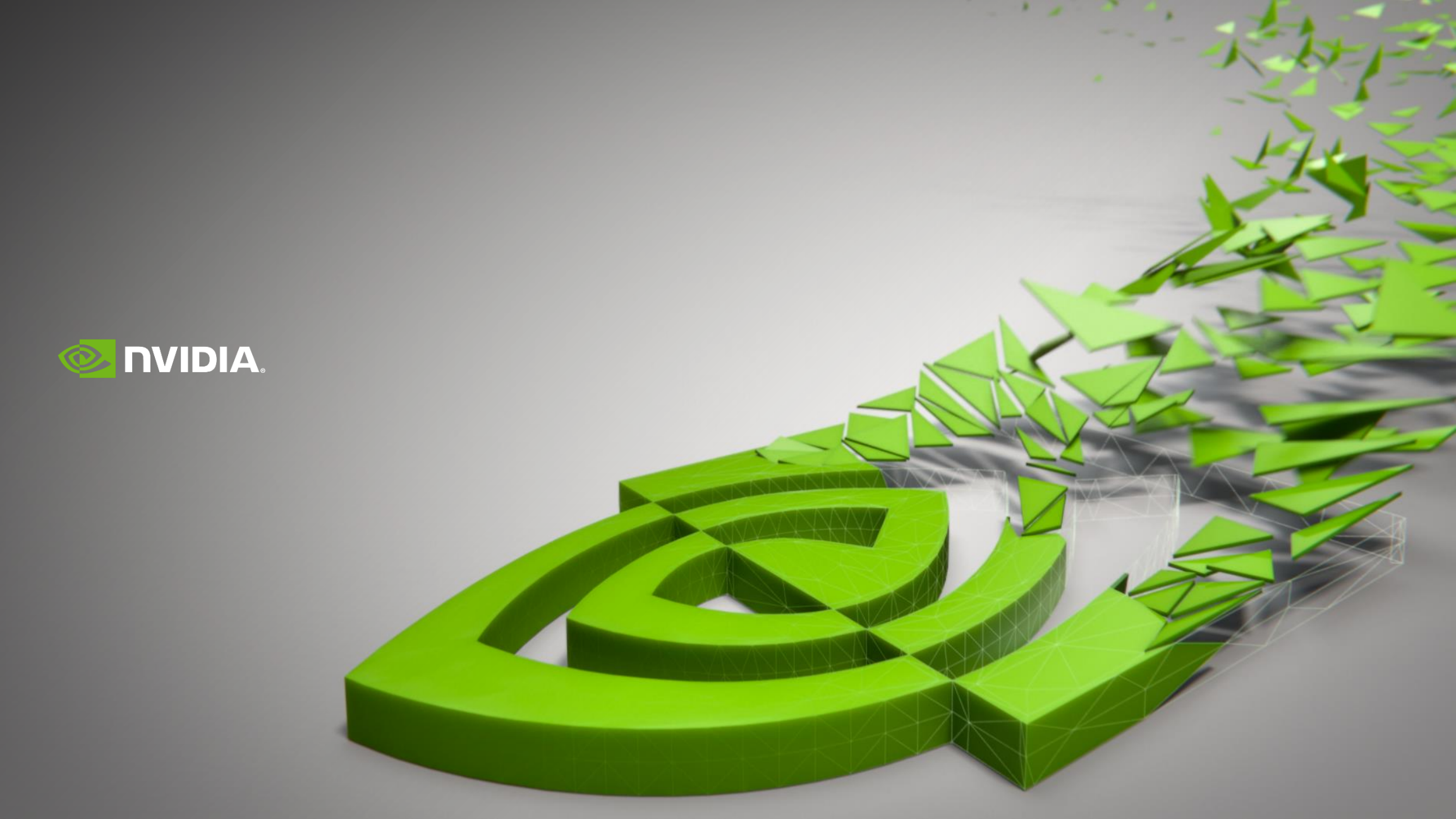
New Algorithms

Tensor Core



125 Programmable TFLOPS Deep Learning

Deep LearningとHPC、両方に最適なGPU


NVIDIA