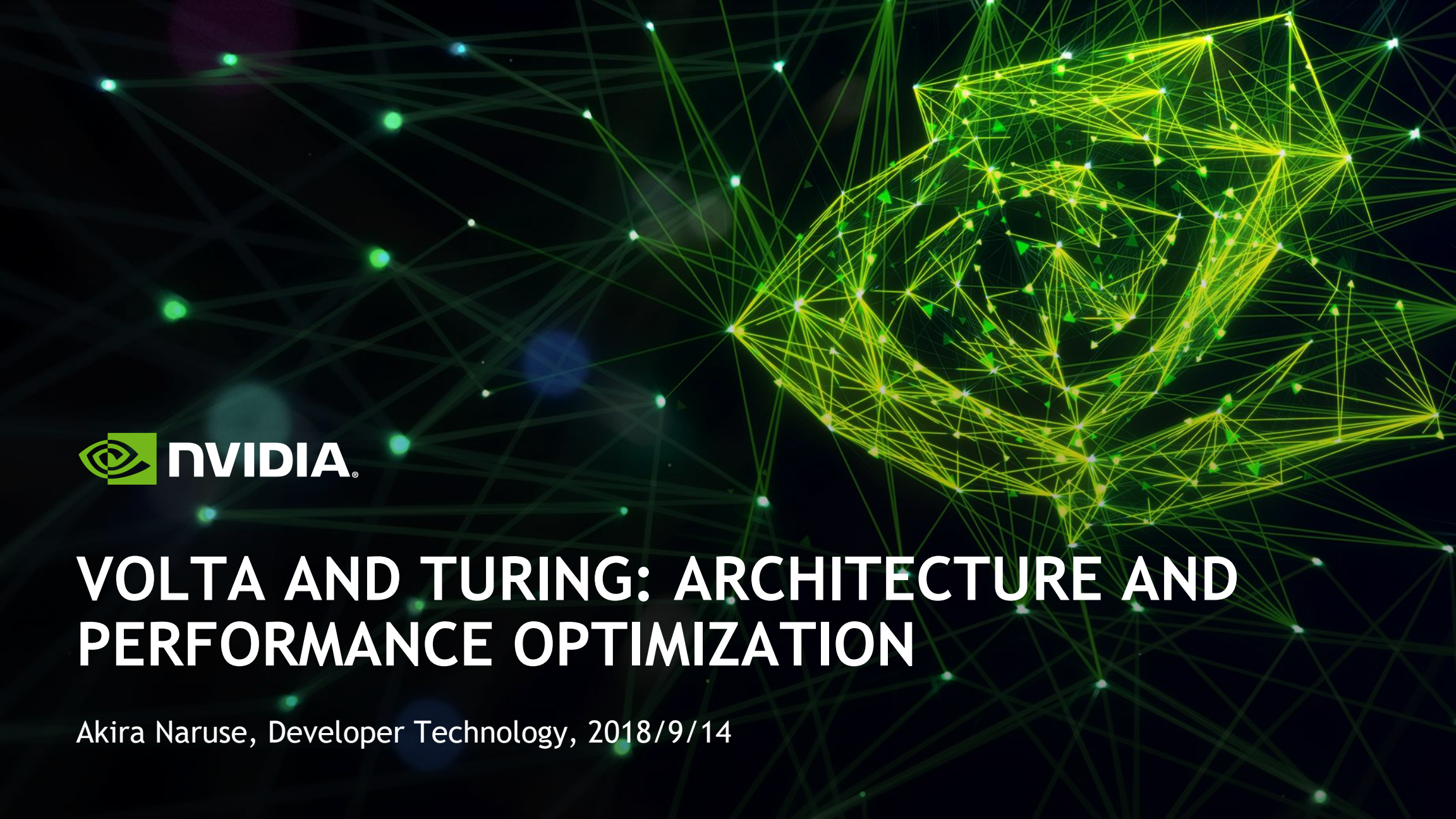


VOLTA AND TURING: ARCHITECTURE AND PERFORMANCE OPTIMIZATION

Akira Naruse, Developer Technology, 2018/9/14

VOLTA AND TURING: ARCHITECTURE

Akira Naruse, Developer Technology, 2018/9/14

VOLTA AND TURING

Pascalから進化、多くの機能を共有

Volta (cc70)

For HPC and Deep Learning

Tesla V100
(GV100)

Turing (cc75)

For Graphics and Deep Learning

QUADRO RTX6000
(TU102)

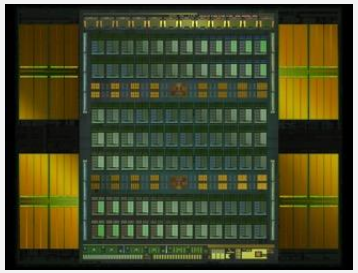
Tesla P100 (GP100)

QUADRO P6000 (GP102)


VOLTA

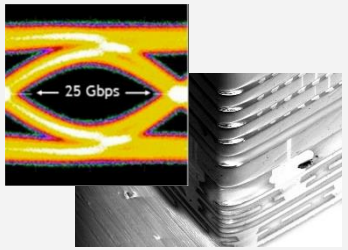
VOLTAの概要

Volta Architecture



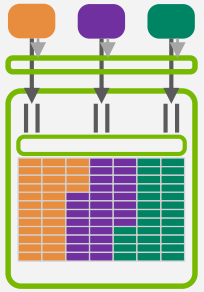
Most Productive GPU

Improved NVLink & HBM2



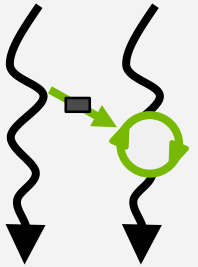
Efficient Bandwidth

Volta MPS



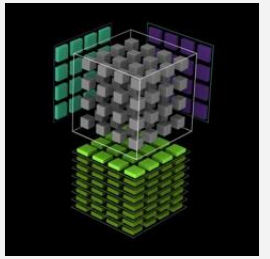
Inference Utilization

Improved SIMT Model



New Algorithms

Tensor Core



125 Programmable TFLOPS Deep Learning

HPCとDeep Learning、両方に最適なGPU

TESLA V100 (GV100)

80 SM
5120 CUDAコア
640 Tensorコア

HBM2
32 GB, 900 GB/s

NVLink 300 GB/s

ピーク性能比較: P100 vs V100

	P100	V100	性能UP
FP32	10 TFLOPS	15.6 TFLOPS	1.5x
FP64	5 TFLOPS	7.8 TFLOPS	1.5x
DLトレーニング	10 TFLOPS (FP32)	125 TFLOPS (Tensorコア)	12.5x
HBM2バンド幅	720 GB/s	900 GB/s	1.2x
L2キャッシュ	4 MB	6 MB	1.5x
NVLinkバンド幅	160 GB/s (4リンク)	300 GB/s (6リンク)	1.9x

VOLTA GV100 SM

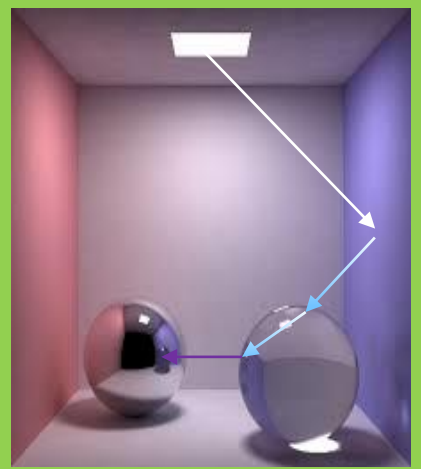
	GV100
INT32	64
FP32	64
FP64	32
Tensorコア	8
レジスターファイル	256 KB
L1キャッシュ/共有メモリ	128 KB
最大スレッド数	2048

NVIDIA


TURING

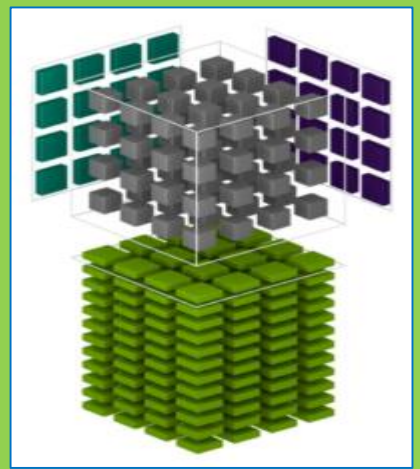
TURINGの概要

リアルタイム
レイトレーシング



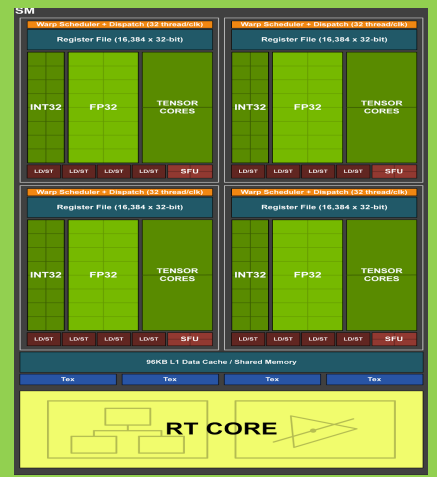
RTコア

ディープラーニング
高速化



TENSORコア

進化した
プログラマブルシェーダー



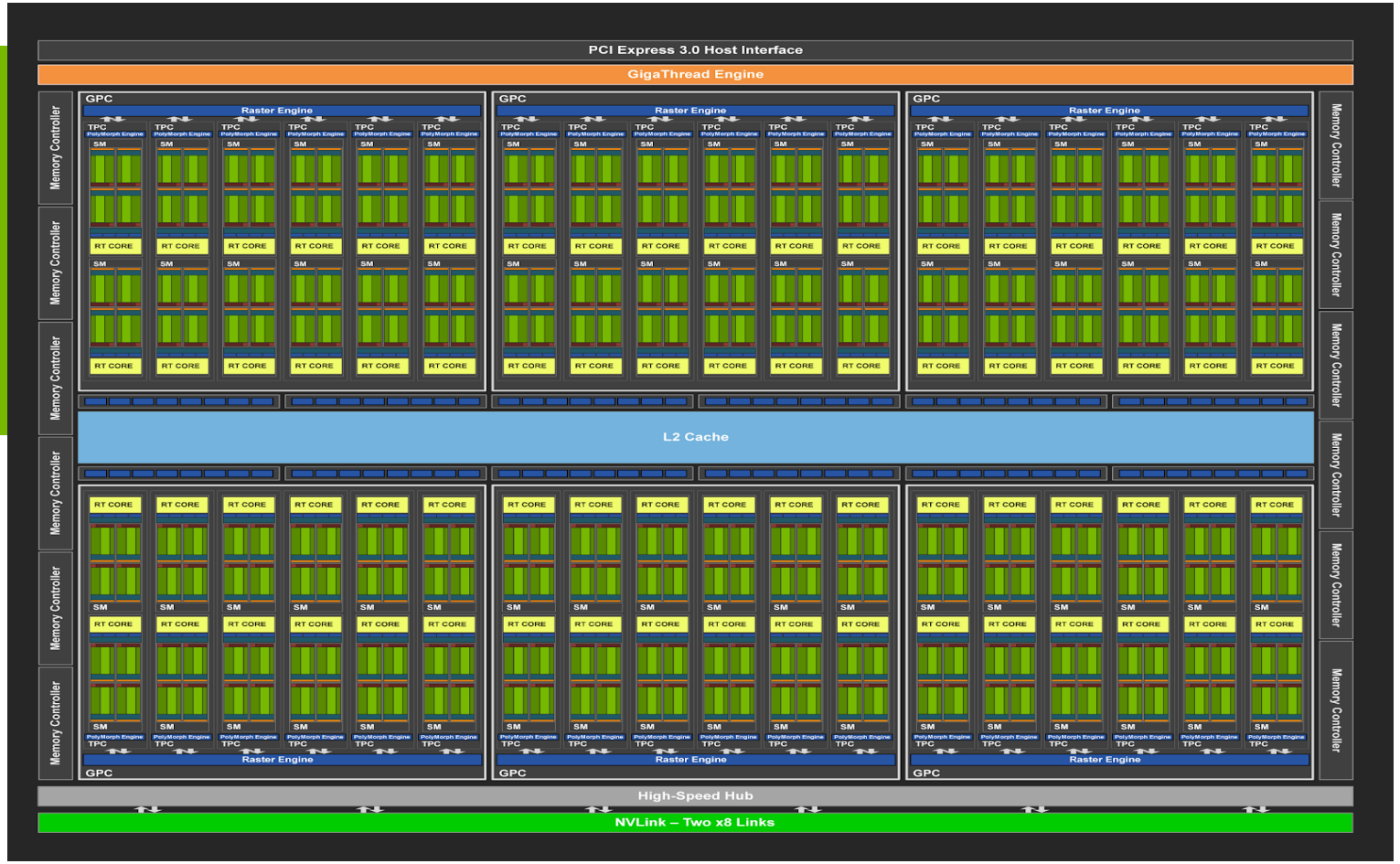
STREAMING MULTIPROCESSOR

QUADRO RTX6000 (TU102)

72 SM
4608 CUDAコア
576 Tensorコア
72 RTコア

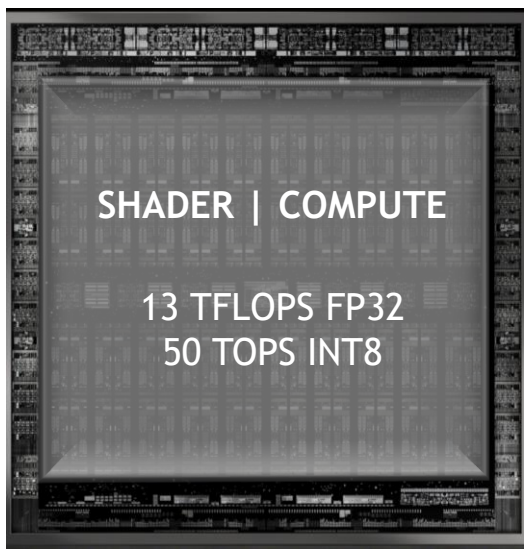
DDR6
24 GB, 672 GB/s

NVLink 100 GB/s



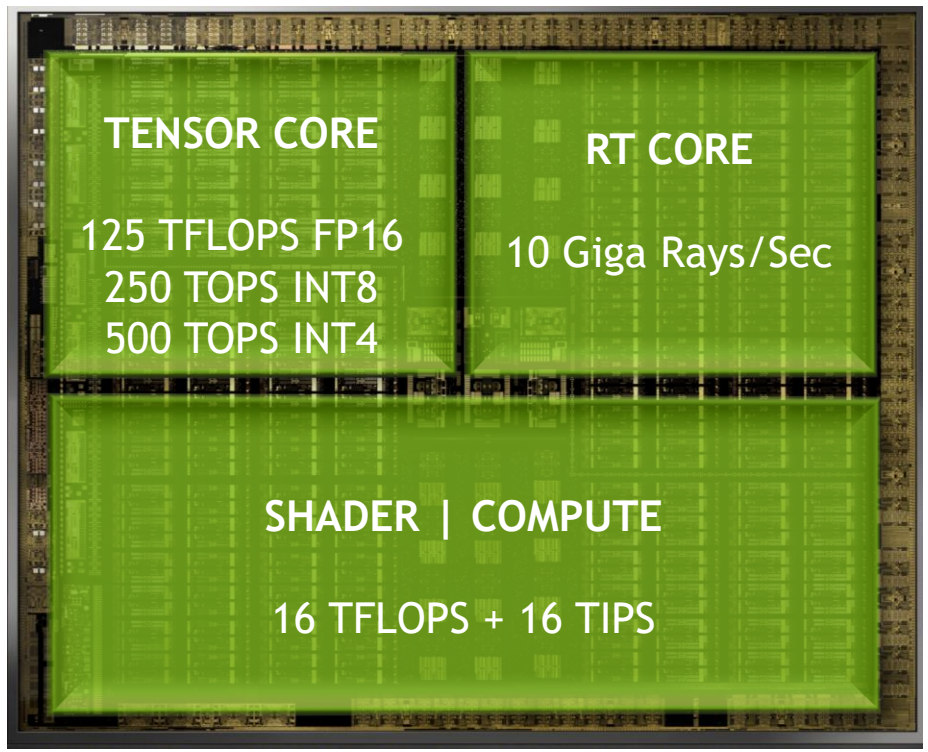
大きな進化: GP102 → TU102

PASCAL (GP102)



11.8 Billion xstr | 471 mm^2 | 24 GB 10GHz

TURING (TU102)



18.6 Billion xstr | 754 mm^2 | up to 48+48 GB 14GHz

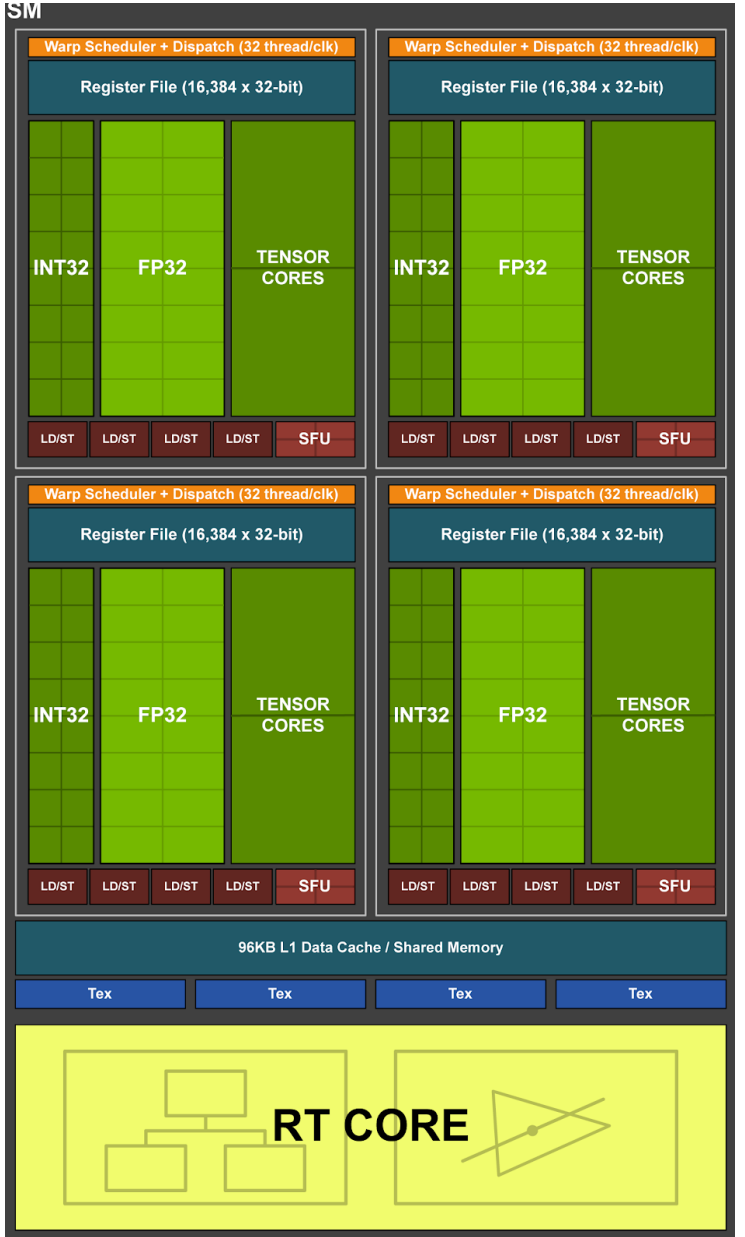
ピーク性能比較: P6000 vs RTX6000

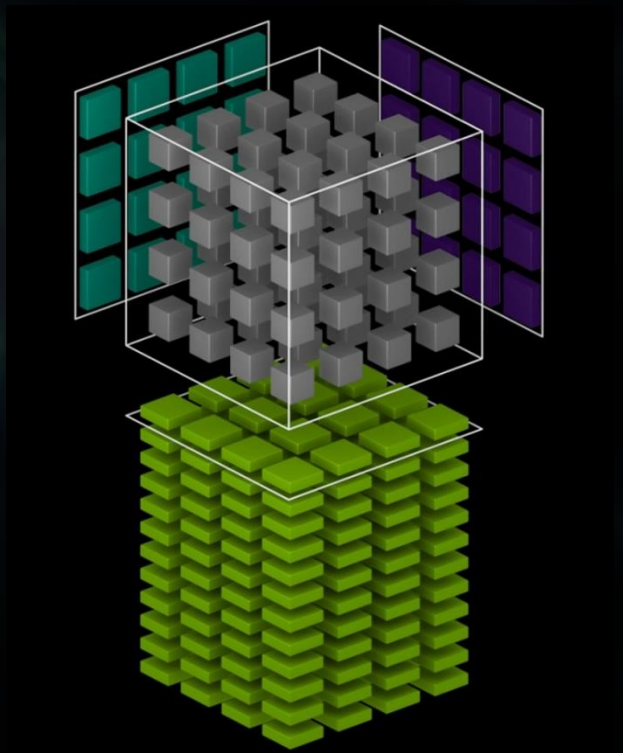
	P6000	RTX6000 (*)	性能UP
FP32	12.5 TFLOPS	15.6 TFLOPS	1.2x
DLインファレンス (FP16)	NA	125 TFLOPS (TensorCore)	
DLインファレンス (INT8)	50 TOPS (DP4A)	250 TOPS (TensorCore)	5.0x
DRAMバンド幅	432 GB/s (GDDR5X)	672 GB/s (GDDR6)	1.6x
L2キャッシュ	3 MB	6 MB	2.0x
NVLinkバンド幅	NA	100 GB/s (2リンク)	

(*) RTX6000はクロック設定が変更の可能性有り

TURING TU102 SM

	TU102
INT32	64
FP32	64
Tensorコア	8
RTコア	1
レジスターファイル	256 KB
L1キャッシュ/共有メモリ	96 KB
最大スレッド数	1024





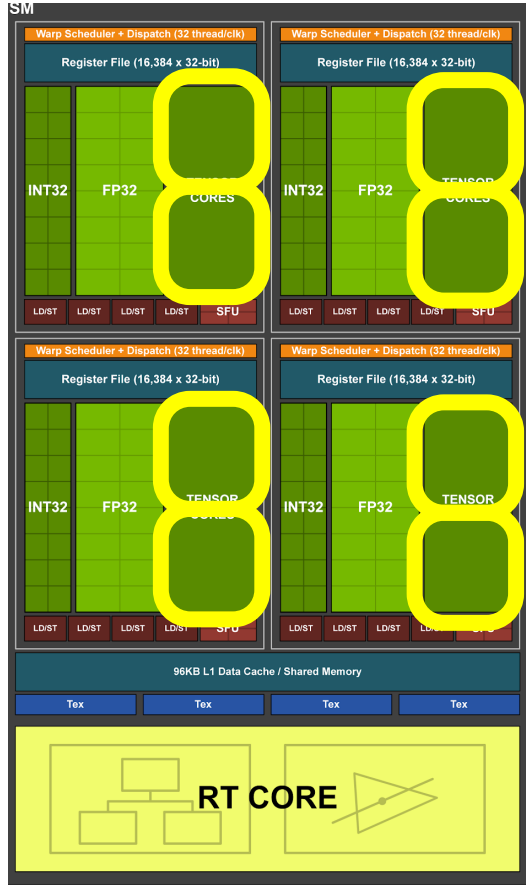
TENSOR CORES

TENSORコア

Volta SM

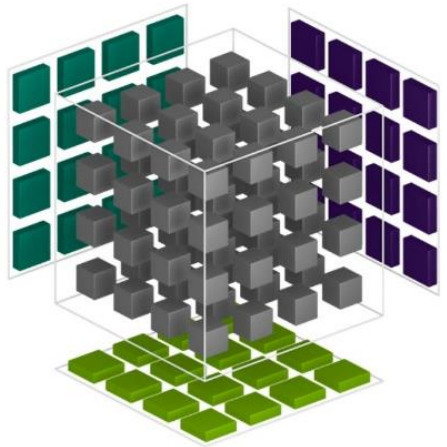
8 Tensorコア / SM

Turing SM



NVIDIA

TENSORコア
混合精度行列演算ユニット



行列のFMA (Fused Multiply-Add)

4x4の行列の積和演算を1サイクルで計算する性能:

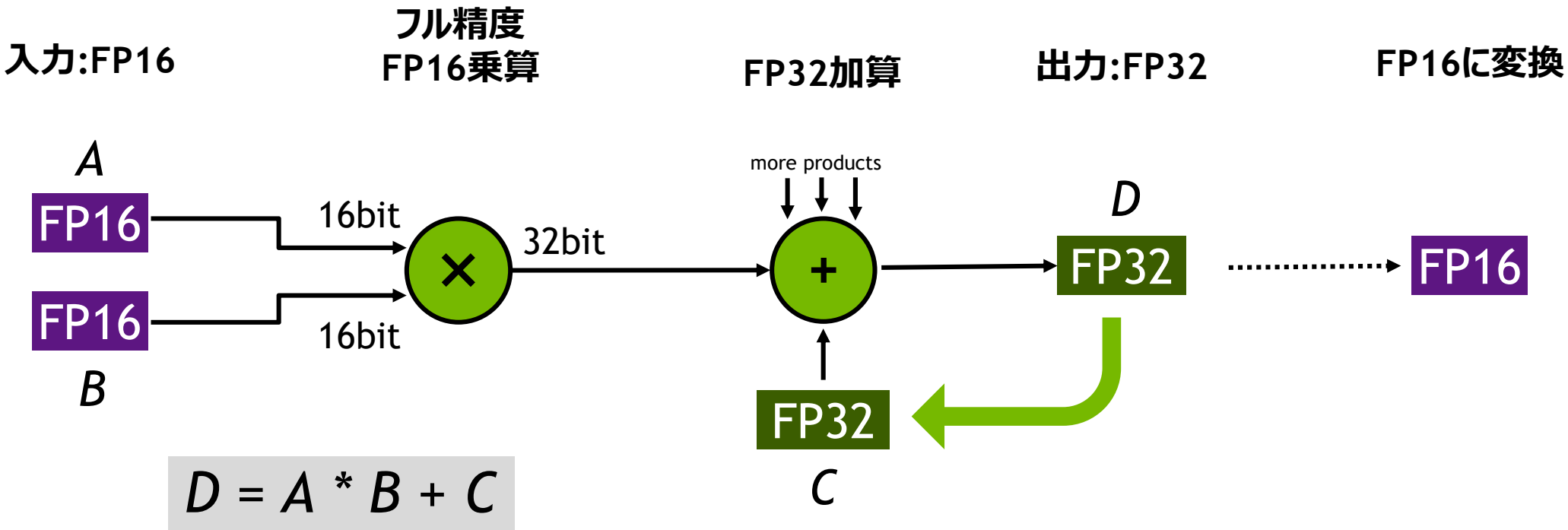
128演算/サイクル/Tensorコア、1024演算/サイクル/SM

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

FP32 (FP16) FP16 FP16 FP32 (FP16)

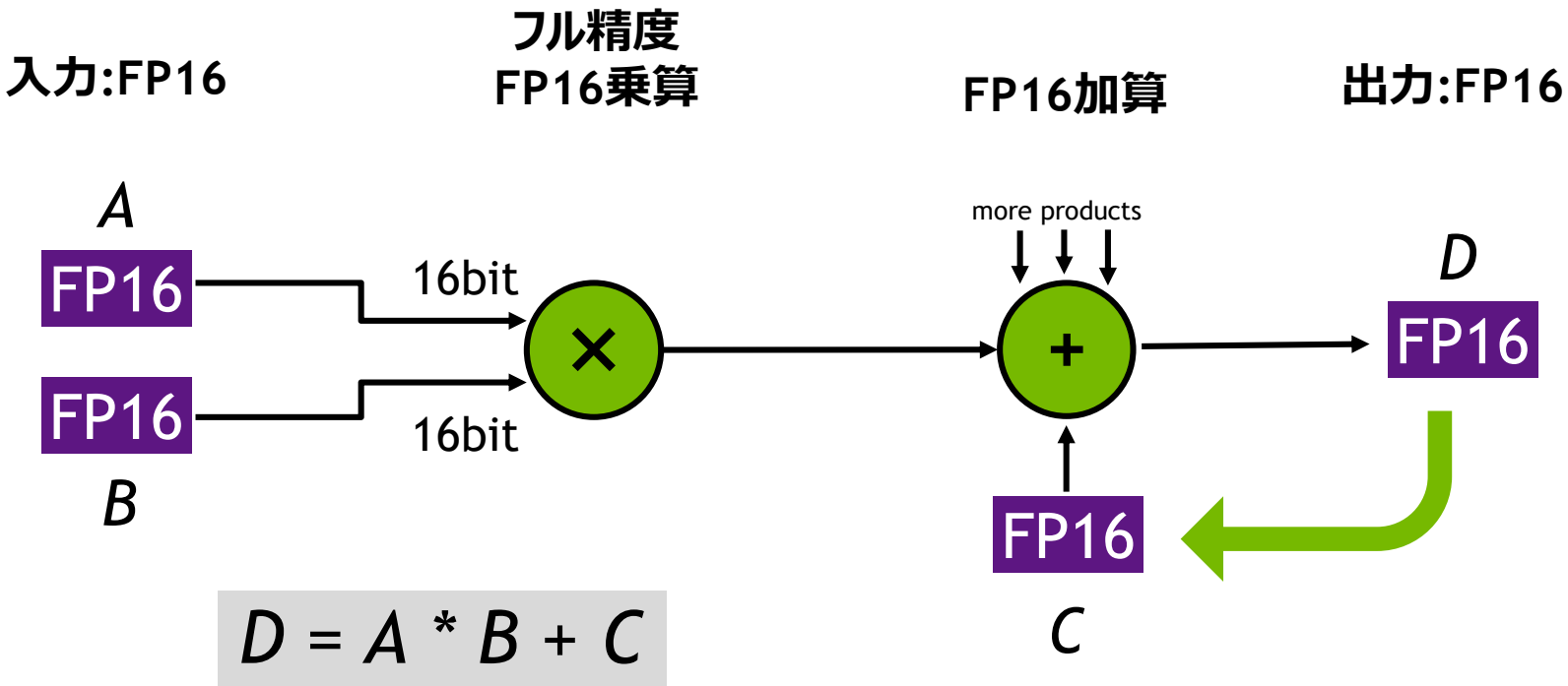
TENSOR演算 (FP16)

VOLTA, TURING



TENSOR演算 (FP16)

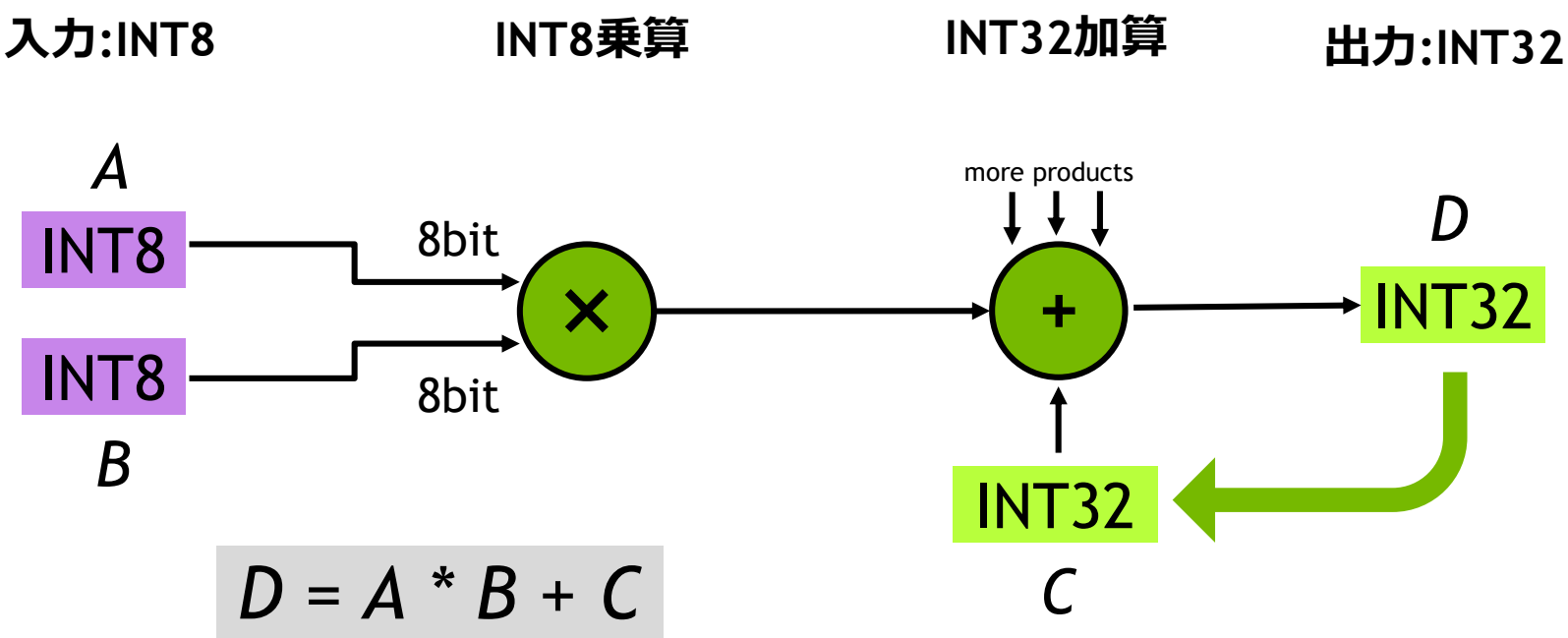
VOLTA, TURING



FP16加算もサポート (インファレンス用)

TENSOR演算 (INT8)

TURING

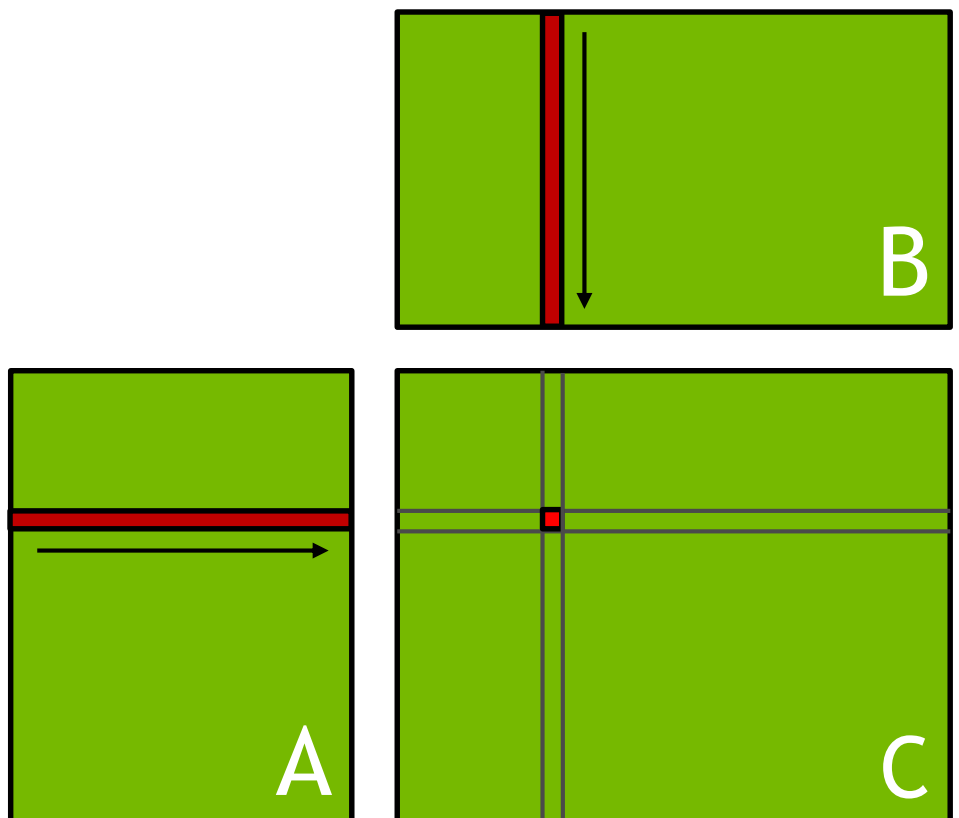


TENSORコアは何のため?

大きな行列積 (行列と行列の乗算)

- $O(N^3)$
- Deep Learningで典型的な計算
  - トレーニング (FP32, FP16)
  - インファレンス (FP16, INT8)

cuBLAS:密行列演算ライブラリ



CUBLASはTENSORコア対応

例: cuBLAS cublasGemmEx (FP16)

Tensorコア使用モードを選択

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

algo = CUBLAS_GEMM_DEFAULT_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

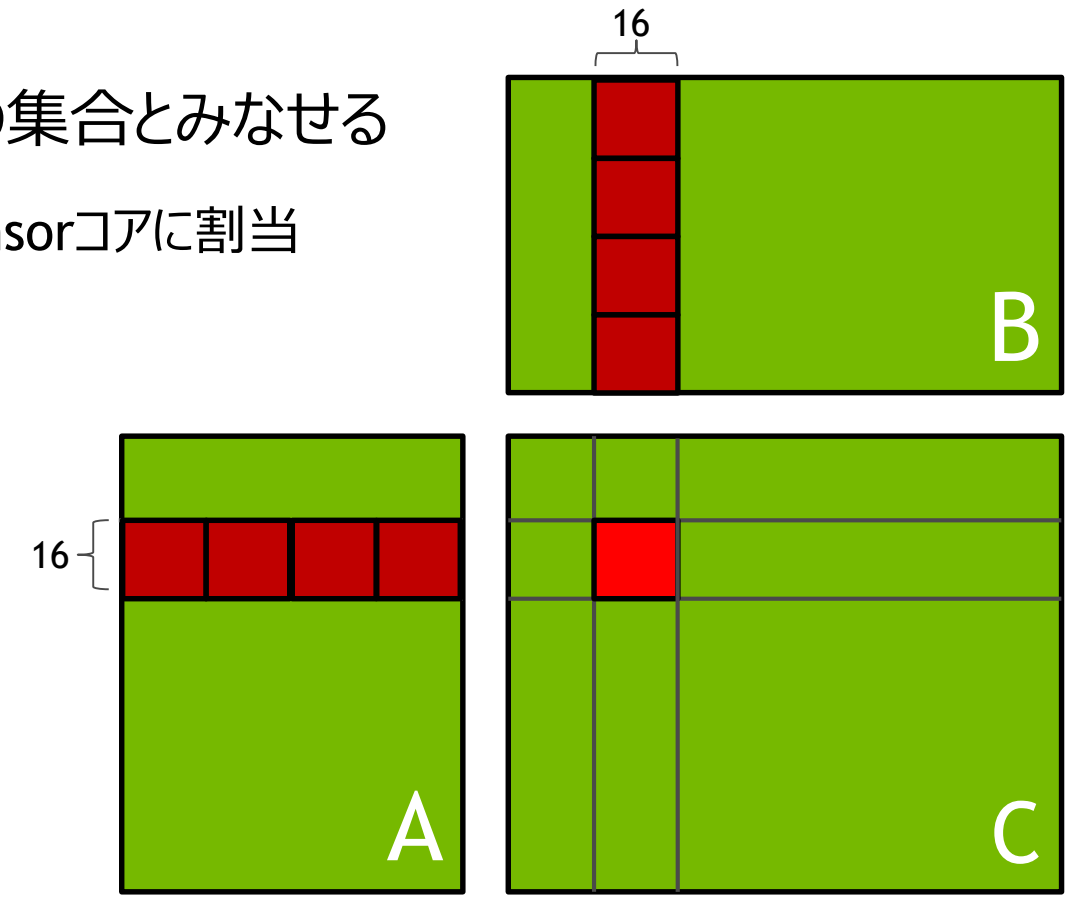
出力行列Cのデータ型を指定

計算型を指定

TENSORコアはどう使われているの?

大きな行列積は、小さな行列積の集合とみなせる

- 多くの16x16行列積を、それぞれ、Tensorコアに割当



TENSORコアの使い方 (FP16)

CUDA WMMA API

```
__device__ void tensor_op_16_16_16(half *a, half *b, float *c)
{
   wmma::fragment<wmma::matrix_a, 16, 16, 16, half, …> a_frag;
   wmma::fragment<wmma::matrix_b, 16, 16, 16, half, …> b_frag;
   wmma::fragment<wmma::accumulator, 16, 16, 16, float, …> c_frag;

   wmma::load_matrix_sync(a_frag, a, …);
   wmma::load_matrix_sync(b_frag, b, …);

   wmma::fill_fragment(c_frag, 0.0f);

   wmma::mma_sync(c_frag, a_frag, b_frag, c_frag);

   wmma::store_matrix_sync(c, c_frag, …);
}
```

Tensorコアへの入出力データ型(fragment)の宣言

入力行列の一部を入力fragmentに読み込み

出力fragmentを初期化

Tensorコア演算

出力fragmentを出力行列に書き込み

TENSORコアの使い方 (INT8, CUDA10, TURING)

CUDA WMMA API

```
__device__ void tensor_op_16_16_16(char *a, char *b, int *c)
{
    wmma::fragment<wmma::matrix_a, 16, 16, 16, char, …> a_frag;
    wmma::fragment<wmma::matrix_b, 16, 16, 16, char, …> b_frag;
    wmma::fragment<wmma::accumulator, 16, 16, 16, int, …> c_frag;

    wmma::load_matrix_sync(a_frag, a, …);
    wmma::load_matrix_sync(b_frag, b, …);

    wmma::fill_fragment(c_frag, 0);

    wmma::mma_sync(c_frag, a_frag, b_frag, c_frag);

    wmma::store_matrix_sync(c, c_frag, …);
}
```

TENSORコアのピーク性能比較

VoltaとTuring

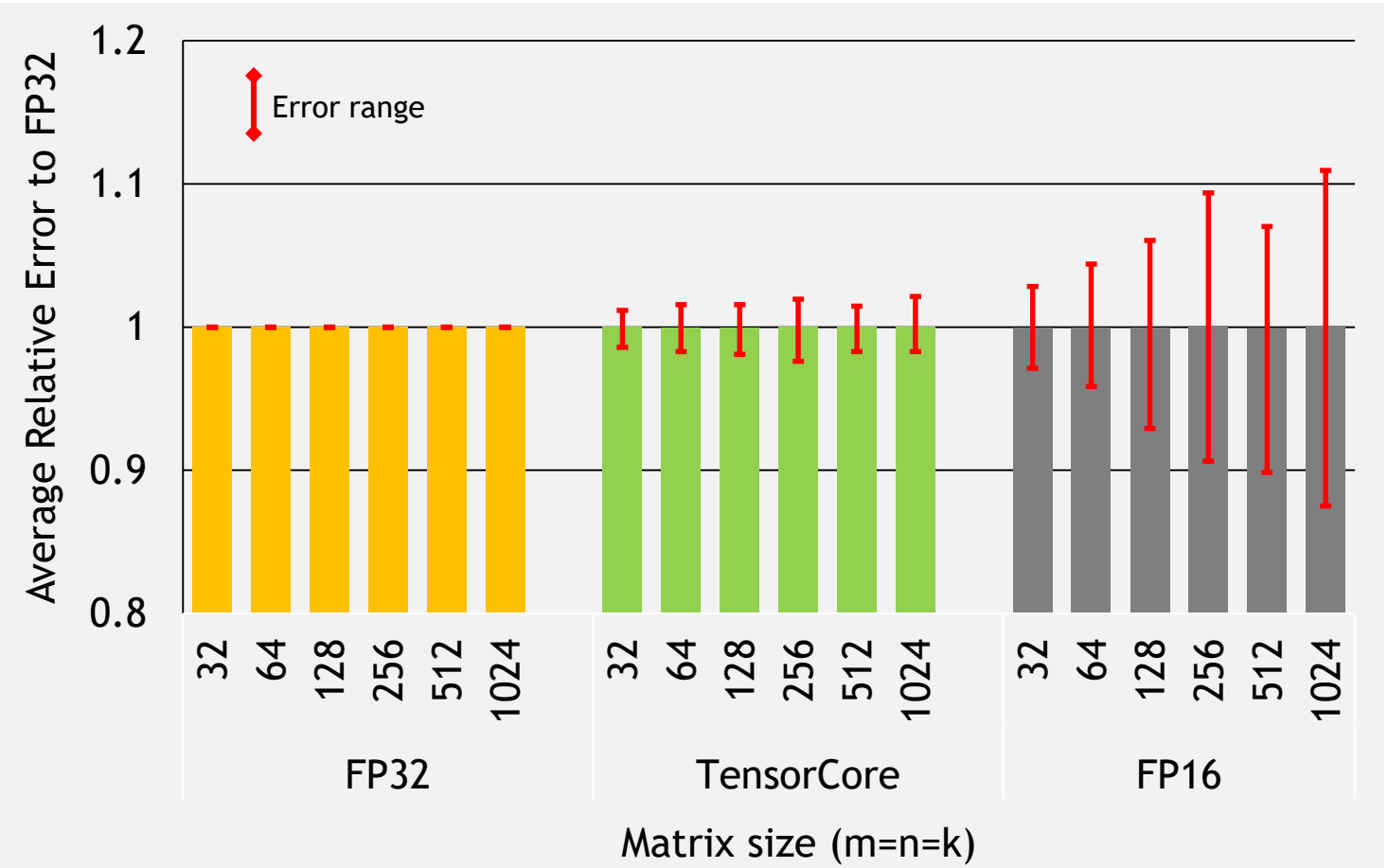
	TESLA V100 (GV100)	QUADRO RTX6000 (*2) (TU102)
FP16	125 TFLOPS	125 TFLOPS
INT8	NA	250 TOPS
INT4 (*1)	NA	500 TOPS

Turing TensorCoreは実験用としてINT4とINT1もサポート
詳しくは「2018-2052 CUDA10の新機能とその性能」にて

(*1) INT4: 実験用　　(*2) RTX6000はクロック設定が変更の可能性有り

TENSORコア計算の「誤差」は?

FP16より、FP32に近い



Note: depends on application!

C = A * B

- Matrix A: exponential distribution (activation)
- Matrix B: gaussian distribution (weight)
- Error range: 99%

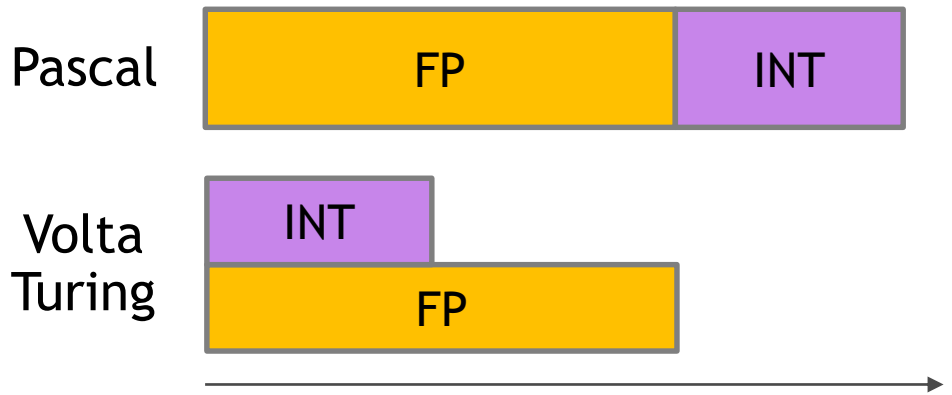

STREAM MULTIPROCESSORS

FP演算とINT演算を同時実行

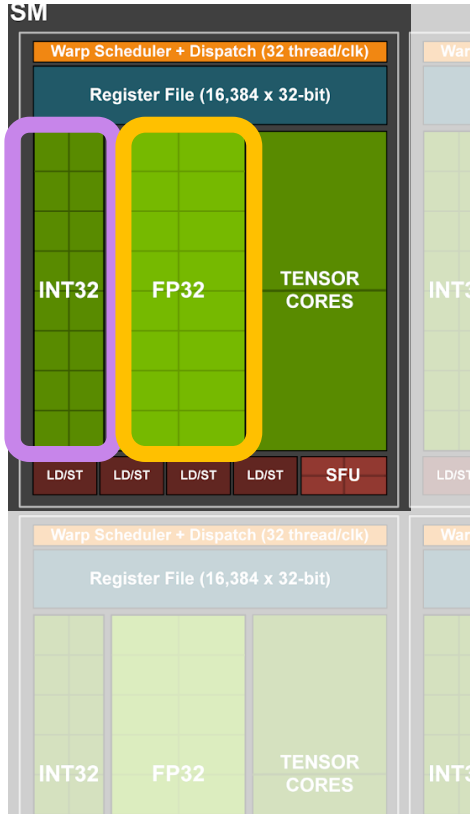
Volta SM

Volta/Turingから、

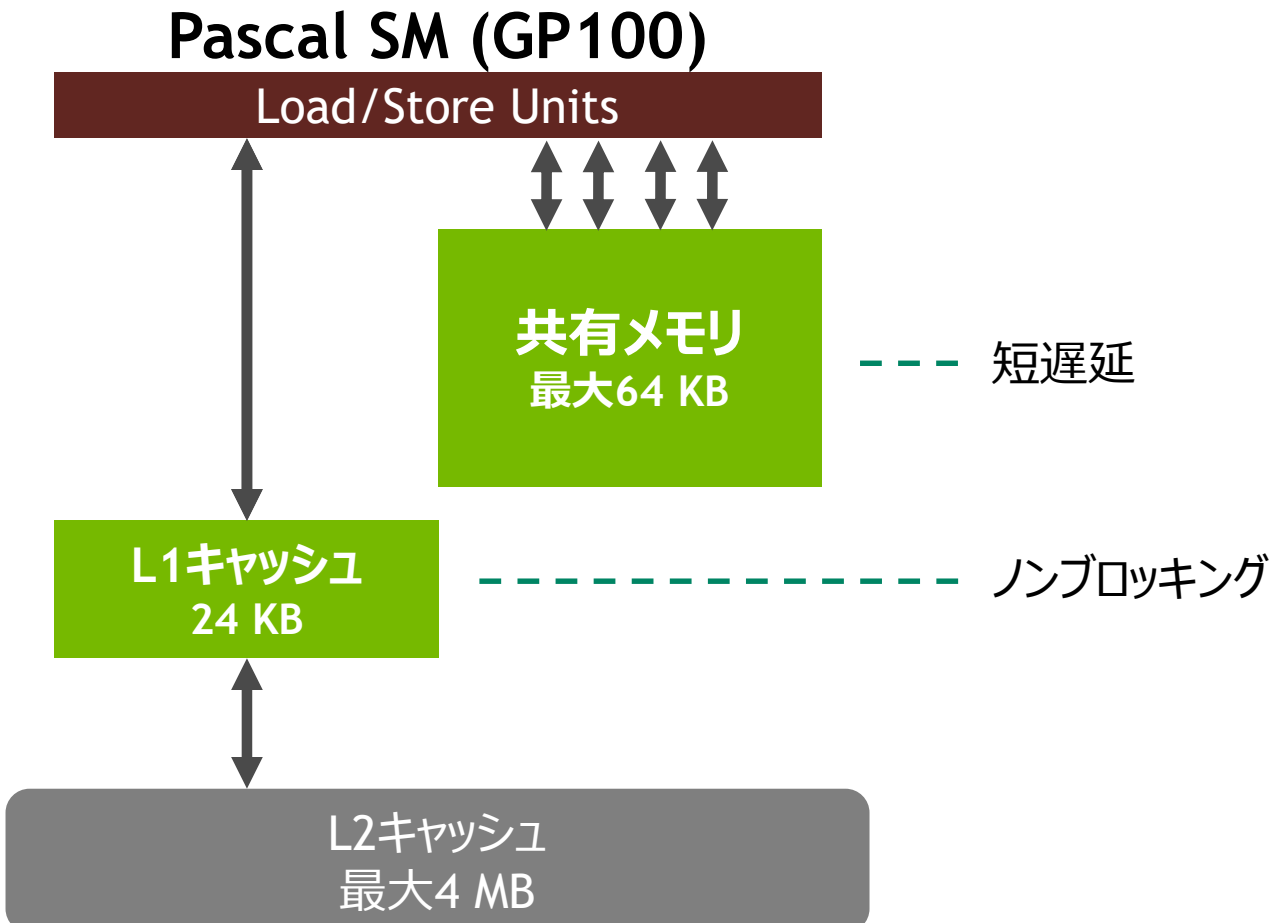
- FP32ユニットとINT32ユニットを分離
- FP演算とINT演算の同時実行が可能



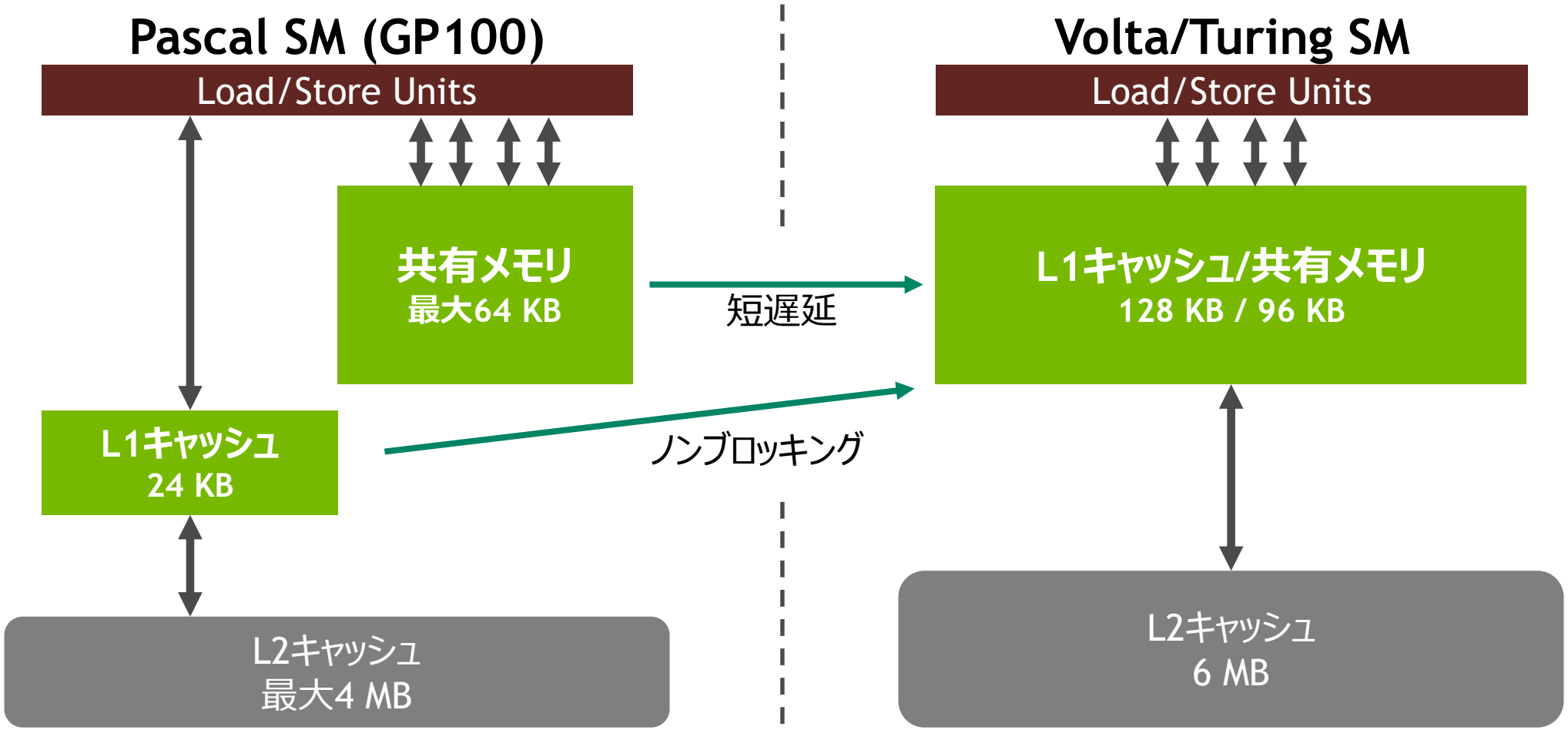
Turing SM



PASCAL: L1キャッシュと共有メモリは分離



統合L1キャッシュ/共有メモリ



統合L1キャッシュ/共有メモリ

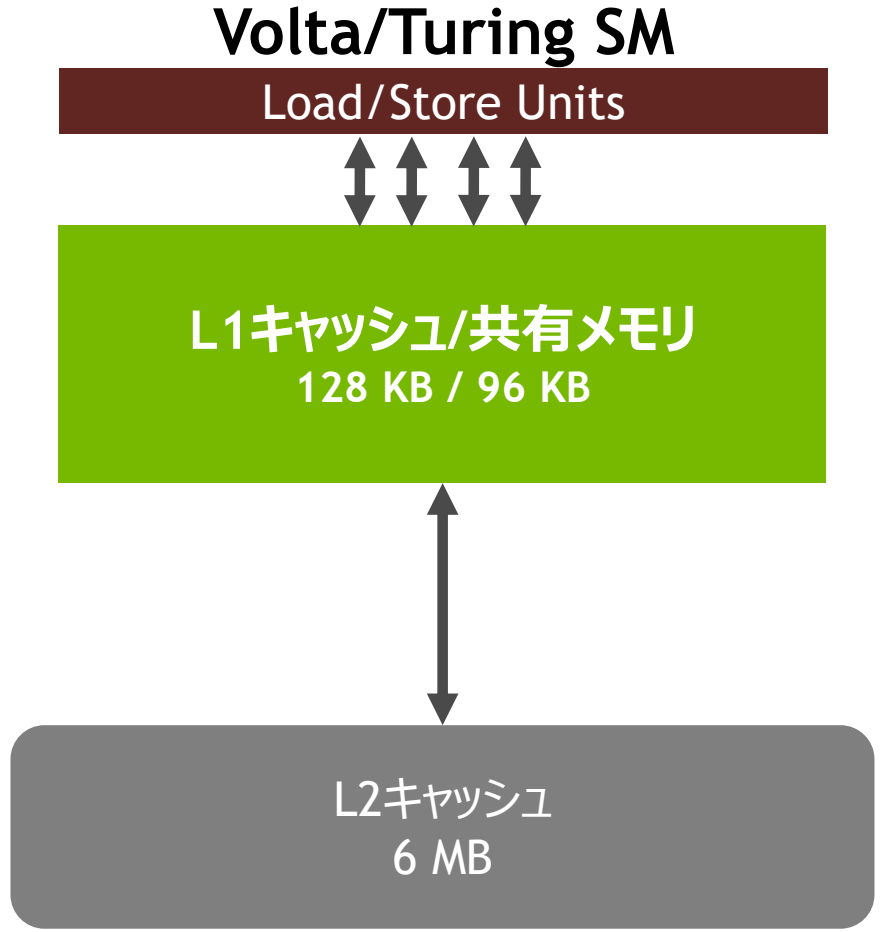
Pascalと比べて、

L1キャッシュ:

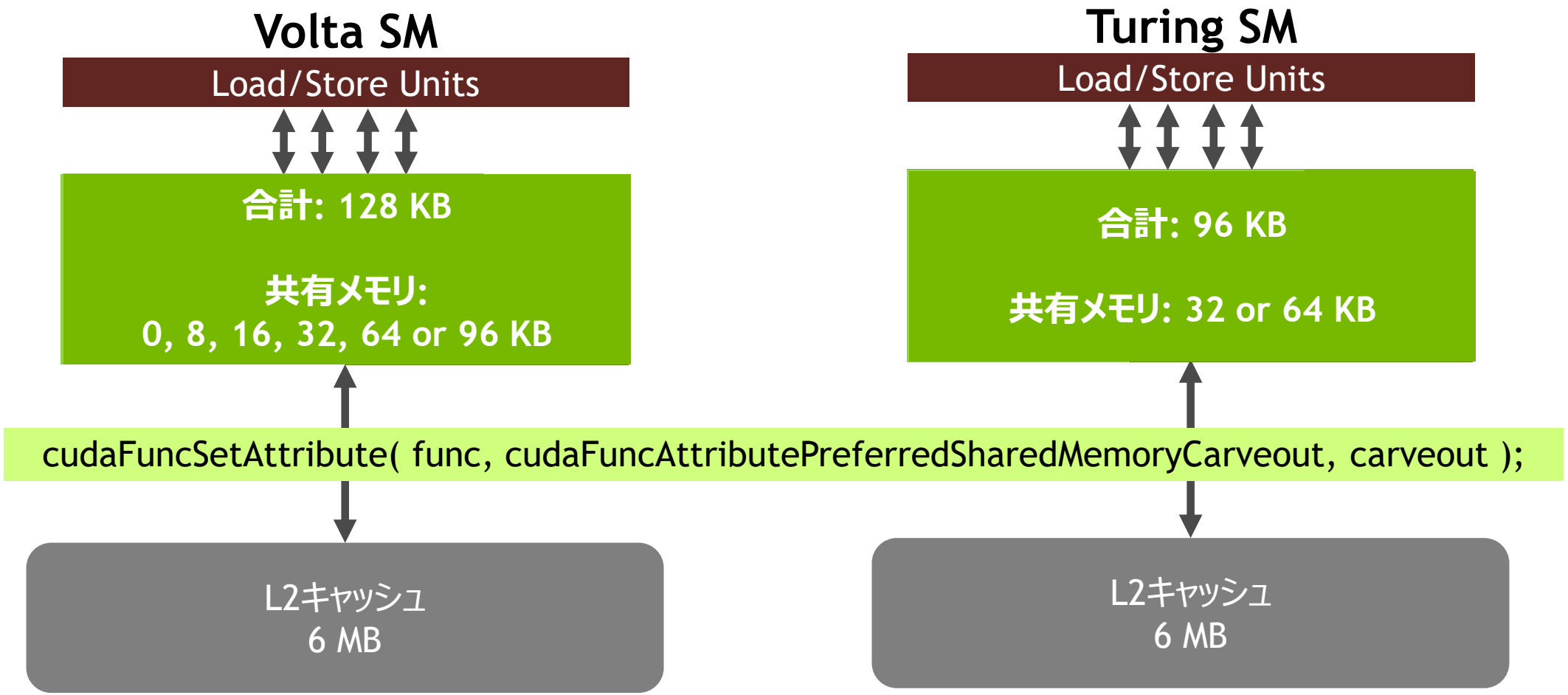
- 2倍以上の容量 (ヒット率UP)
- 2倍のバンド幅
- 短い遅延

L2

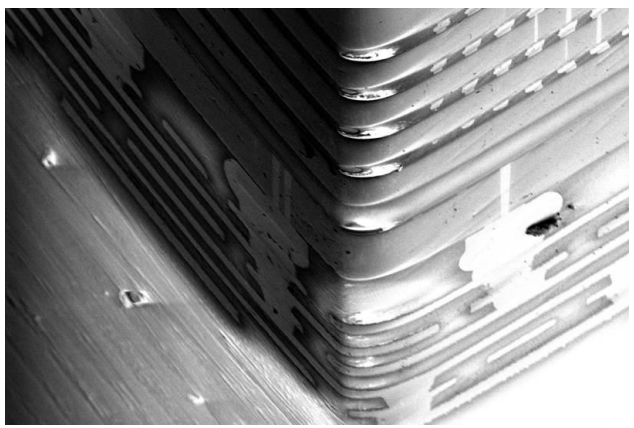
- 2倍の容量 (ヒット率UP)



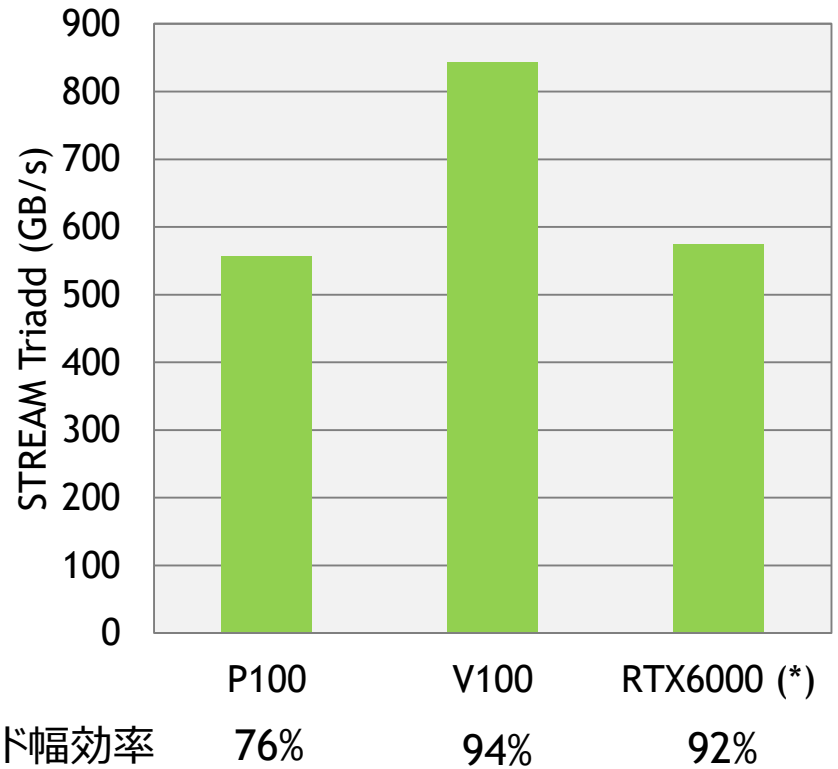
共有メモリサイズは設定可能

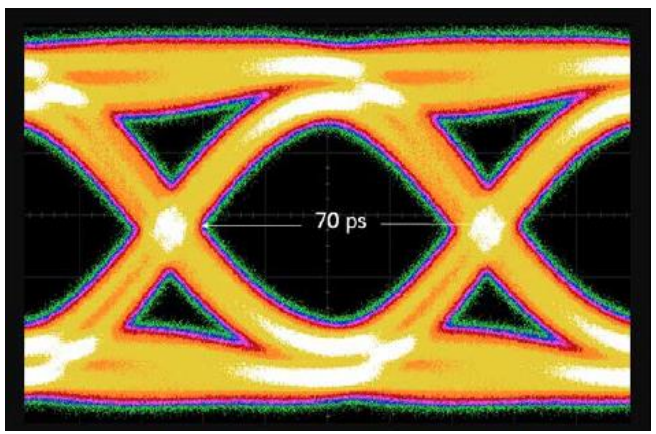


実効メモリバンド幅



HBM2
P100 and V100





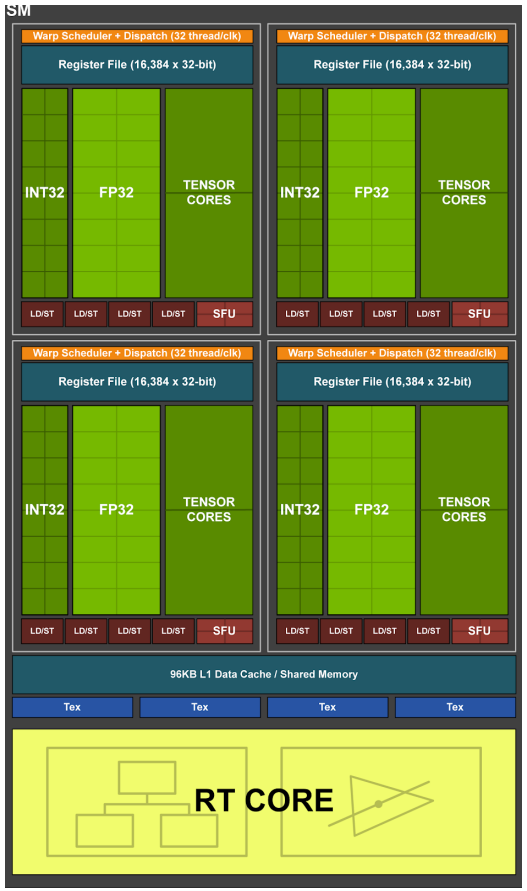
GDDR6
RTX6000

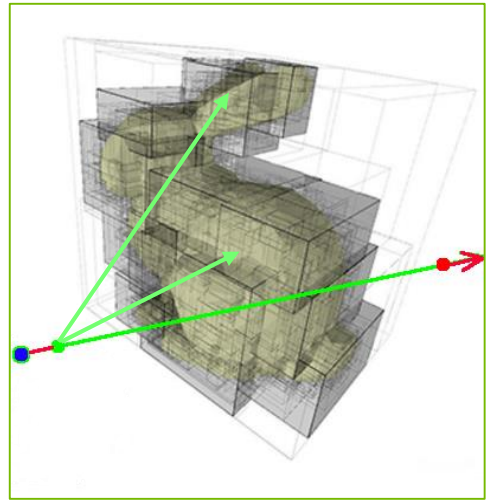
(*) データレート1.3GHzの開発ボード、ECC Off


RT CORE

RTコア

Turing SM





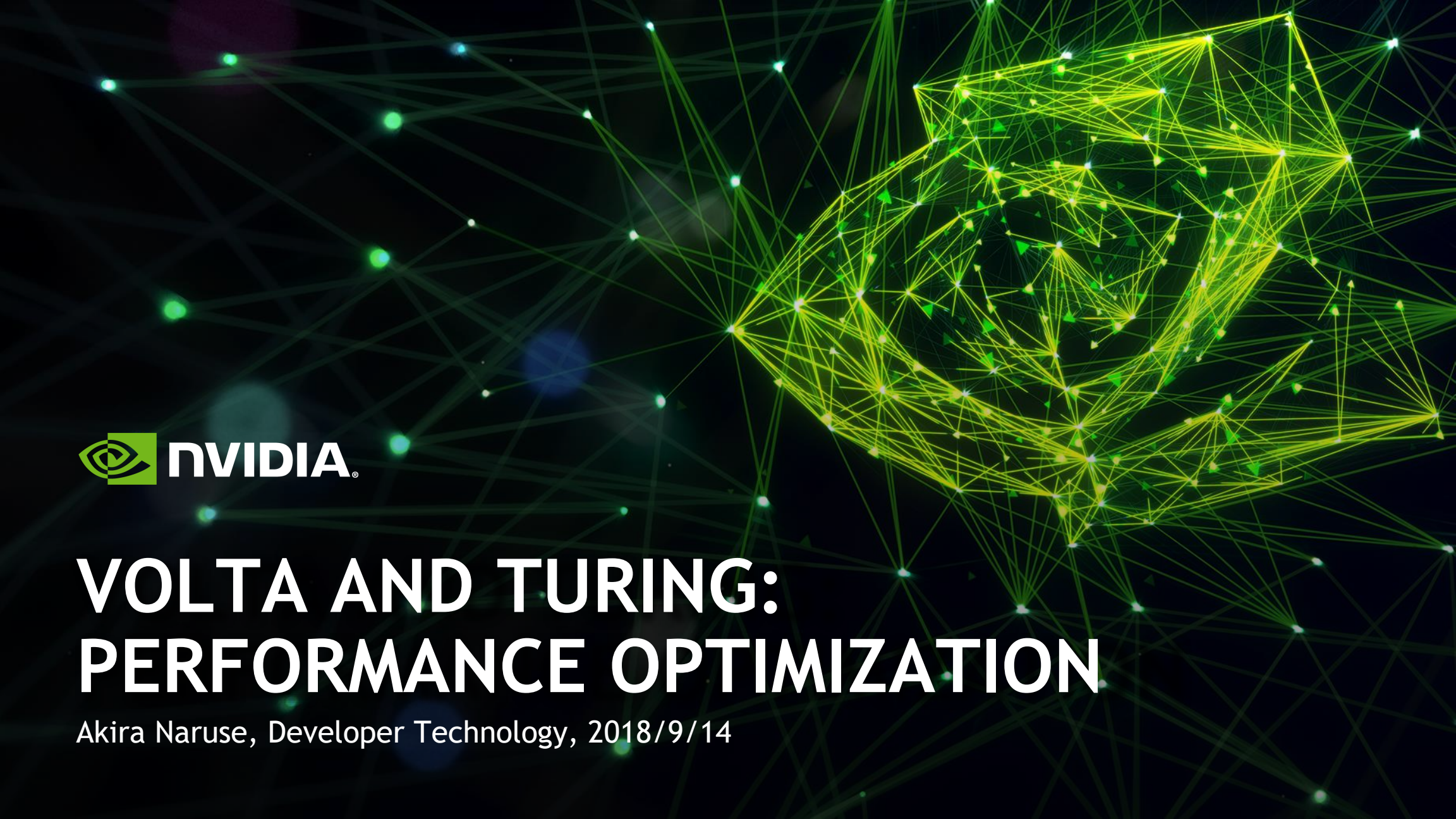
10 Giga Rays /sec

Ray Tracingをハードウェアで加速

- Bounding Volume Hierarchy (BVH) traversal
- Ray/Triangle intersection

SM毎に、RTコアを1つ搭載

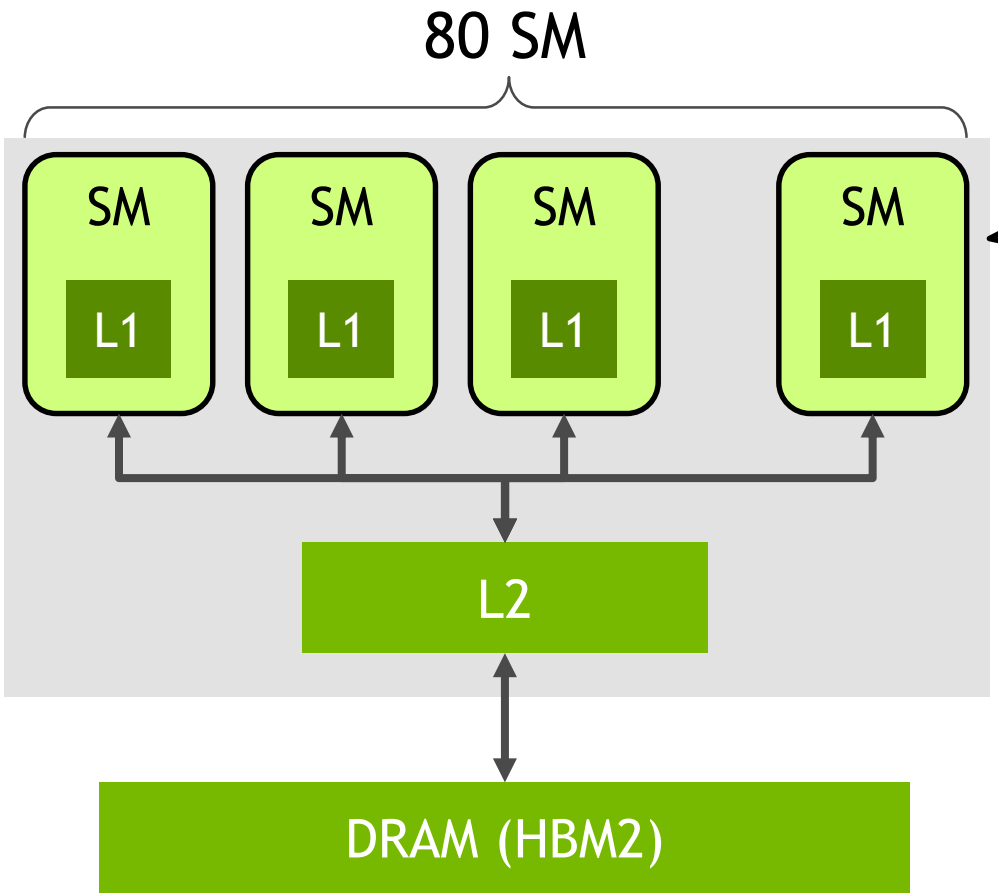
Ray Tracing APIs: NVIDIA OptiX, Microsoft DXR, Vulkan ray tracing



VOLTA AND TURING: PERFORMANCE OPTIMIZATION

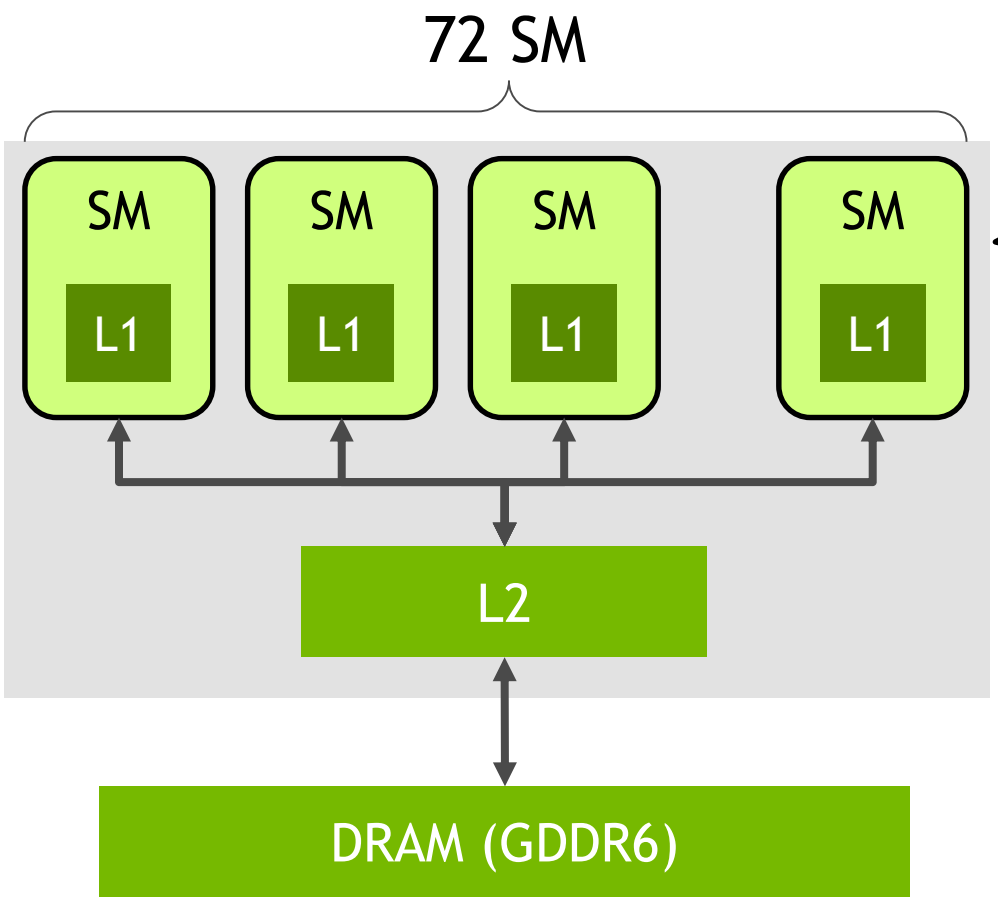
Akira Naruse, Developer Technology, 2018/9/14

VOLTA GV100



- FP32ユニット: 64
- INT32ユニット: 64
- FP64ユニット: 32
- Tensorコア: 8
- SFUユニット: 16

TURING TU102



- FP32ユニット: 64
- INT32ユニット: 64

- Tensorコア: 8
- SFUユニット: 16
- RTコア: 1

GPUを性能を最大限活用するには?

- 演算ユニットの稼働率を上げる
  - INT32, FP32, FP64, Tensorコア, ...
- DRAMからのデータ転送効率を上げる

- 遅延隠蔽 (Latency Hiding)

稼働率を上げるには

エスカレーターの場合

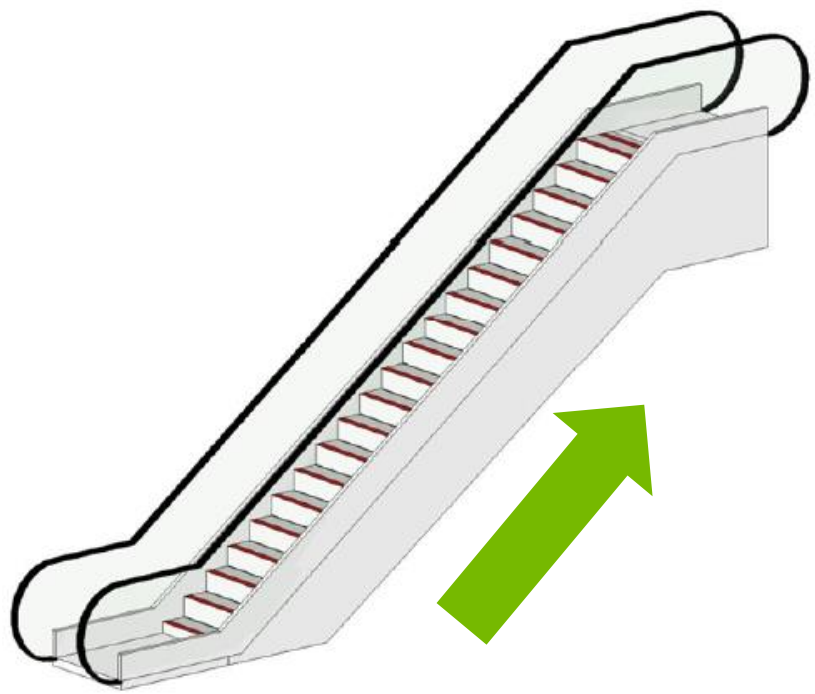
エスカレータの設定:

- 1段あたり、1人乗れる
- 2秒毎に、1段が到着

  最大バンド幅: 0.5人/秒

- 段数は20

  遅延: 40秒



(*) 遅延: 下でエスカレーターに乗ってから、上に到着するまでの時間

稼働率を上げるには

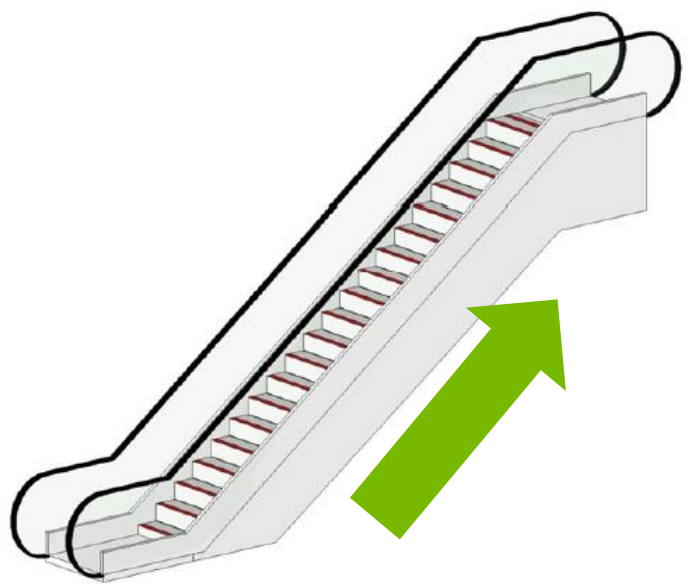
エスカレーターの場合

エスカレータの設定:
- 1段あたり、1人乗れる
- 2秒毎に、1段が到着
  最大バンド幅: 0.5人/秒
- 段数は20
  遅延: 40秒

- もし、同時に1人しか乗らなかったら?
  実効バンド幅: 1人/40秒 = 0.025人/秒

- 稼働率を100%にするには、各段に1人必要、常に20人が乗っている必要がある
  20人 = 最大バンド幅 * 遅延

多くの人を乗せて、遅延を隠蔽する
Latency hiding



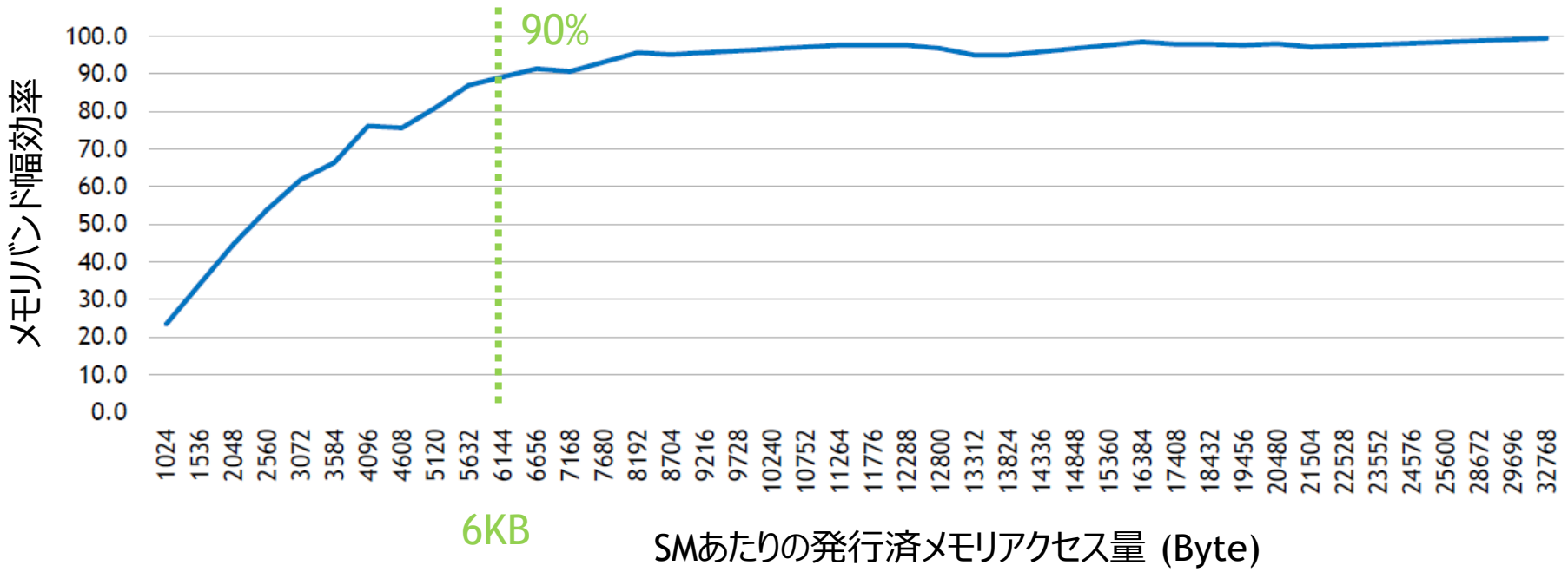
GPUの稼働率を上げるには

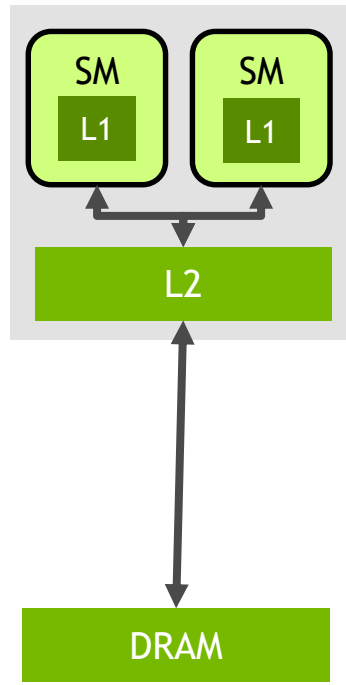
GPUは、CPUと比べて遅延が長い → たくさんの人を乗せる必要がある

- 演算遅延: 演算の実行開始(発行)から終了まで
  - より多くの演算を、発行しておく必要がある
- メモリアクセス遅延: ロード・ストア要求開始(発行)から終了まで
  - より多くのメモリアクセスを、発行しておく必要がある

メモリアクセス発行量と実効バンド幅

Volta GV100





各SMから6KB程度のメモリアクセス要求が発行してあれば、最大実効バンド幅に対して、約90%のバンド幅が得られる (4Bとすると、リクエスト数1.5K)。


CUDA

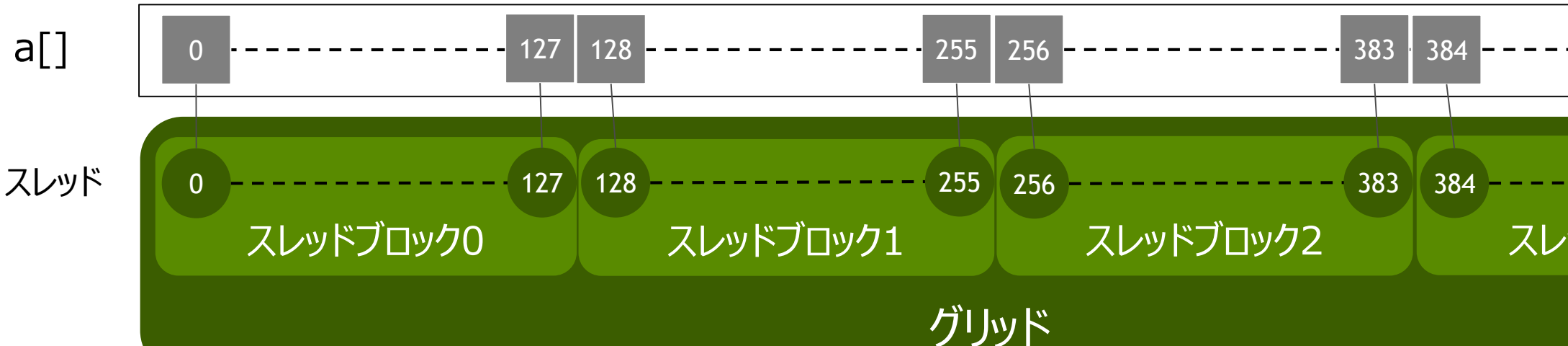
CUDAの基本

グリッド、スレッドブロック、スレッド

```
__global__ void kernel( float *a, float *b, float *c)
{
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    c[i] += a[i] * b[i];
}
```

カーネル:
各スレッドの動作が記述されたプログラム

基本:
1スレッドが、1要素を担当



CUDAの基本

- 命令実行モデル: SIMT
  - Single Instruction Multiple Threads
  - 一つの同じ命令を、複数のスレッドが、同時に実行する
- 複数のスレッドって、何スレッド?
  - **32スレッド = 1ワープ**
- ワープ単位で命令が実行される、これがCUDAのプログラム実行の基本

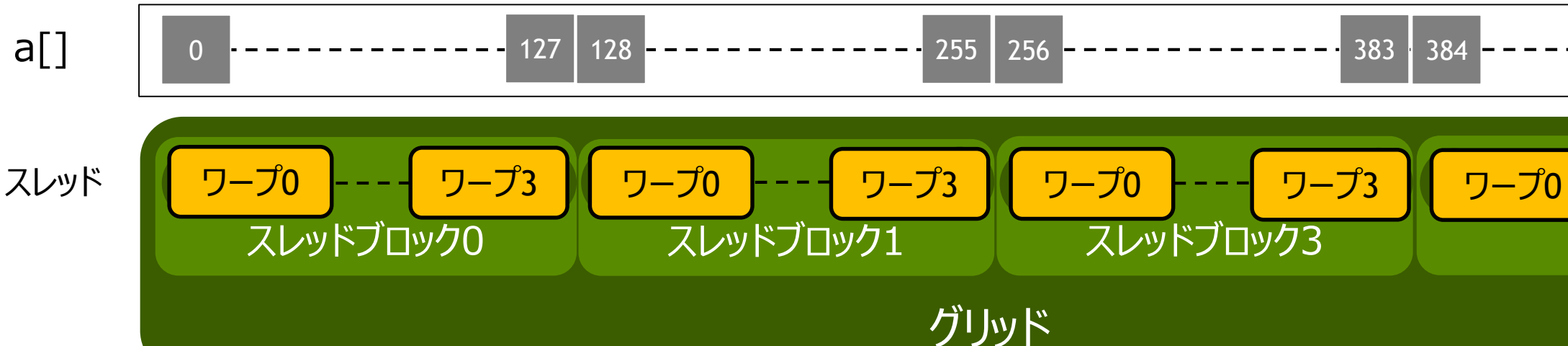
CUDAの基本

グリッド、スレッドブロック、(ワープ、) スレッド

```
__global__ void kernel( float *a, float *b, float *c)
{
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    c[i] += a[i] * b[i];
}
```

カーネル:
各スレッドの動作を記述したコード

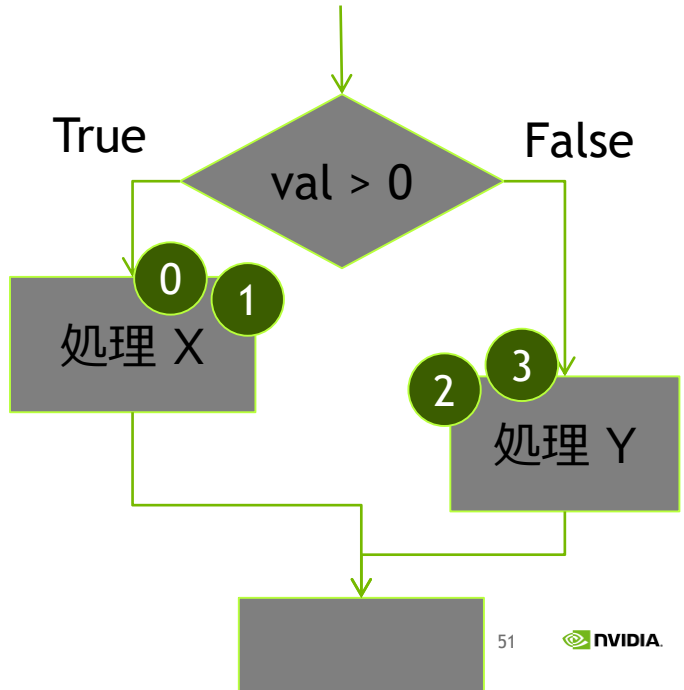
基本:
1スレッドが、1要素を担当



CUDAの基本

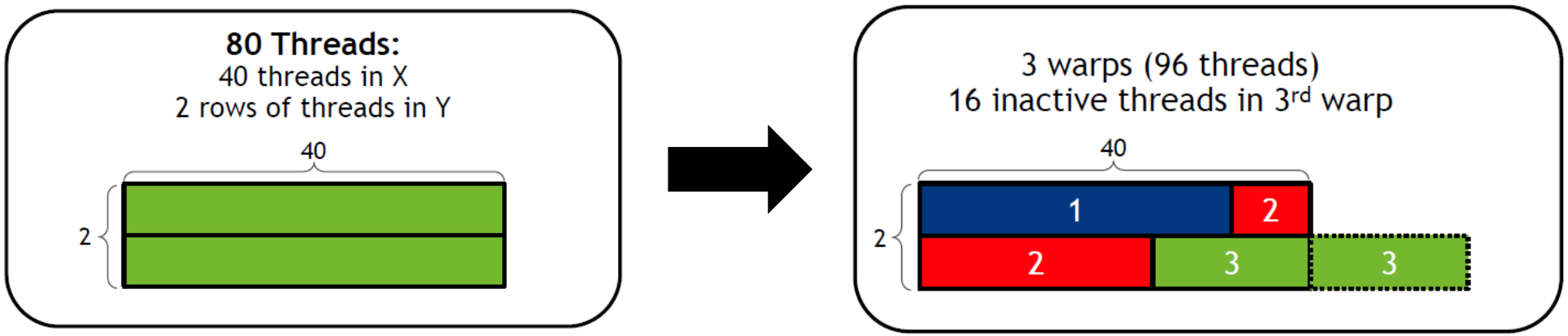
命令実行フロー

- プログラムカウンタ(PC)は、ワープ毎に存在
  - 異なるワープのスレッドは、異なる命令を実行できる、性能低下無しで
- 同じワープの別スレッドが条件分岐で、別パスに行くと、どうなるの?
  - そのワープは、両方のパスを実行する
  - ただし、実行中のパスにいないスレッドは、無効化される

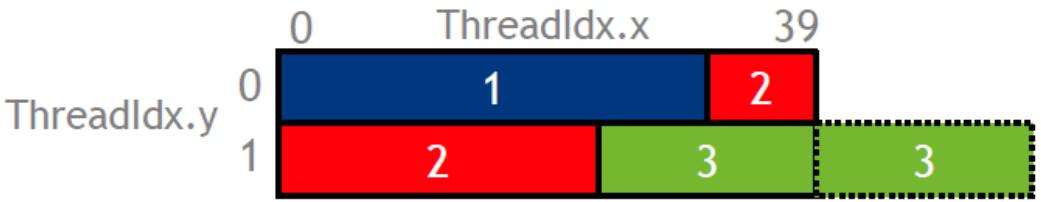


CUDAの基本
スレッド・マッピング

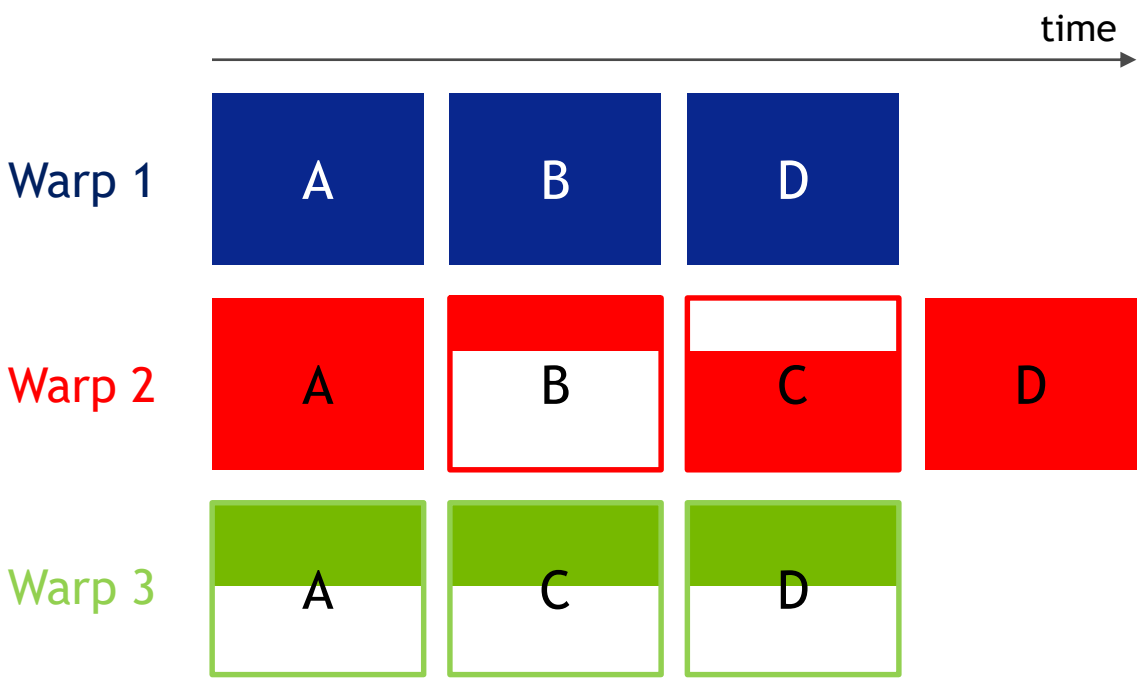
- スレッドブロックの形状は、プログラムからは、1D、2D or 3Dで指定可能
  - ハードウェアは、あくまで1Dとして認識
  - スレッドブロックの形状がどうであれ、連続32スレッドが、1ワープにマップされる



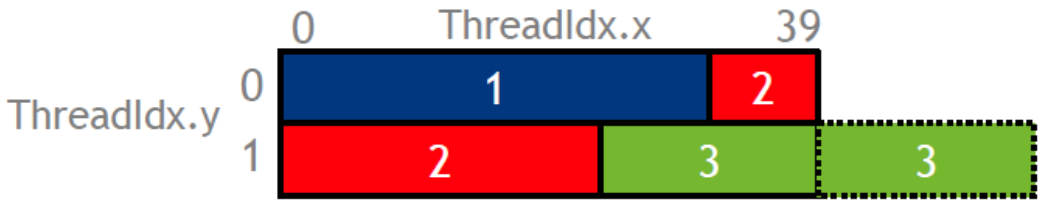
命令実行フロー




```
A;
if (threadIdx.y == 0)
        B;
else
        C;
D;
```



命令実行フロー

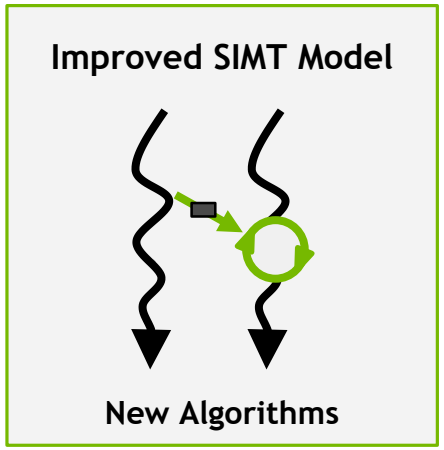


スレッドが無意味なサイクルを消費するのを避ける

- スレッドブロック内のスレッド数を、32の倍数にする
- 同じワープ内のスレッドが、別のコードパスに分岐する機会を減らす

INDEPENDENT THREAD SCHEDULING

Volta/Turingから



- プログラムカウンター(PC):
  - Pascalまで、ワープ単位で管理
  - Volta/Turingから、スレッド単位で管理
- 何ができるようになる?
  - 同じワープ内のスレッド間で、非対称なデータの受け渡しが可能になる
  - producer consumer型のコードを、より自然な形で記述できる

INDEPENDENT THREAD SCHEDULING

例: 同じワープ内のスレッド間でのロックの受け渡し

```
lock = 0;
do {
        lock = tryLock();
} while (lock == 0);
...;
releaseLock();
```

(*) atomic命令やvolatileポインターを使用する必要がある

PascalまでのGPUでは、deadlock発生

- lockを取得できなかったスレッドが、lock取得を繰り返す → lockを取得したスレッドがlock解放まで進めない → deadlock

Volta/Turingから、正常に実行できる

INDEPENDENT THREAD SCHEDULING

注意

同じワープ内のスレッドが、lock-step実行される保証はない

- 一旦、実行パスが分離すると、スレッドは自然には合流しない
- 同じワープ内のスレッドでも、明示的な同期が必要なケース、__syncwarp()

暗黙的にスレッド間同期を想定したコードは危険 (warp-synchronous)

- shuffle命令やvote関数のAPIを見直し
- 例: __shfl_xor(val, 0x1) → __shfl_xor_sync(0xffffffff, val, 0x1)
  - 第一引数で、ワープ内のどのスレッドが、この命令に到達するかを指定
- 従来APIのshuffle命令, vote関数の使用は非推奨

INDEPENDENT THREAD SCHEDULING

Warp-synchronousコードがあるけど、どうすれば良い?

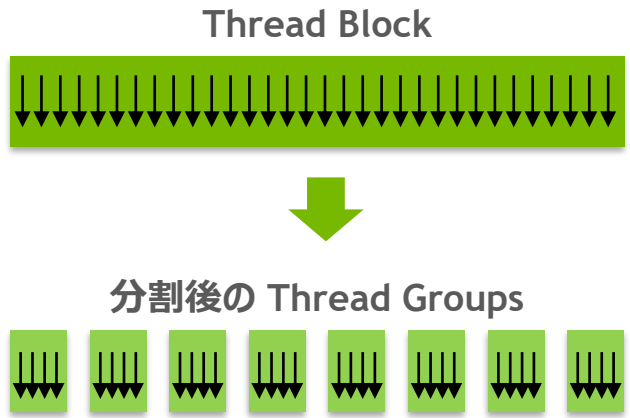
- CUDA 9で導入したAPI を使用する: *_sync(mask, …)
- コンパイラで古いアーキテクチャを指定する
  - -arch=compute_60,sm_70 (Volta binary)
  - -arch=compute_60,sm_75 (Turing binary)
  - -arch=compute_60 (PTX JIT)
- Cooperative Groupsを使用する
  - Cooperative Groupsって何?

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 (CUDA 9.0から)

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



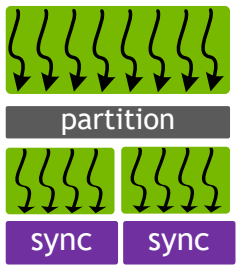
* Note: Multi-Block and Mult-Device Cooperative Groups are only supported on Pascal and above GPUs

多様なスレッド間同期を簡単に

3つのスケール

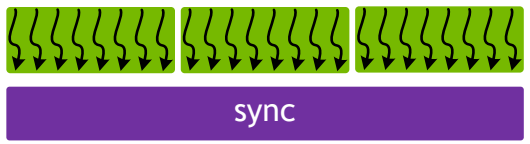
スレッドブロック内

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

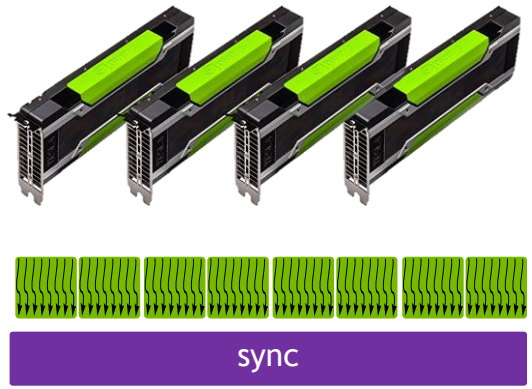


シングルGPU内 (SM間の同期)

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

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)
```

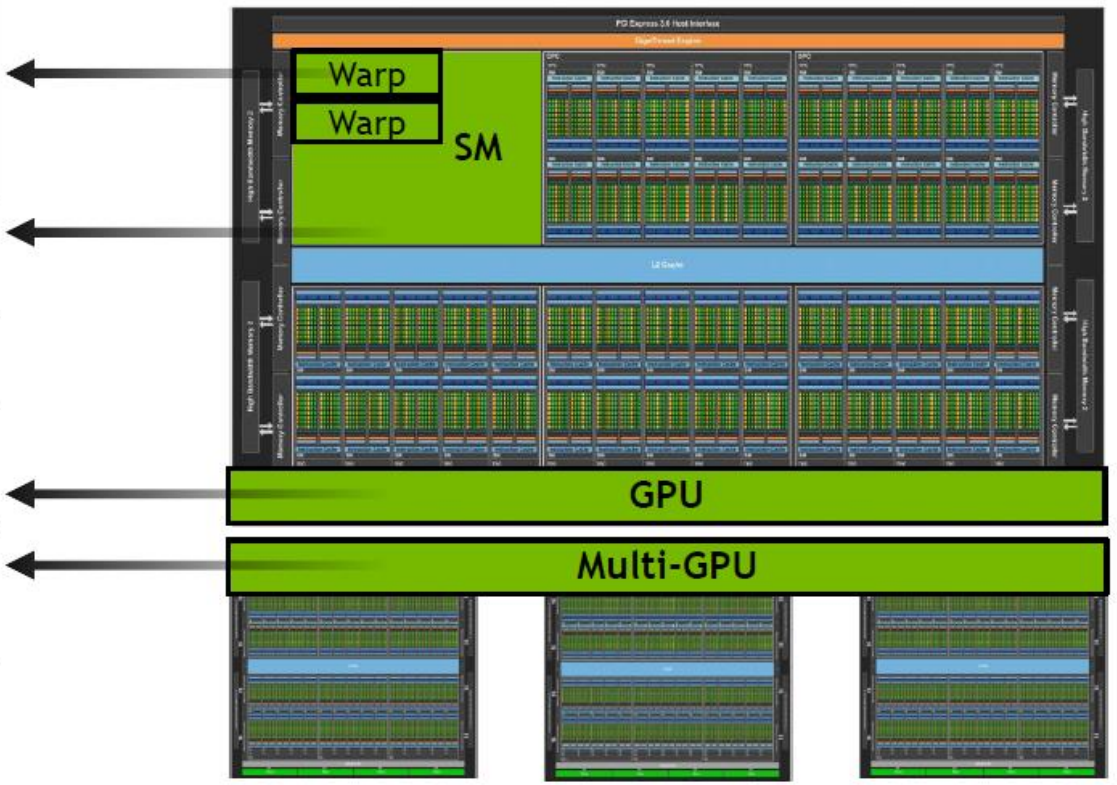
For CUDA thread blocks:
```
auto g = this_thread_block();
```

スレッドブロック

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

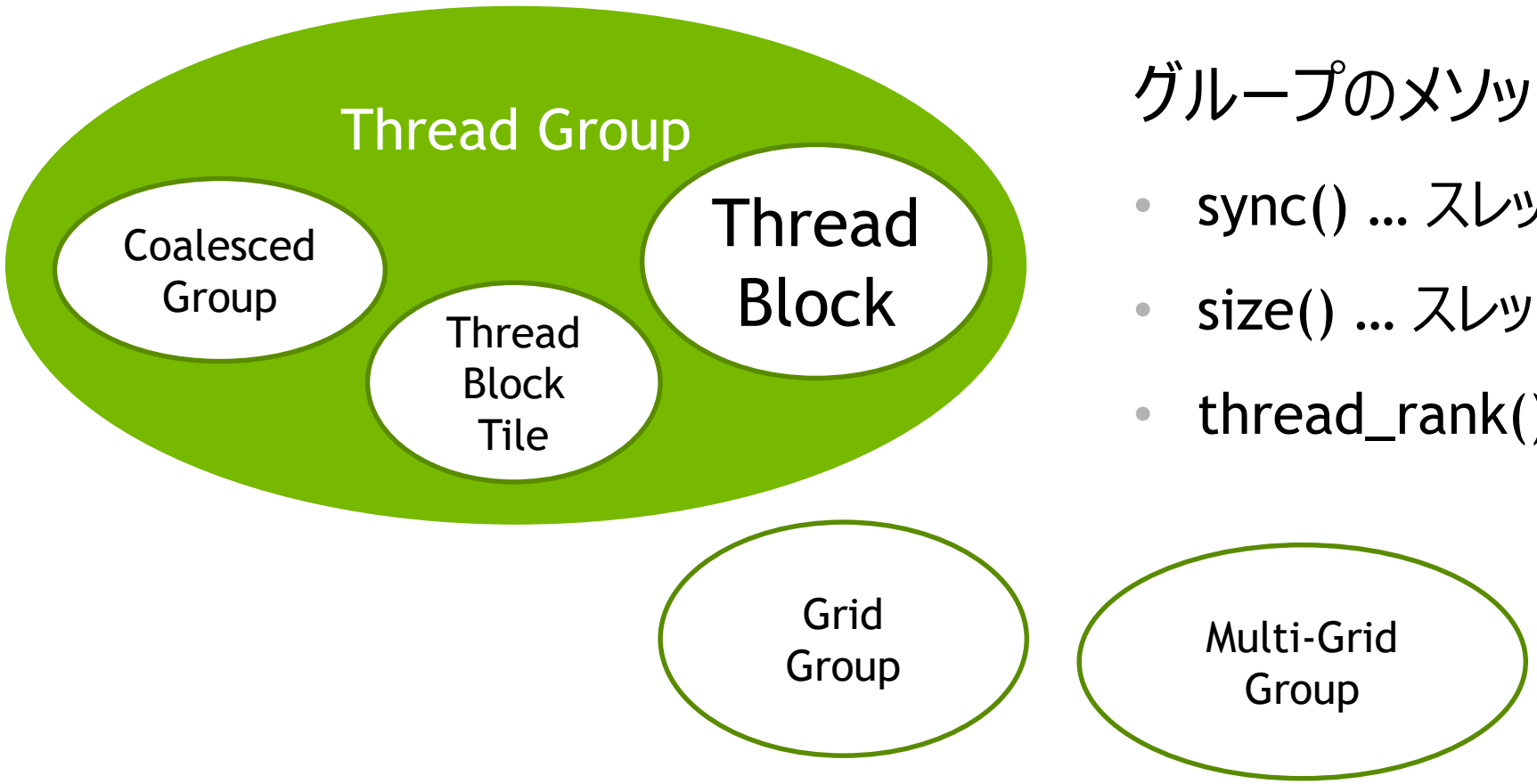
For multiple grids spanning GPUs:
```
auto g = this_multi_grid();
```

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

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

SM占有率 (OCCUPANCY)

$$\text{SM占有率} = \frac{\text{実効スレッド数}}{\text{最大スレッド数}}$$

SM占有率

SM占有率は、高いほど良い

- 何故？ → 占有率が高い → スレッド数が多い → 命令を発行できるスレッドが増える → 発行済み命令数・メモリアクセス数を増やせる
  - 遅延を隠蔽できる
- 実効スレッド数をどうやって増やす？　レジスタ使用量を減らす、共有メモリ使用量を減らす、適切なスレッドブロックサイズを選択する...

実効スレッド数

各スレッドブロックが使用するリソース量で決まる

	Volta GV100	Turing TU102
最大スレッド数 (ワープ数)	2048 (64ワープ)	1024 (32ワープ)
レジスターサイズ	256KB	256KB
共有メモリ量	最大96KB	最大64KB

使用リソース量 / スレッドブロック:

- レジスター = スレッドあたり使用レジスター数 * スレッド数
- 共有メモリ使用量

CUDAツールキット内の Occupancy Calculatorで確認可能

CUDA OCCUPANCY CALCULATOR

/usr/local/cuda/tools/CUDA_Occupancy_Calculator.xls

CUDA Occupancy Calculator

Click Here for detailed instructions on how to use this occupancy calculator.

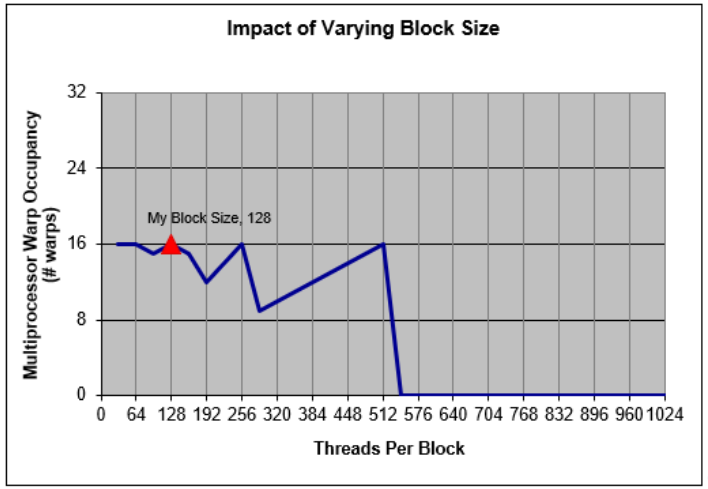
For more information on NVIDIA CUDA, visit http://developer.nvidia.com/cuda

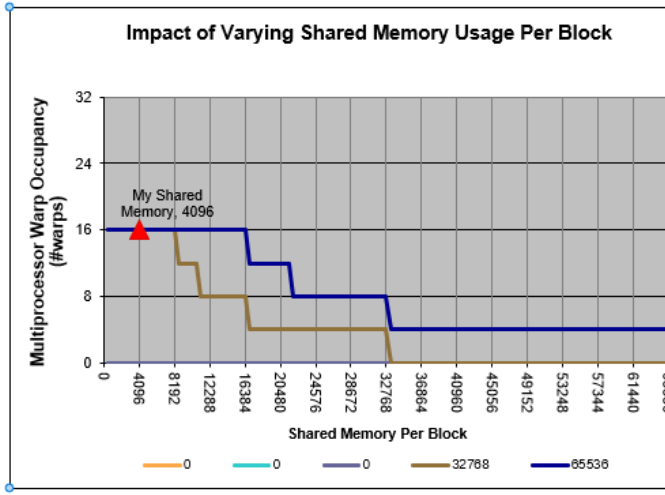
Just follow steps 1, 2, and 3 below! (or click here for help)

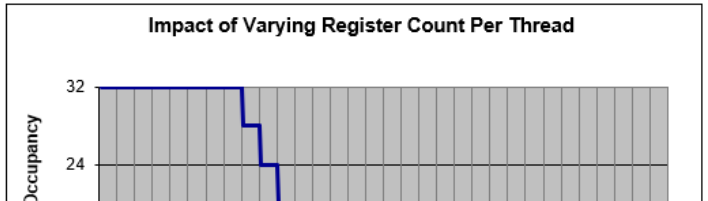
1.) Select Compute Capability (click):	7.5	(Help)
1.b) Select Shared Memory Size Config (bytes)	65536	
2.) Enter your resource usage:		
Threads Per Block	128	(Help)
Registers Per Thread	128	
Shared Memory Per Block (bytes)	4096	
(Don't edit anything below this line)		
3.) GPU Occupancy Data is displayed here and in the graphs:		
Active Threads per Multiprocessor	512	(Help)
Active Warps per Multiprocessor	16	
Active Thread Blocks per Multiprocessor	4	
Occupancy of each Multiprocessor	50%	

Physical Limits for GPU Compute Capability:	7.5
Threads per Warp	32
Max Warps per Multiprocessor	32
Max Thread Blocks per Multiprocessor	32
Max Threads per Multiprocessor	1024
Maximum Thread Block Size	1024
Registers per Multiprocessor	65536
Max Registers per Thread Block	65536
Max Registers per Thread	255
Shared Memory per Multiprocessor (bytes)	65536
Max Shared Memory per Block	65536
Register allocation unit size	256
Register allocation granularity	warp

Your chosen resource usage is indicated by the red triangle on the graphs. The other data points represent the range of possible block sizes, register counts, and shared memory allocation.







同時発行できる命令数

並列度をどうやって増やすか

SM占有率の改善

- スレッドが増えれば、発行可能命令数も増える

Instruction Level Parallelism (ILP) の改善

- 各スレッドが実行する命令間の依存性が減れば、発行可能命令数は増える

GPUの命令発行

命令は順番に発行

- GPUの命令発行は in-order (out-of-orderではない)

次命令の発行条件が満たされないと、そのワープはストール (後続命令も発行されない)

- 条件1: その命令が使用するデータが準備できている
  - 先行命令によるデータ読み込み・生成が完了している
- 条件2: その命令が使用する演算ユニットが使える
  - 先行命令による演算ユニットの使用が終わっている

命令発行の例

```
__global__ void kernel( float *a, float *b, float *c)
{
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    c[i] += a[i] * b[i];
}
```

```
LDG.E.32 R2, [R2];
LDG.E.32 R4, [R4];
LDG.E.32 R8, [R6];

FFMA R8, R2, R4, R8;

STG.E.32 [R6], R8
```

Load (12B)

Multiply-Add

Store (4B)

命令発行の例

```
__global__ void kernel( float *a, float *b, float *c)
{
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    c[i] += a[i] * b[i];
}
```

```
LDG.E.32 R2, [R2];
LDG.E.32 R4, [R4];
LDG.E.32 R8, [R6];
stall
FFMA R8, R2, R4, R8;

STG.E.32 [R6], R8
```

Load (12B)

Multiply-Add

Store (4B)

命令発行の例

```
__global__ void kernel( float *a, float *b, float *c)
{
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    c[i] += a[i] * b[i];
}
```

```
LDG.E.32 R2, [R2];
LDG.E.32 R4, [R4];
LDG.E.32 R8, [R6];
stall
FFMA R8, R2, R4, R8;
stall
STG.E.32 [R6], R8
```

Load (12B)

Multiply-Add

Store (4B)

2つの要素を同時に計算

```
__global__ void kernel( float2 *a, float2 *b, float2 *c)
{
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    c[i].x += a[i].x * b[i].x;
    c[i].y += a[i].y * b[i].y;
}
```

2倍量の
メモリ要求を
発行

```
LDG.E.64 R2, [R2];
LDG.E.64 R4, [R4];
LDG.E.64 R8, [R6];
stall
FFMA R8, R2, R4, R8;
FFMA R9, R3, R5, R9;
stall
STG.E.64 [R6], R8
```

Load (24B)

独立に発行可能

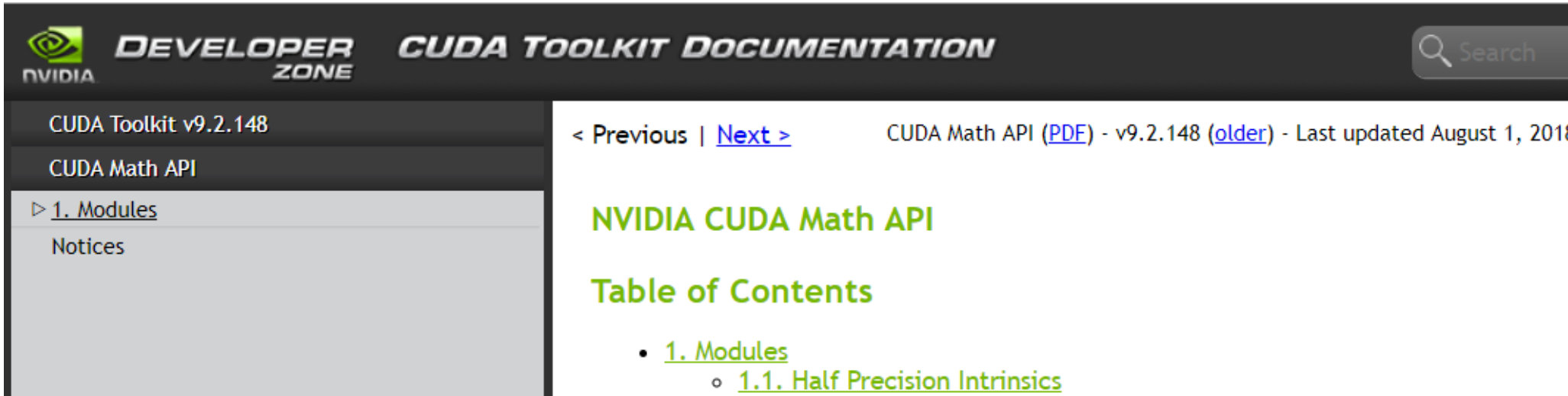
2 Multiply-Add

Store (8B)

FAST MATH

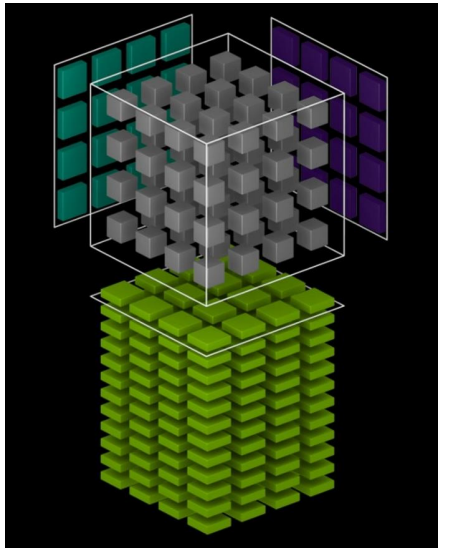
より高速に実行できるintrinsicが使用可能（精度は低下）

- 全体に適用: コンパイルオプション: --fast-math
- 個別に適用: __cosf(x), __logf(x), __expf(x)



TENSORコア

Volta/Turingで使用可能

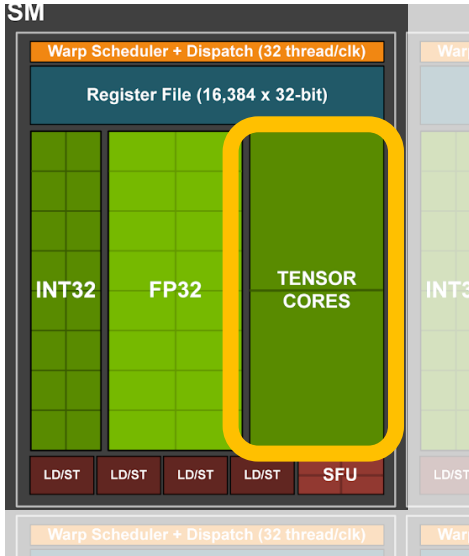


行列積専用の演算ユニット

- Volta:
  - 125 TFLOPS (fp16)
- Turing:
  - 125 TFLOPS (fp16)
  - 250 Tops (int8)
  - 500 Tops (int4)

Volta SM

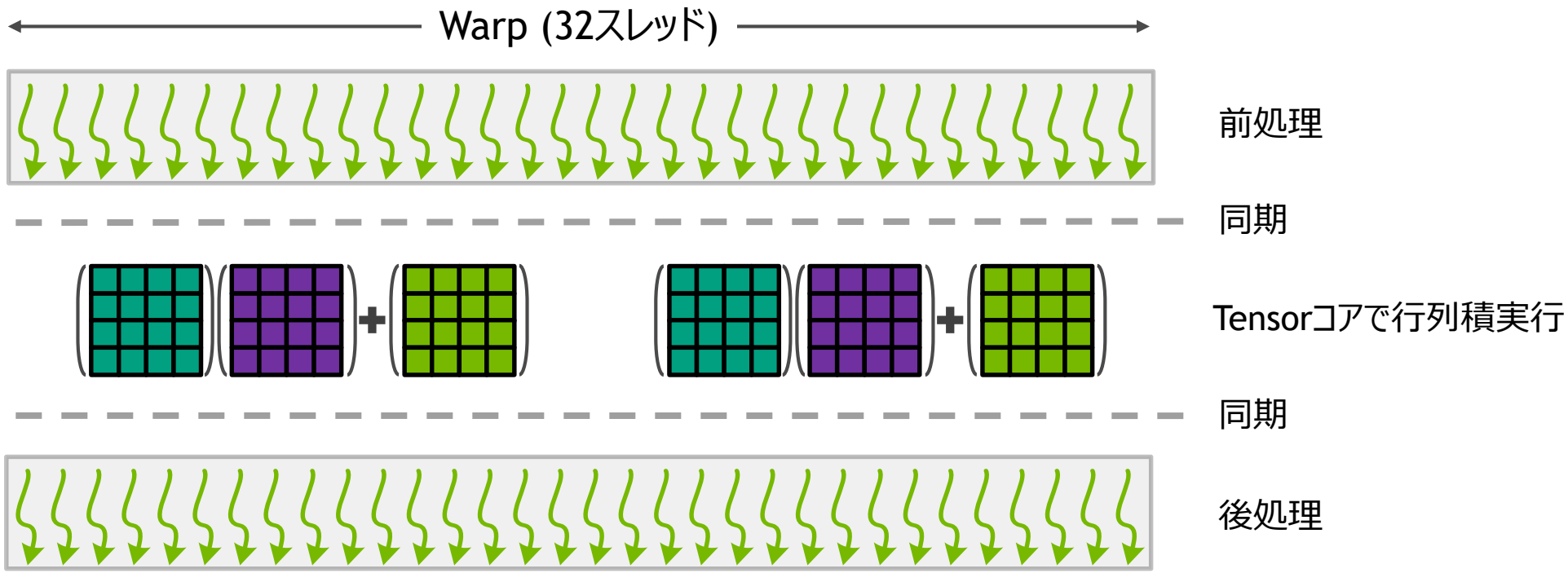
Turing SM



TENSORコアの使われ方

Warp単位で実行

16x16の行列の積和演算を、Warp単位(32スレッド)で協調実行



TENSORコアの使い方 (FP16)

CUDA WMMA API

```
__device__ void tensor_op_16_16_16(half *a, half *b, float *c)
{
   wmma::fragment<wmma::matrix_a, 16, 16, 16, half, …> a_frag;
   wmma::fragment<wmma::matrix_b, 16, 16, 16, half, …> b_frag;
   wmma::fragment<wmma::accumulator, 16, 16, 16, float, …> c_frag;

   wmma::load_matrix_sync(a_frag, a, …);
   wmma::load_matrix_sync(b_frag, b, …);

   wmma::fill_fragment(c_frag, 0.0f);

   wmma::mma_sync(c_frag, a_frag, b_frag, c_frag);

   wmma::store_matrix_sync(c, c_frag, …);
}
```

Tensorコアへの入出力データ型(fragment)の宣言

入力行列の一部を入力fragmentに読み込み

出力fragmentを初期化

Tensorコア演算

出力fragmentを出力行列に書き込み

TENSORコアの使い方 (INT8)

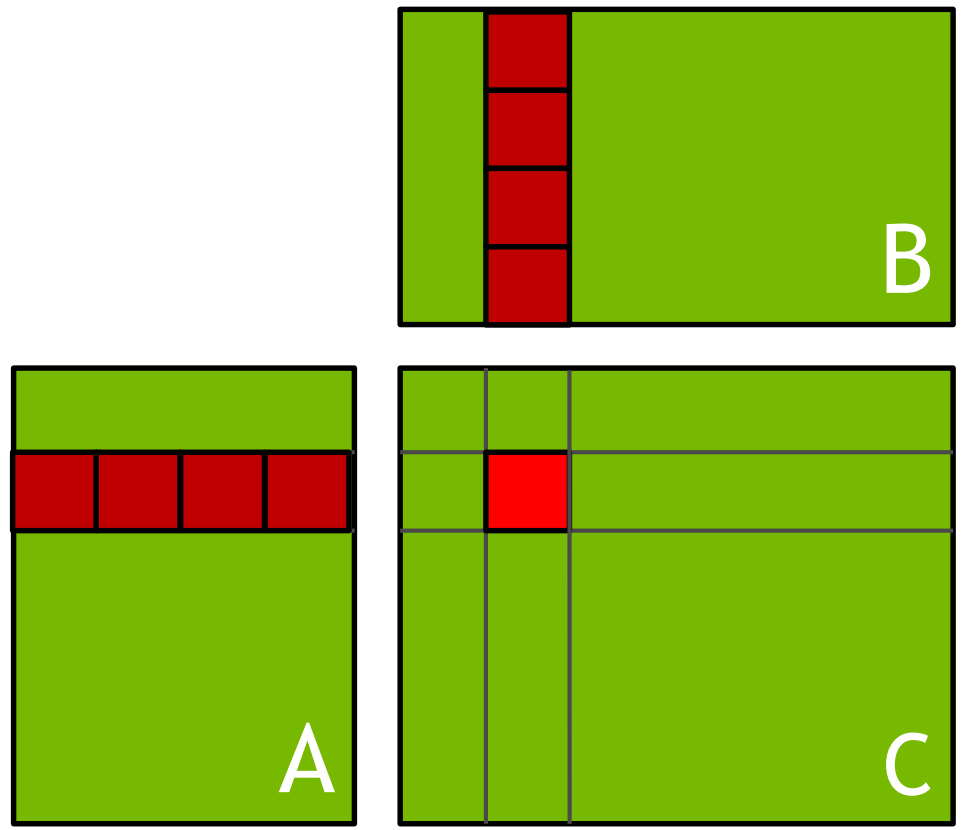
CUDA WMMA API

```
__device__ void tensor_op_16_16_16(char *a, char *b, int *c)
{
    wmma::fragment<wmma::matrix_a, 16, 16, 16, char, …> a_frag;
    wmma::fragment<wmma::matrix_b, 16, 16, 16, char, …> b_frag;
    wmma::fragment<wmma::accumulator, 16, 16, 16, int, …> c_frag;

    wmma::load_matrix_sync(a_frag, a, …);
    wmma::load_matrix_sync(b_frag, b, …);

    wmma::fill_fragment(c_frag, 0);

    wmma::mma_sync(c_frag, a_frag, b_frag, c_frag);

    wmma::store_matrix_sync(c, c_frag, …);
}
```

TENSORコアの使用例

C += A * B

- 大きな行列乗算は、小さな行列乗算の集合とみなせる
- 行列Cを、16x16の小行列に分ける
- 各小行列にワープを割り当てる
- 各ワープは、入力行列A,Bの関連部分を16x16のサイズで読み込み、Tensorコアを使って16x16で行列積を実行、計算が完了するまでこれを繰り返す
- 16x16でないと駄目なのか?



TENSORコアの使い方

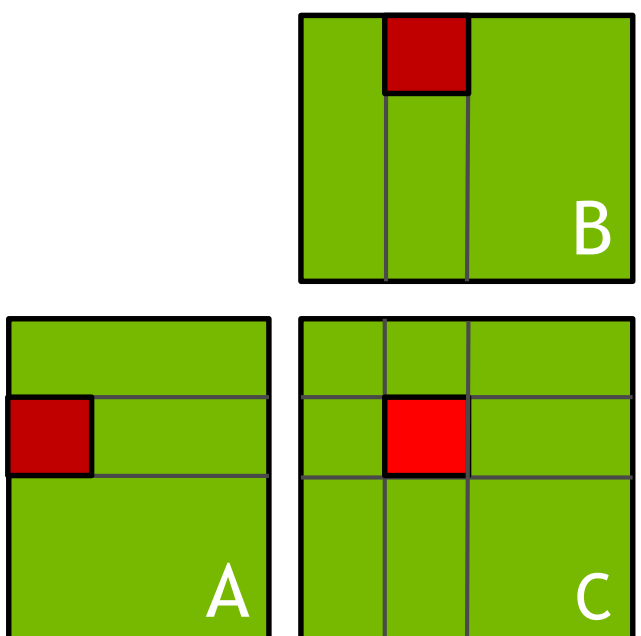
CUDA WMMA API

```
__device__ void tensor_op_16_16_16(half *a, half *b, float *c)
{
    wmma::fragment<wmma::matrix_a, 16, 16, 16, half, …> a_frag;
    wmma::fragment<wmma::matrix_b, 16, 16, 16, half, …> b_frag;
    wmma::fragment<wmma::accumulator, 16, 16, 16, float, …> c_frag;

    wmma::load_matrix_sync(a_frag, a, …);
    wmma::load_matrix_sync(b_frag, b, …);

    wmma::fill_fragment(c_frag, 0.0f);

    wmma::mma_sync(c_frag, a_frag, b_frag, c_frag);

    wmma::store_matrix_sync(c, c_frag, …);
}
```

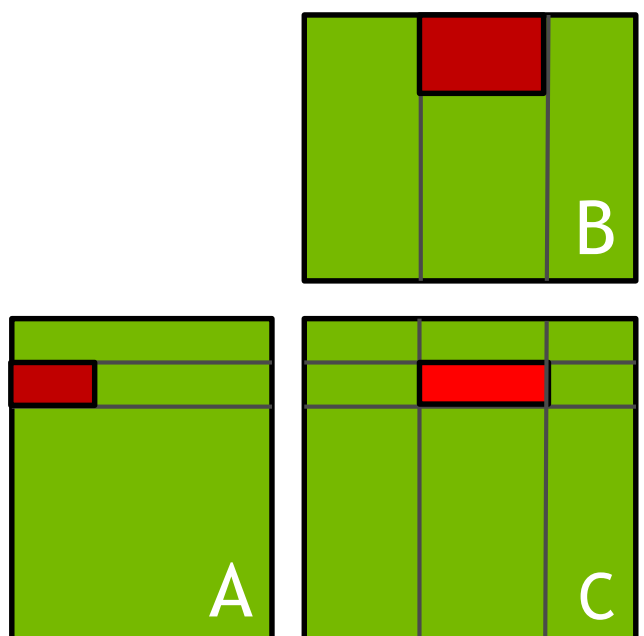
CUDA WMMA API

3種類のタイルパターン

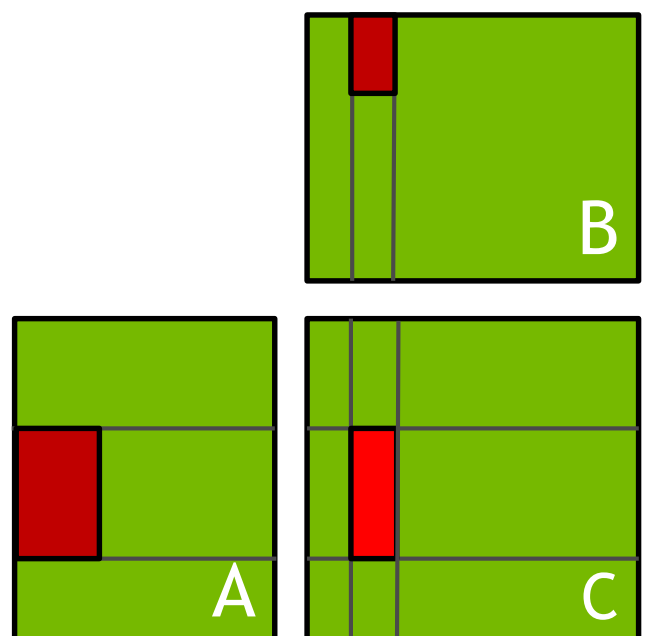
16x16x16



8x32x16



32x8x16



TENSORコア

DLフレームワークで使えるの？

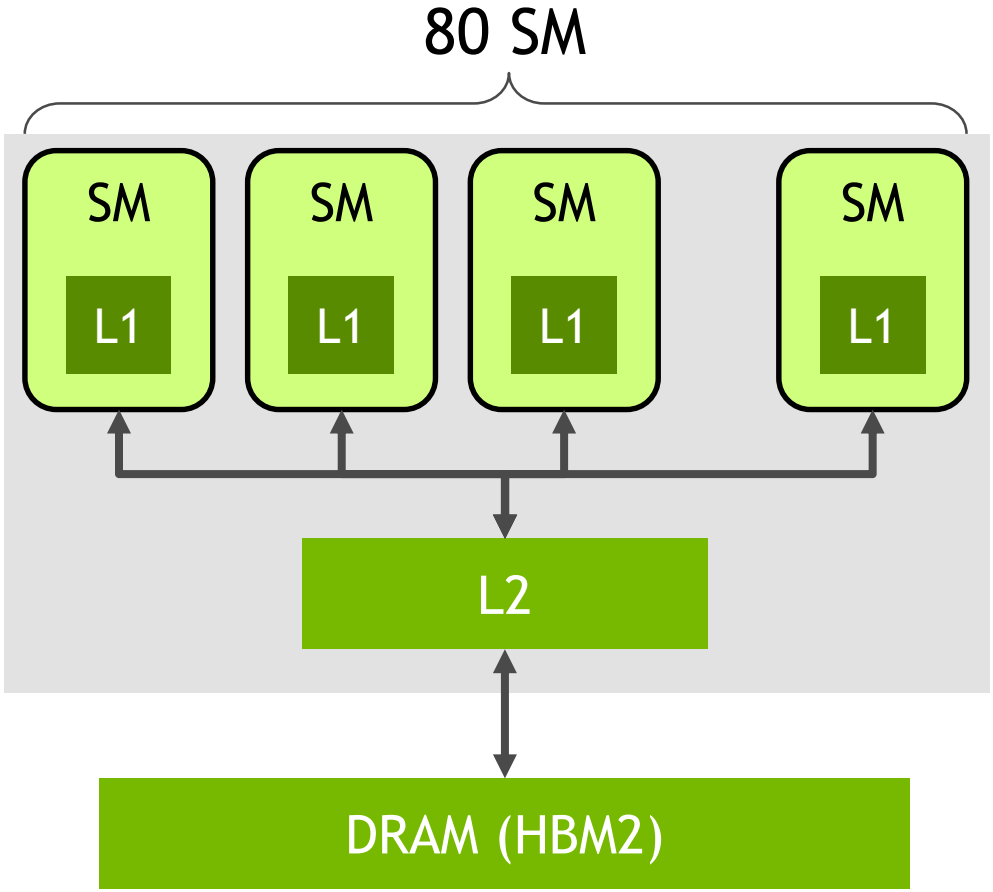
NVIDIAのライブラリーでサポート

- cuBLAS
- cuDNN
- TensorRT
- CUTLASS

DL frameworks

メモリシステム

VOLTA GV100



レジスタファイル

- 256KB/SM (トータル20MB)

統合共有メモリ/L1キャッシュ

- 128KB/SM (トータル10MB, 14TB/s)

L2キャッシュ

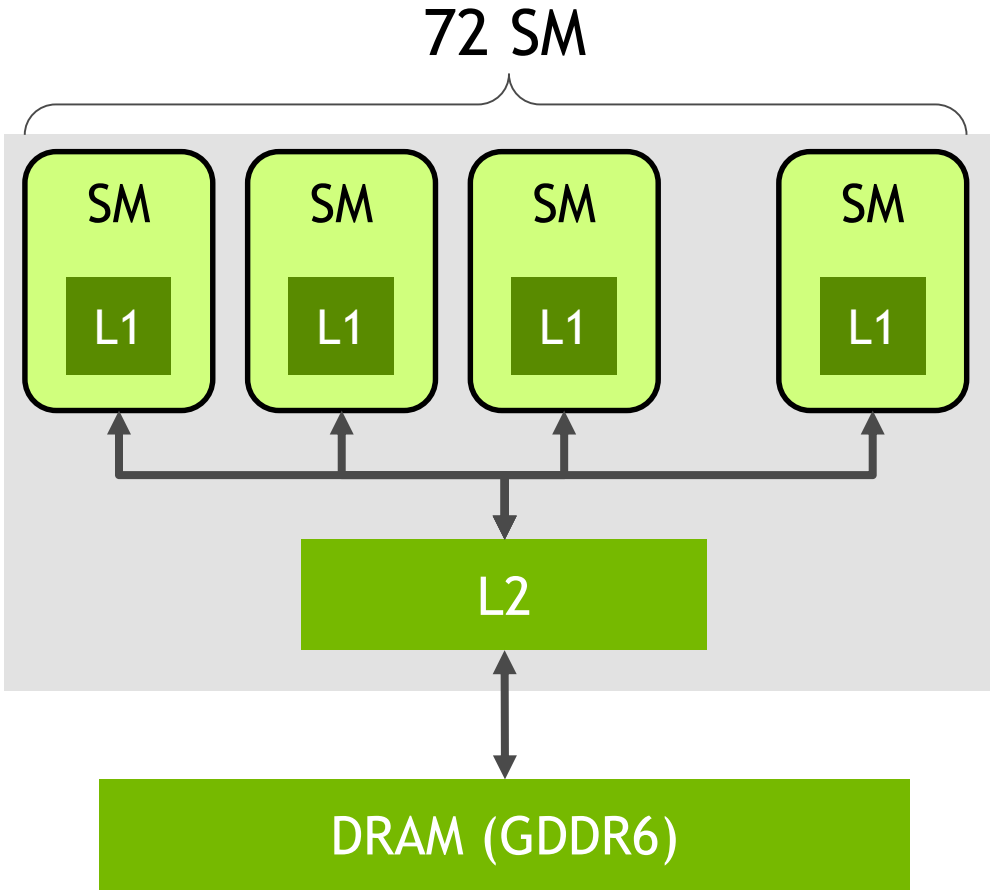
- 6MB、Read: 2.5TB/s、Write: 1.6TB/s

DRAM (HBM2):

- 16/32GB、900GB/s

メモリシステム

Turing TU102



レジスタファイル

- 256KB /SM (トータル 18.5MB)

統合共有メモリ/L1キャッシュ

- 96KB /SM (トータル 6.75MB, 8TB/s)

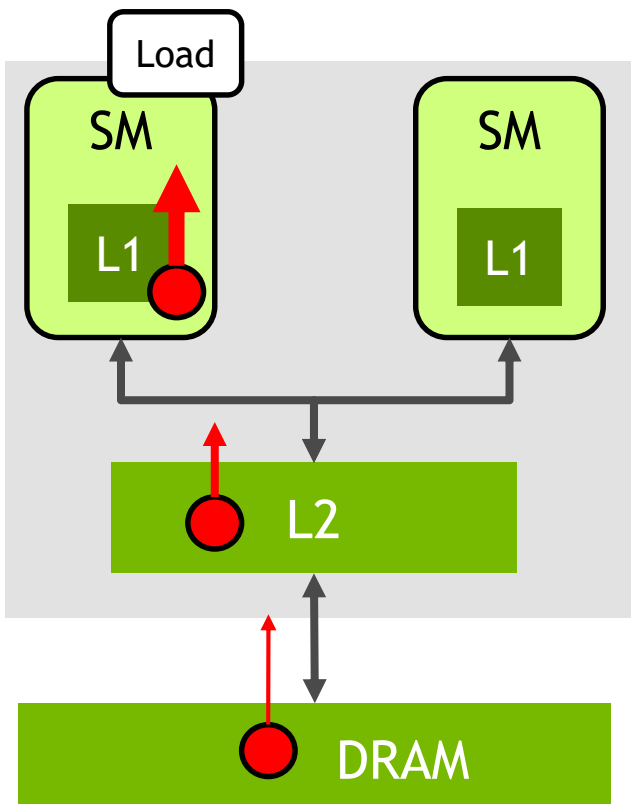
L2キャッシュ

- 6MB

DRAM (GDDR6):

- 24GB、672GB/s

メモリ読み出し



L1キャッシュにデータがあるかをチェック

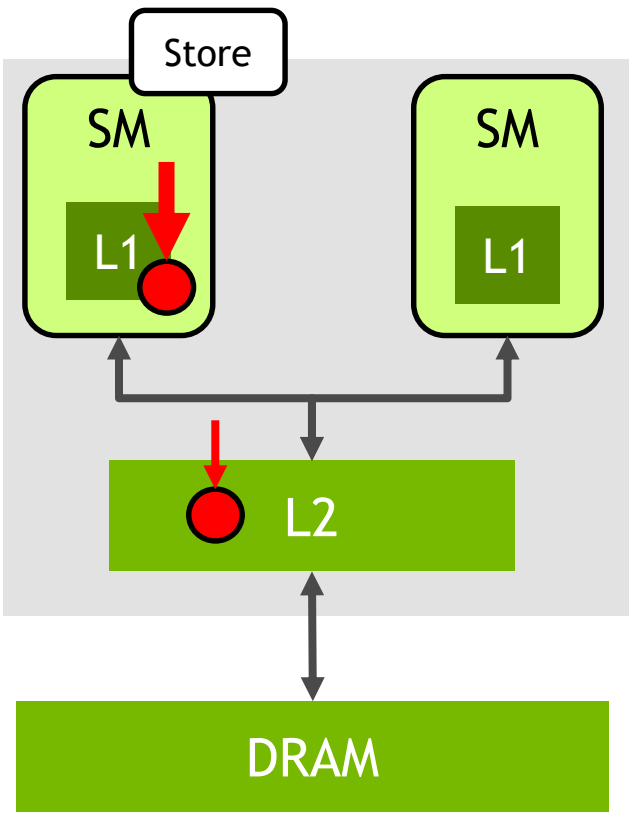
- もしあれば、L1からデータを供給（終了）

L2キャッシュにデータあるかをチェック

- もしあれば、L2からデータを供給（終了）

DRAMからデータを読み出し

メモリ書き込み



L1キャッシュ

- Pascal: L1には書き込まれず、L2に書き込まれる
- Volta/Turing: L1に書き込まれるが、L2にも書き込まれる (write-through)

L2キャッシュ

- DRAMへの書き込みは、必要なときに行われる

(*) キャッシュの挙動は、各LD/ST命令にOperatorを付けることで変えられる (要inline PTX)

GPUのL1/L2キャッシュは何のため?

参照の局所性

- 空間的局所性（ページサイズ、キャッシュラインサイズ）
- 時間的局所性（キャッシュが効く主要因はこちら）

各スレッドの時間的局所性だけでは、キャッシュを活用できない

- 各スレッドが使えるキャッシュ容量:　L1:64B、L2:76B
- (条件) 1024スレッド/SM、 80 SM、L1/SM:64KB、L2:6MB

GPUのL1/L2キャッシュは何のため?

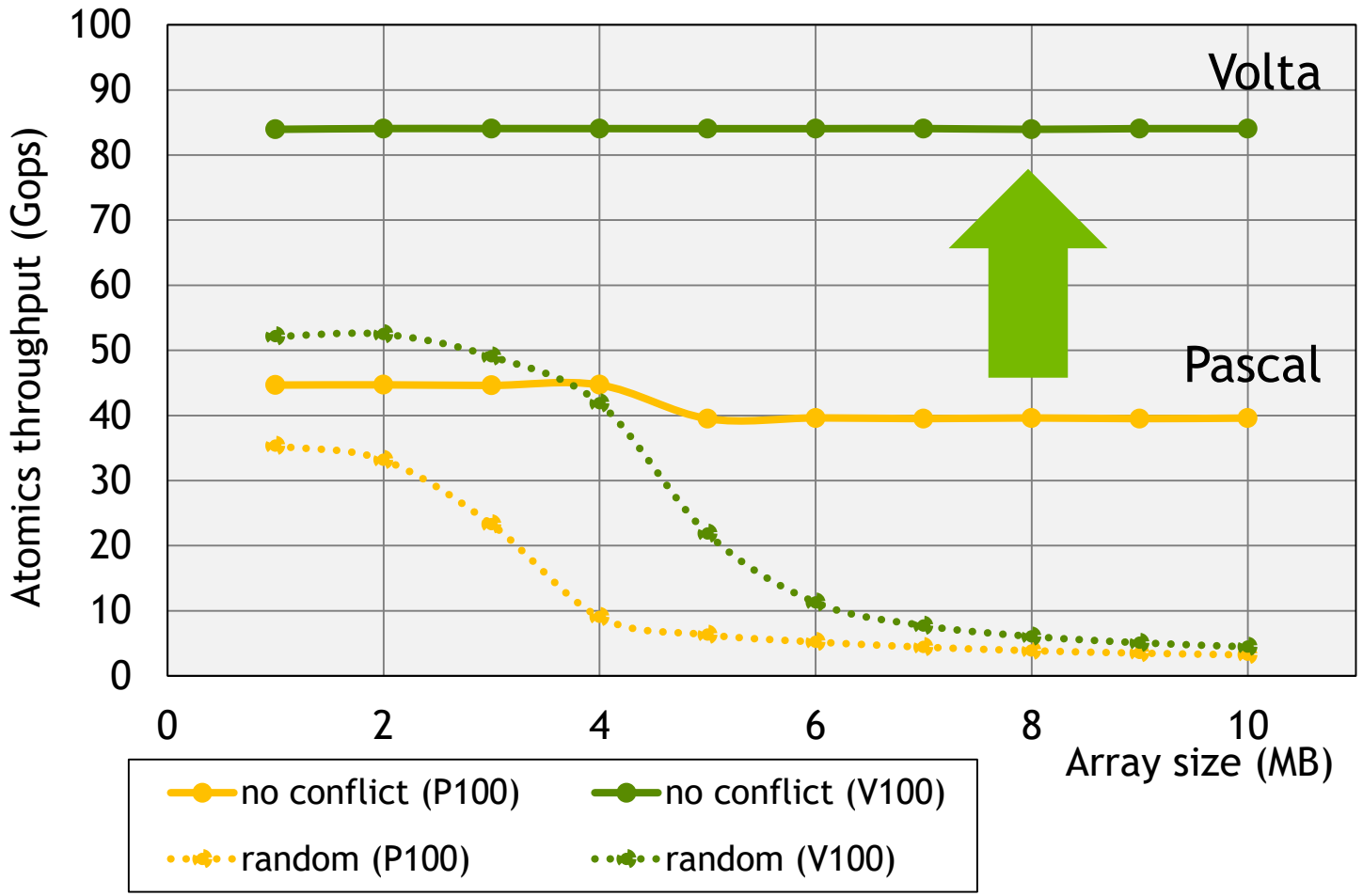
GPUのキャッシュは「スレッド間の局所性」を利用している

- スレッド間の局所性(スレッドブロック内)を明示的に制御するなら、共有メモリ

その他

- 共有メモリを使わないコード (naïveコード, OpenACC等)
- Atomicオペレーションの高速化
- アクセスパターンが不規則で実行前に分からないコード
- レジスタースピルによる速度低下の緩和

L2 ATOMICS性能比較



最大2倍のスループット向上

- AtomicAdd (FP32)
- 256M threads
- Access pattern:
  regular, random

メモリアクセスパターン

Coalescedメモリアクセス

- 同じワープからのメモリアクセスは、可能な限り、一緒にされる

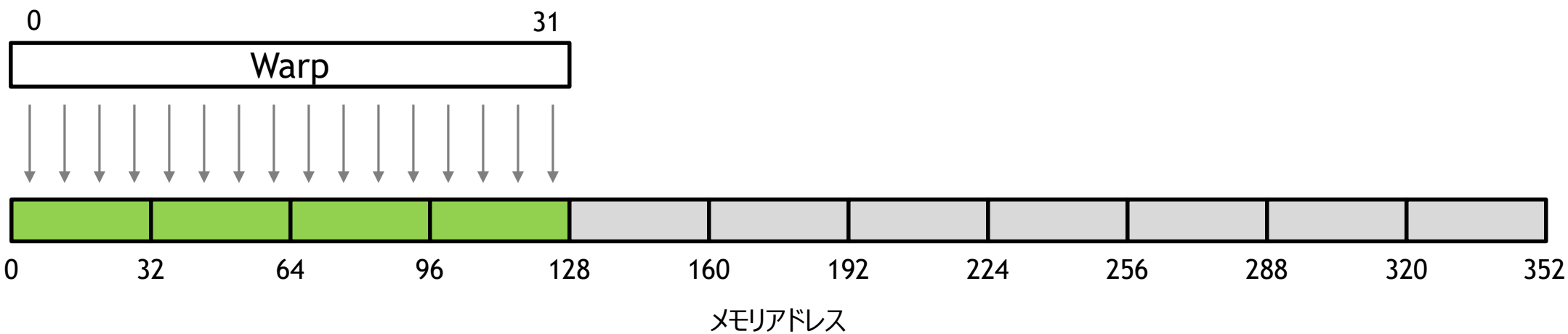
問題: 各ワープのLD/STが、何個のセクターにアクセスするのか

- セクターとは、メモリアクセスの粒度(大きさ)
- セクターサイズは32B or 64B　(*) 以降は32Bとして説明

メモリアクセスパターン

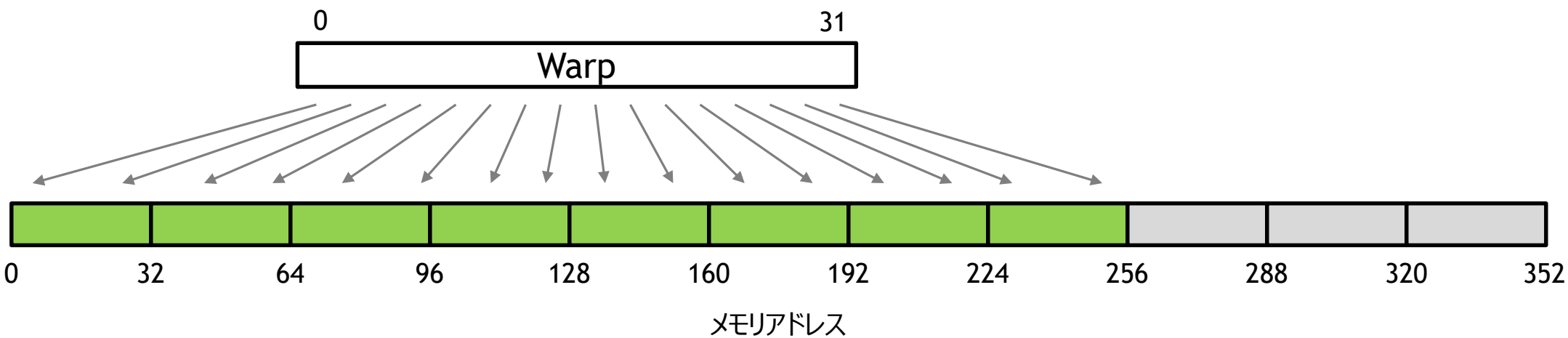
各スレッドが連続的に4Bアクセス(int, float)

4セクター



メモリアクセスパターン

各スレッドが連続的に8Bアクセス
(long, double, int2, float2)
8セクター



メモリアクセスパターン

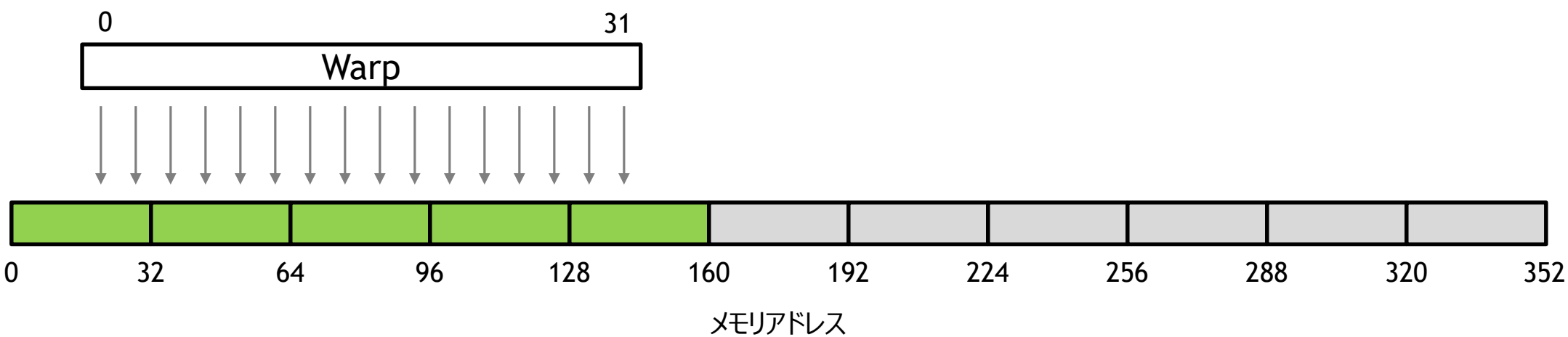
各スレッドが4Bアクセス、連続ではない

4セクター

メモリアクセスパターン

各スレッドが連続的に4Bアクセス、but 境界がずれている

5セクター

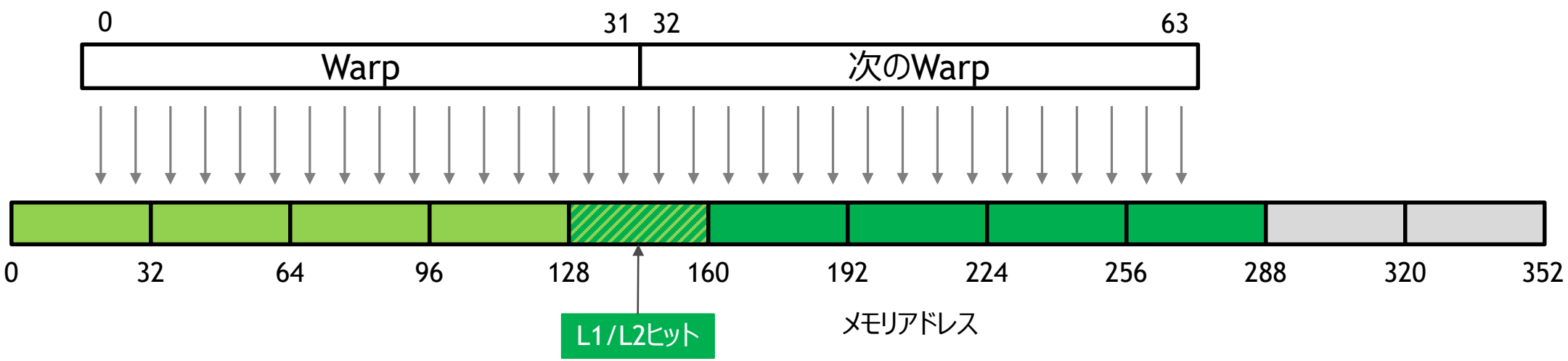


メモリアクセスパターン

多くの場合、次のワープが隣接セクターにアクセスするので、実質上問題無い

各スレッドが連続的に4Bアクセス、but 境界がずれている

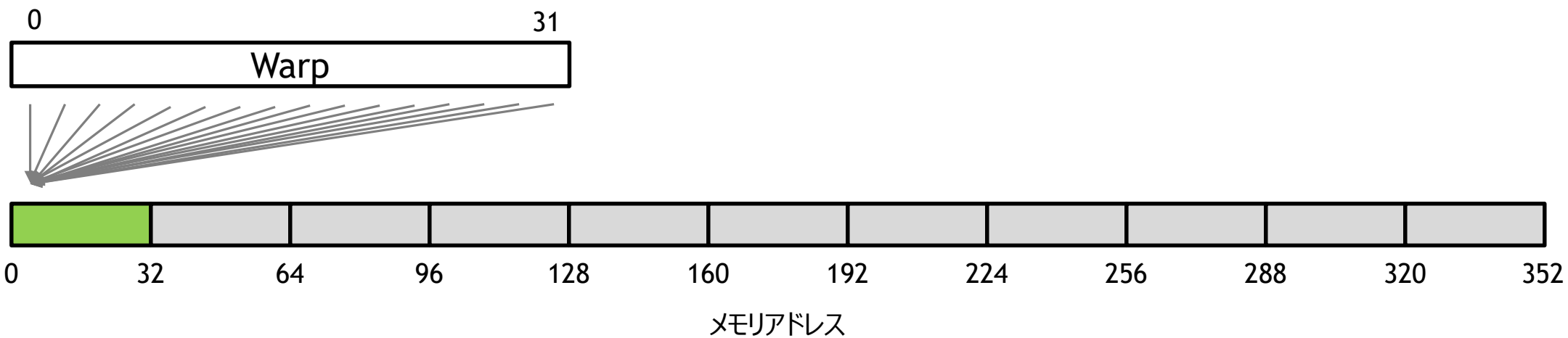
5セクター



メモリアクセスパターン

同じアドレス

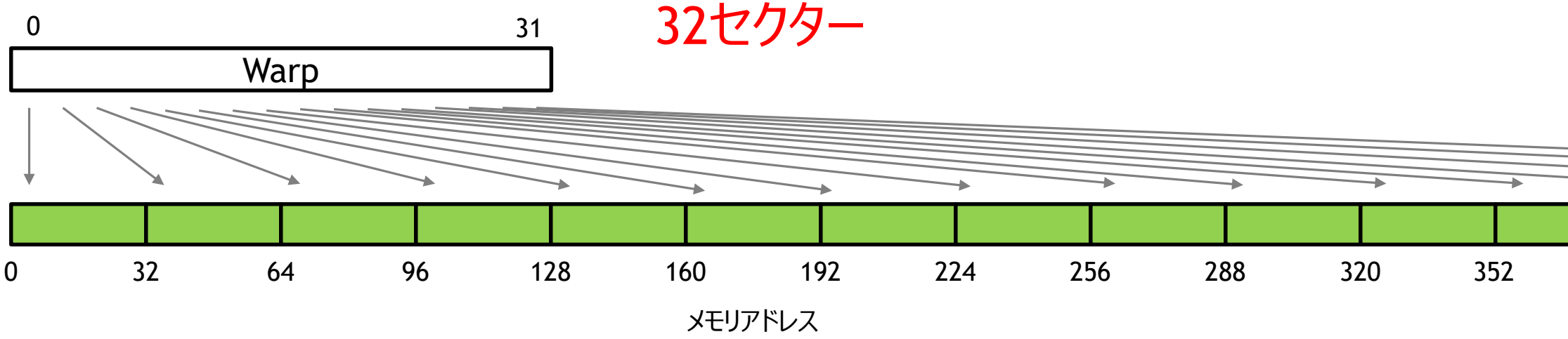
1セクター



メモリアクセスパターン

このアクセスパターンは、可能な限り、避ける必要がある
32B(1セクター)の内、4Bしか使わない、28Bは無駄

各スレッドが4Bアクセス、ストライド
32セクター



メモリアクセスパターン

プログラムのアクセスパターンの把握は重要

- ストライドアクセスになっていないか、プロファイラーで確認する。

もし、アクセスパターンが良くなかったら?

- スレッドへの処理割当てを変える (並列化方法の変更)
- データレイアウトを変更する (例: a[X][Y] → a[Y][X])
- ReadとWriteで二律相反になったら (例:行列転置)、Writeを優先する

大きなデータ型を使う (float → float2 → float4)

共有メモリ

各SM内にある、スクラッチパッドメモリ

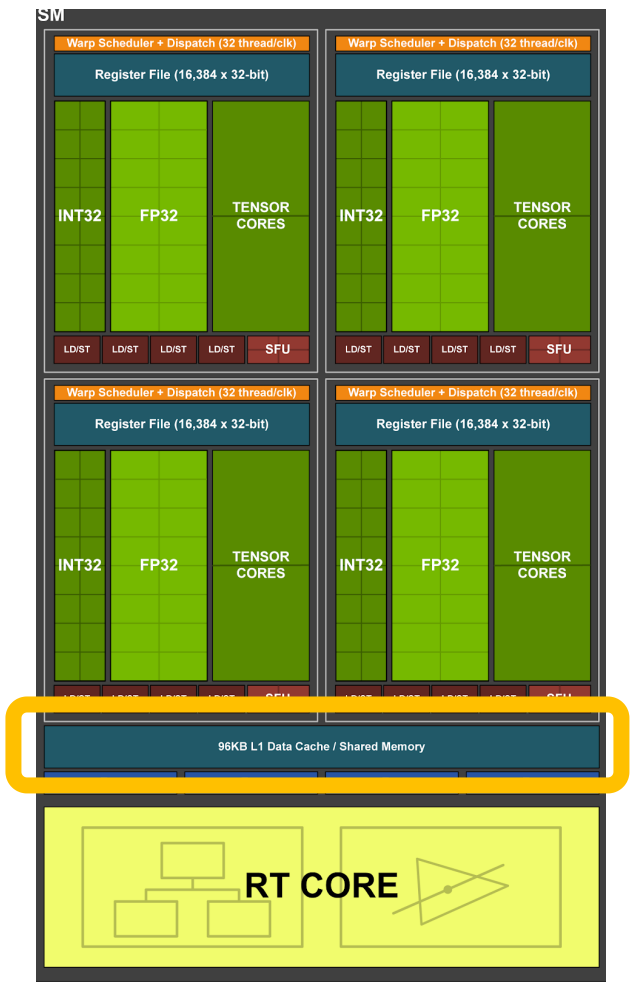
- ユーザーが明示的にデータの出し入れを制御できるキャッシュ

DRAMと比べて、高速なアクセス

- 短遅延: 20-40倍も高速
- バンド幅: 15倍程度の広帯域

どんなときに有効か?

- 同じスレッドブロック内のスレッド間での、高速なデータ送受（別スレッドブロック間では使えない）
- 頻繁にアクセスされるデータの保管（atomic操作など）



共有メモリ

VoltaとTuringで異なる点

	Volta (GV100)	Turing (TU102)
バンド幅	14 TB/s	8 TB/s
サイズ上限 (/スレッドブロック)	48, 96 KB	48 KB

Voltaでは48KB超の共有メモリを必要とするカーネルも実行できるが、Turingでは実行できない

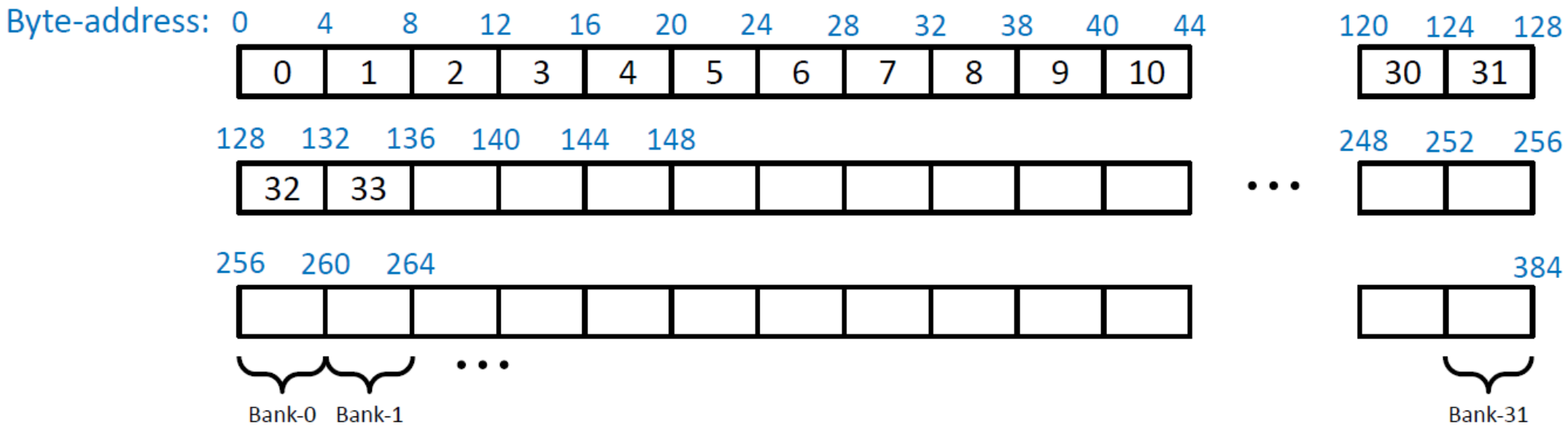
共有メモリのサイズは設定可能（ドライバが適切に選択するので、多くの場合、設定不要）

```
cudaFuncSetAttribute( kernel, cudaFuncAttributePreferredSharedMemoryCarveout, carveout );
```

共有メモリ

基本構造

粒度は4B、32バンク

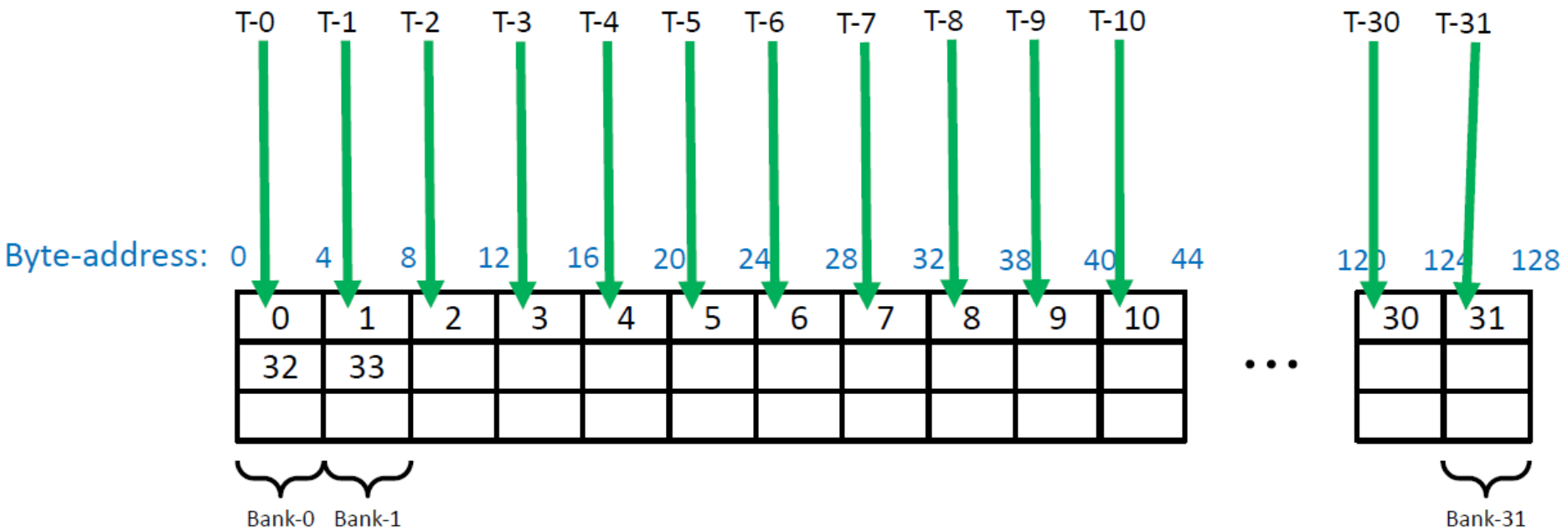


バンク競合すると性能低下 … バンク競合って何？

共有メモリ

バンク競合無し

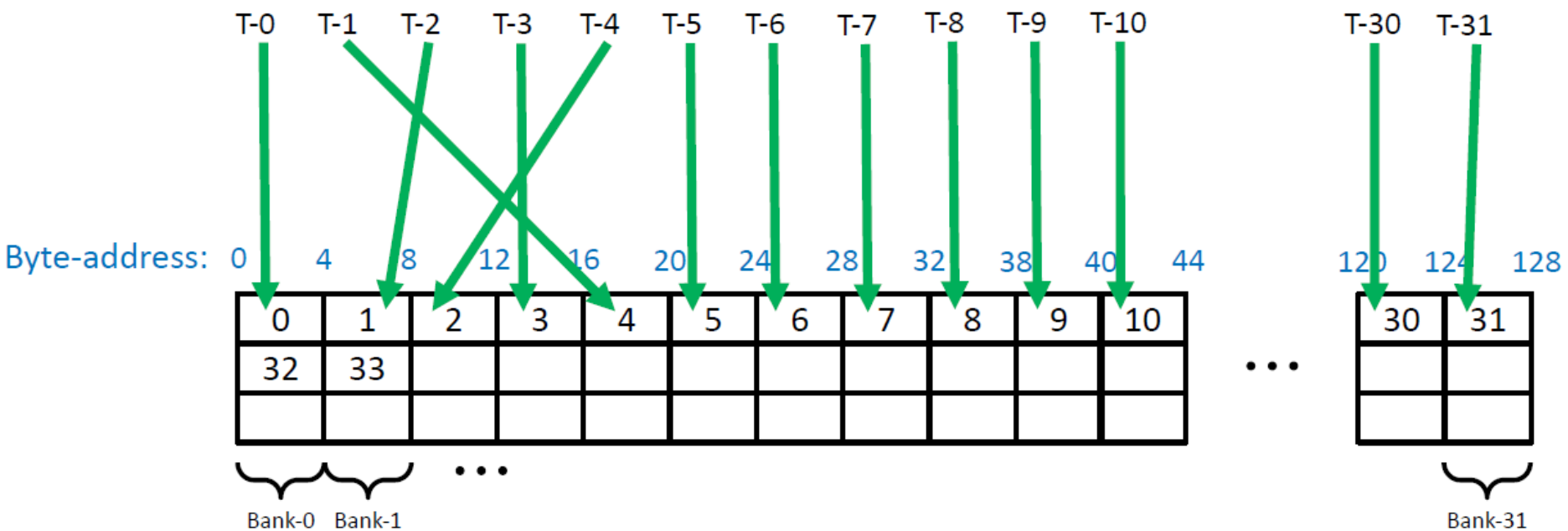
同じワープ内のスレッドが、別バンクにアクセスすれば、バンク競合は発生しない



共有メモリ

バンク競合無し

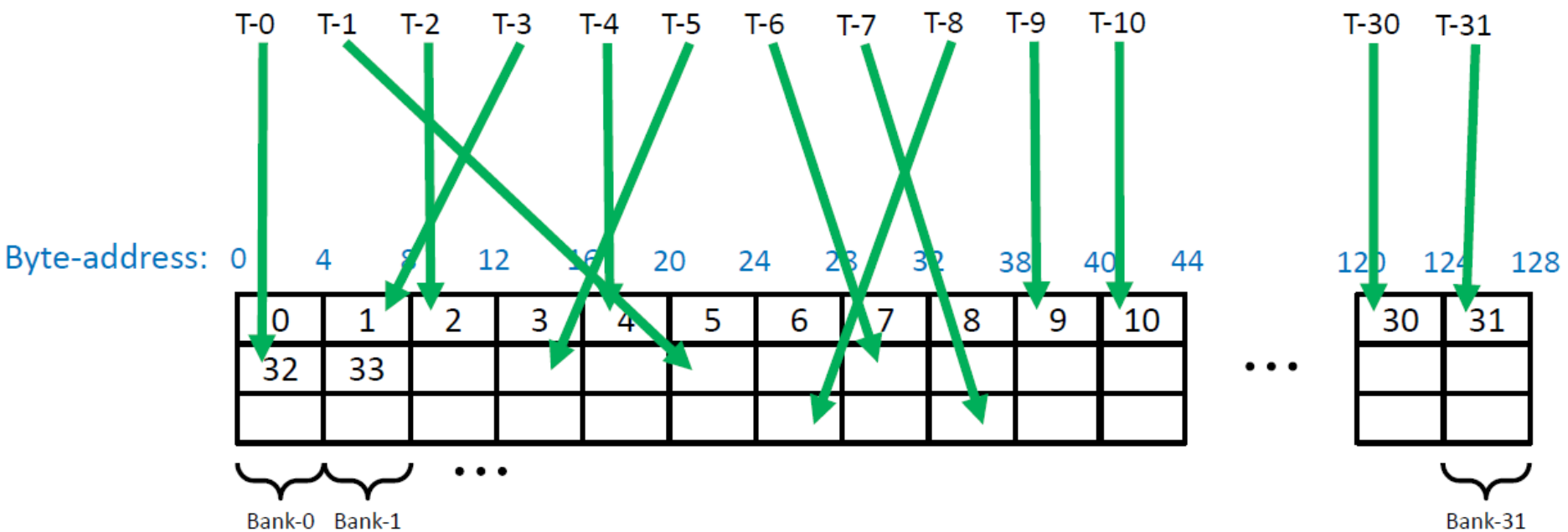
同じワープ内のスレッドが、別バンクにアクセスすれば、バンク競合は発生しない



共有メモリ

バンク競合無し

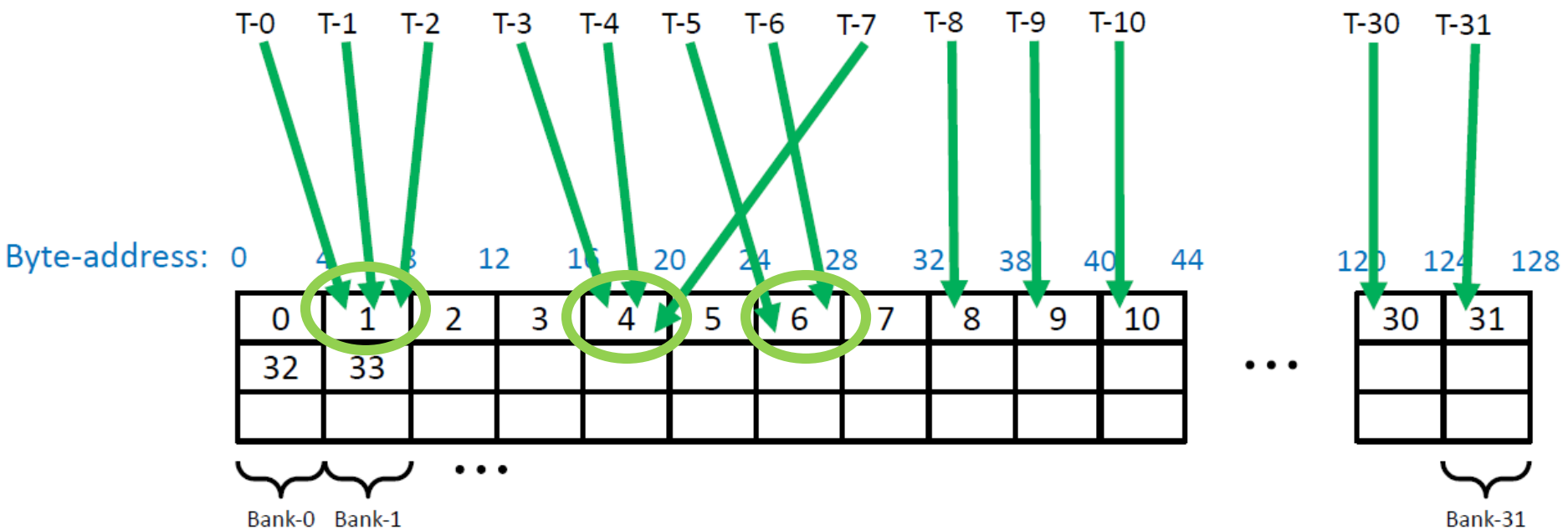
同じワープ内のスレッドが、別バンクにアクセスすれば、バンク競合は発生しない



共有メモリ

バンク競合無し

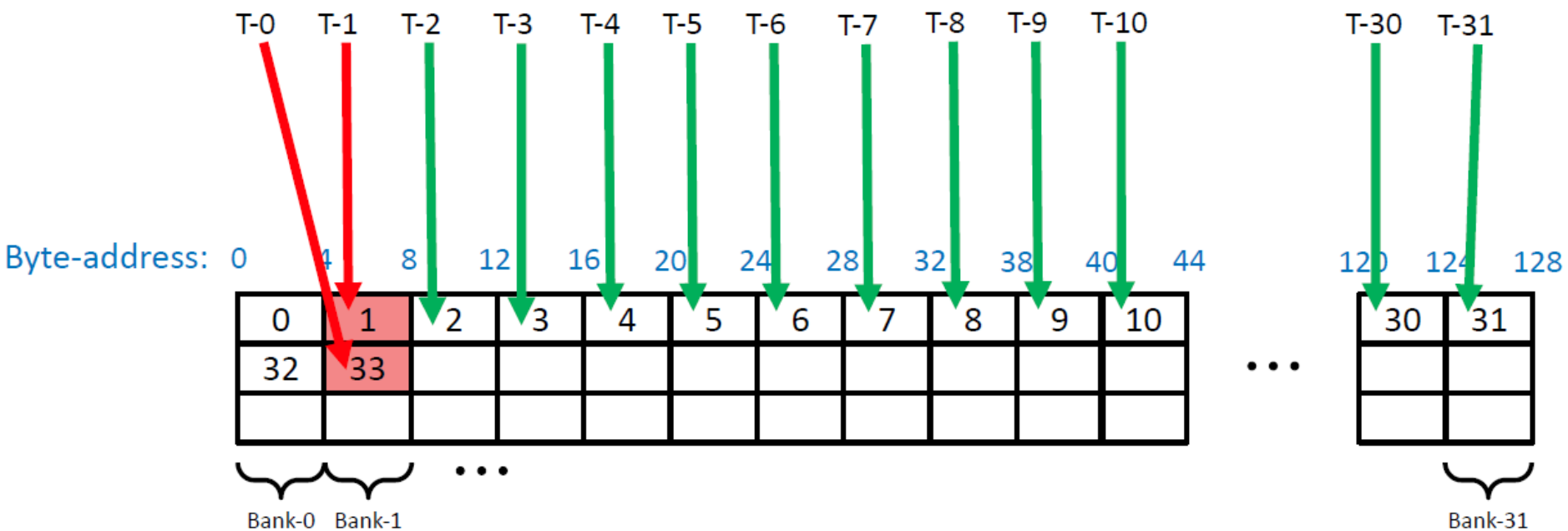
同じバンクにアクセスしても、それが同じアドレスであれば、性能は低下しない
(マルチキャスト)



共有メモリ

2way バンク競合

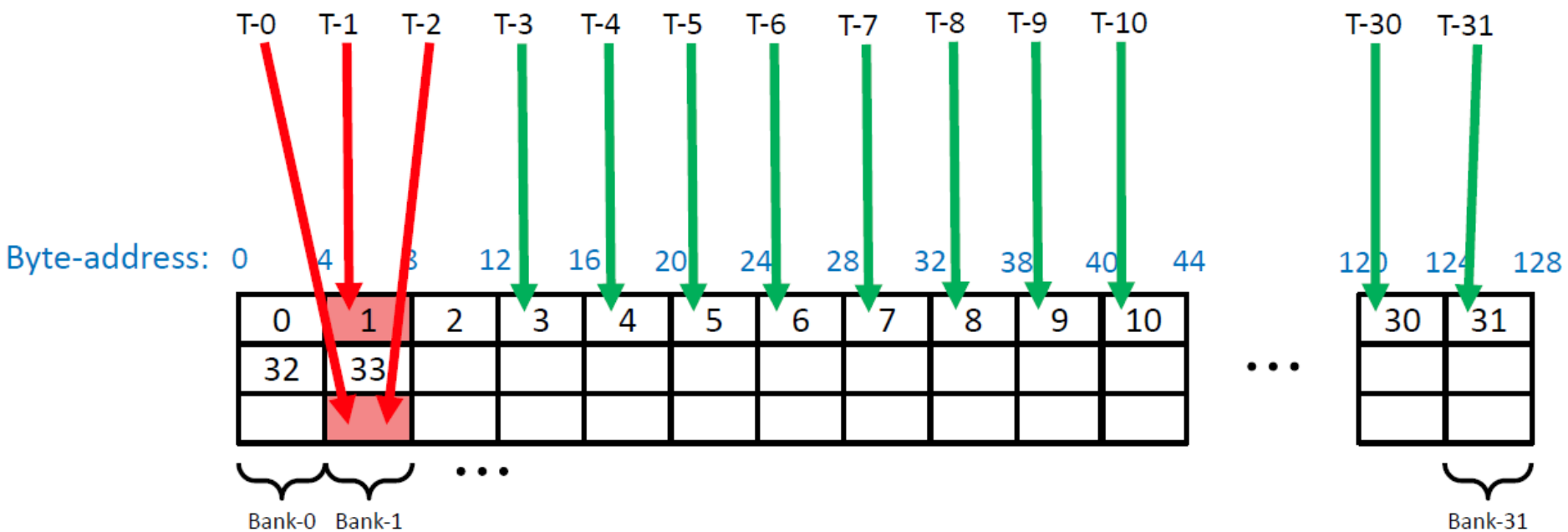
バンクが同じで、かつ、アドレスが異なると、バンク競合が発生



共有メモリ

2way バンク競合

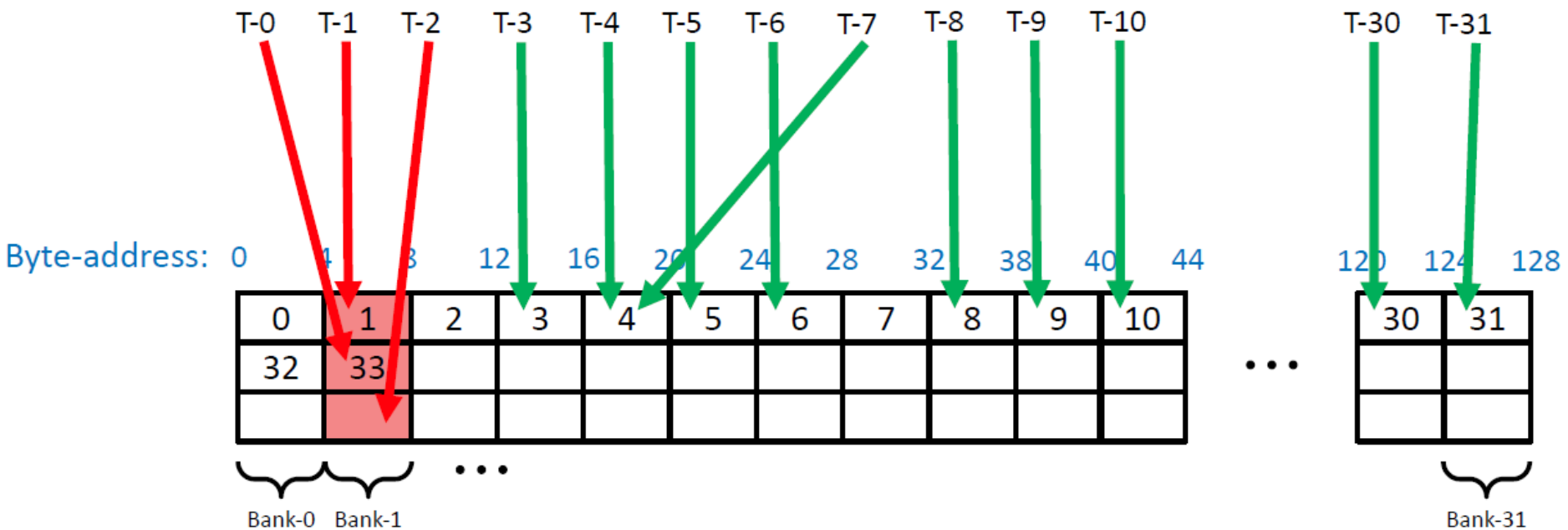
バンクが同じで、かつ、アドレスが異なると、バンク競合が発生



共有メモリ

3way バンク競合

バンクが同じで、かつ、アドレスが異なると、バンク競合が発生



共有メモリ

使用データ型による違い

4B以下: char, short, int, float

- 全32スレッドが同時にアクセス

8B: long long, double, int2, float2

- 2フェーズ: 最初のフェーズに前半16スレッドがアクセス、次フェーズに残り16スレッドがアクセス

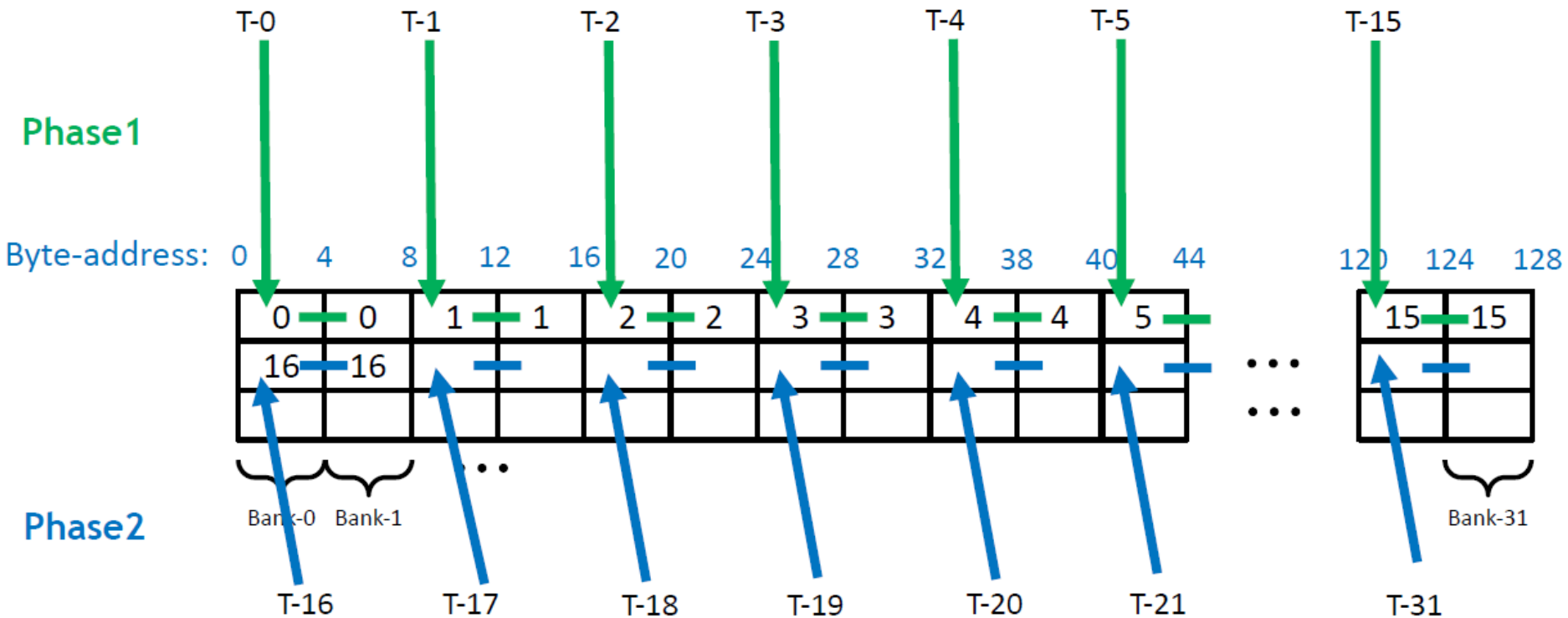
16B: int4, float4, double2

- 4フェーズ: 各フェースで8スレッドがアクセス

バンク競合は、各フェーズ内で発生する

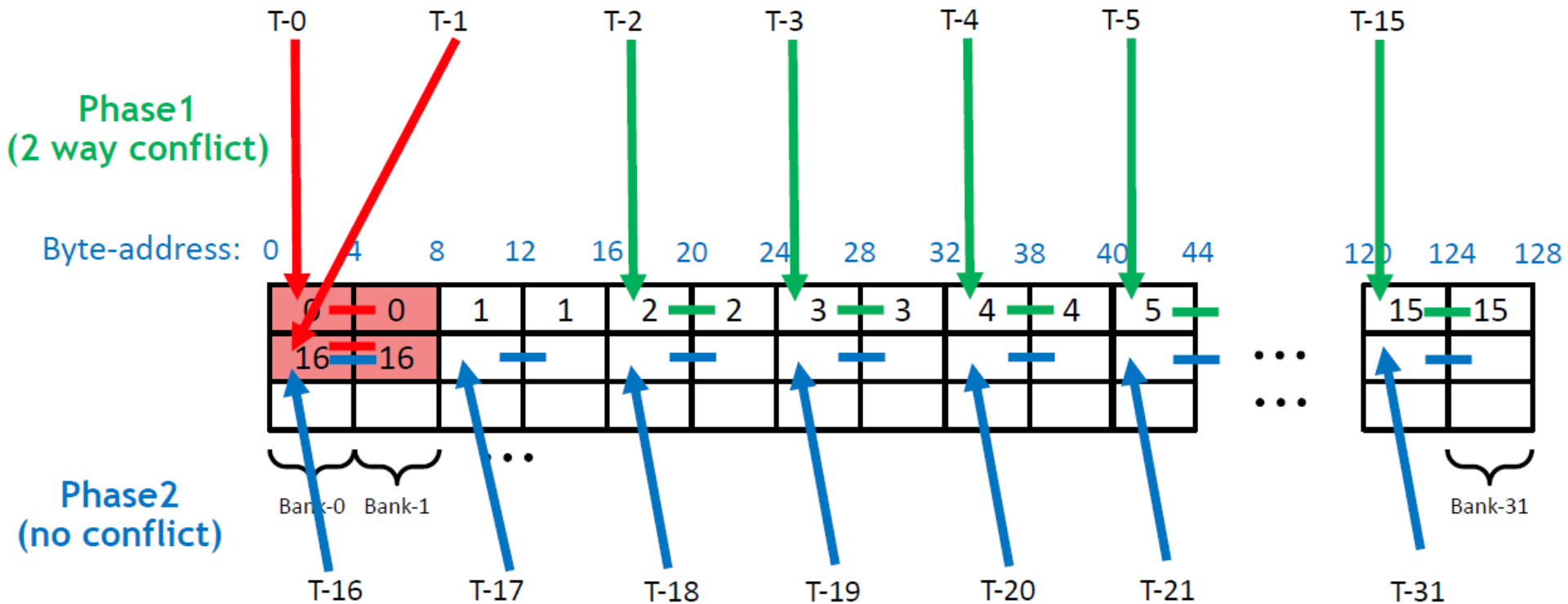
共有メモリ

8Bアクセス、バンク競合無し



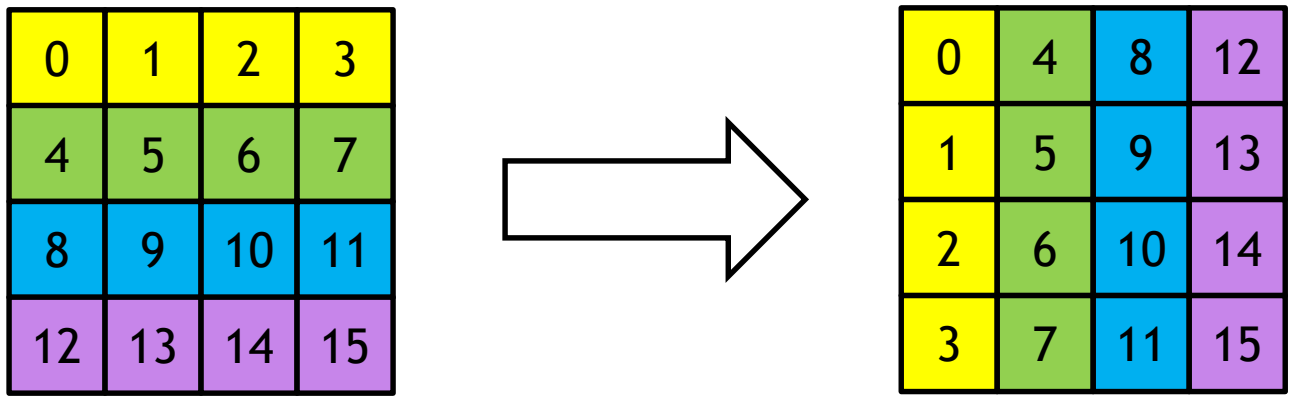
共有メモリ

8Bアクセス、2way バンク競合



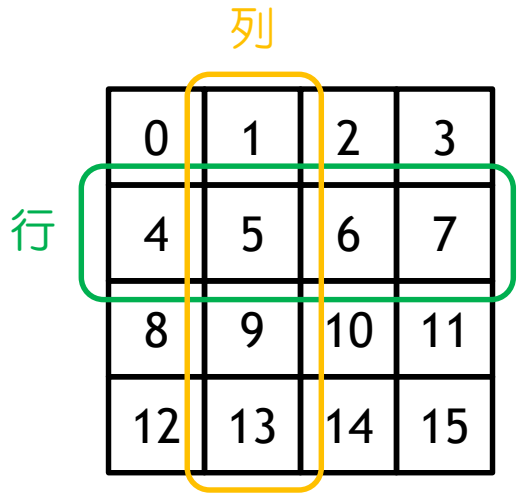
共有メモリ使用例

行列転置



共有メモリ使用例

行列転置



例: 行列サイズ: 32x32、データ型: float、スレッドブロック形状: 32x32 (1024スレッド)

共有メモリ実装:

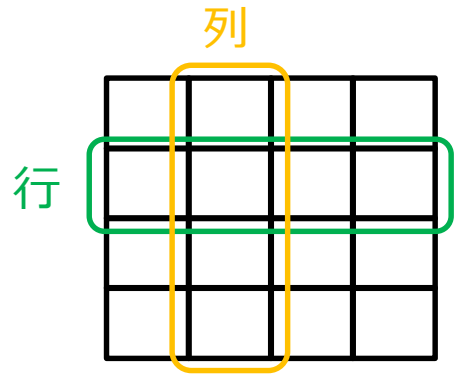
- 32x32形状の共有メモリを宣言 (例: __shared__ float sm[32][32])
- ワープ毎に、DRAM上の入力行列を**1行**読み出し、それを共有メモリの**1行**に書き込み
- スレッド間で同期 (__syncthreads)
- ワープ毎に、共有メモリ内の入力行列を**1列**読み出し、それを出力行列の**1行**としてDRAMに書き込み

これで、DRAMアクセスは行読み出し/行書き込みとなり、naïve実装より高速化

共有メモリへのアクセスは?

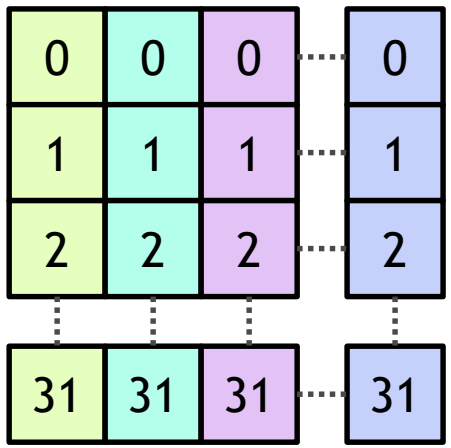
共有メモリ使用例

行列転置



ワープの共有メモリへの書き込み (行アクセス)

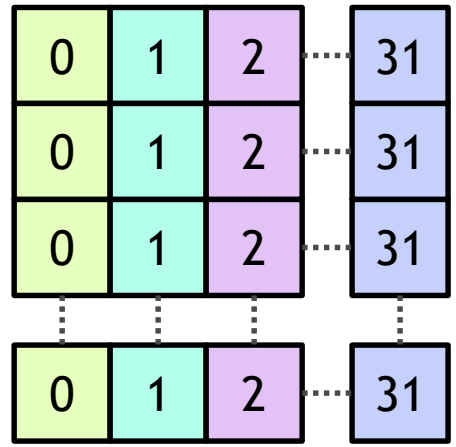
バンク競合無し



共有メモリ
__shared__ float sm[32][32])

ワープの共有メモリの読み出し (列アクセス)

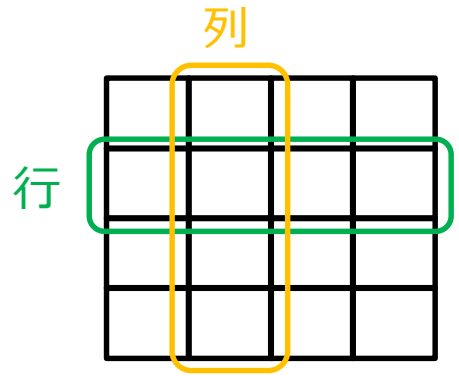
32way バンク競合



(*) 数字はワープ番号、色はバンク番号

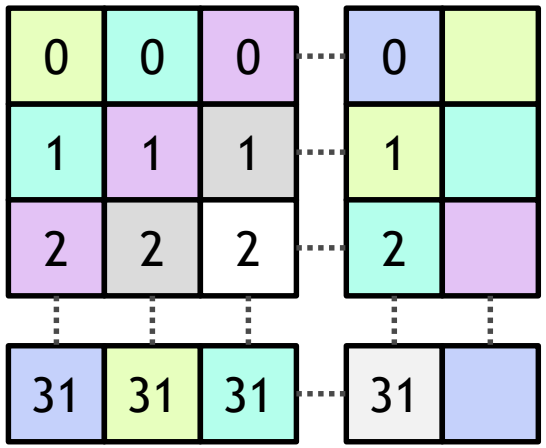
共有メモリ使用例

行列転置



ワープの共有メモリへの書き込み (行アクセス)

バンク競合無し



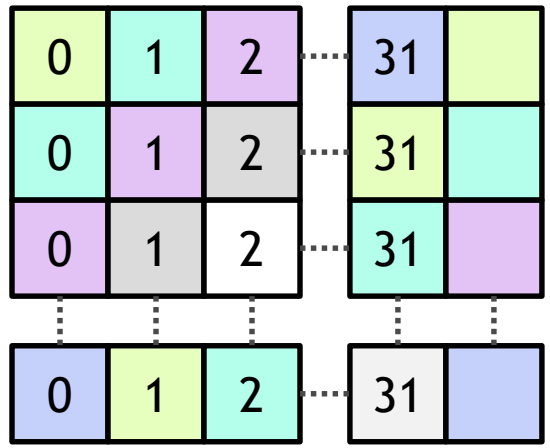
共有メモリ

__shared__ float sm[32][32+1]

パディング

ワープの共有メモリの読み出し (列アクセス)

バンク競合無し



(*) 数字はワープ番号、色はバンク番号

共有メモリ

プログラムを修正する必要はあるが、依然として、強力なリソース

- 高バンド幅 (Volta: 14TB/s、Turing: 8TB/s)
- 短遅延
- Voltaなら、スレッドブロックあたり96KBまで使うことが可能

バンク競合に注意

- バンク競合の発生状況はプロファイラーで確認できる

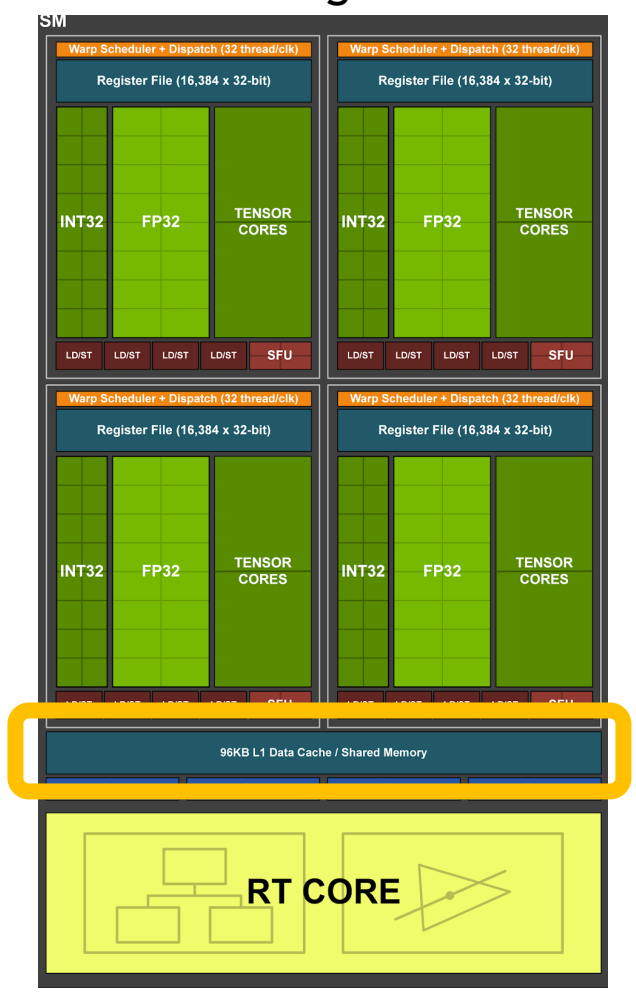
L1キャッシュ

Pascal → Volta/Turing

Volta/Turingから、L1キャッシュは共有メモリと統合 (Pascalは分離)

- PascalのL1キャッシュと比べて、広バンド幅、単遅延
- バンド幅: Pascal: <3 TB/s、Volta: 14 TB/s、Turing: 8 TB/s
- サイズは可変: Volta: 32～128 KB、Turing: 32 or 64 KB
  - 共有メモリのサイズ選択次第

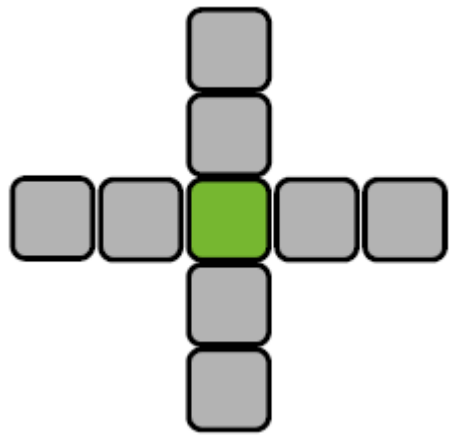
Pascalまでは共有メモリ使用が性能上必要なケースでも、
Volta/TuringはL1キャッシュが高速なので共有メモリを使わなくても性能が出る？



2Dステンシル

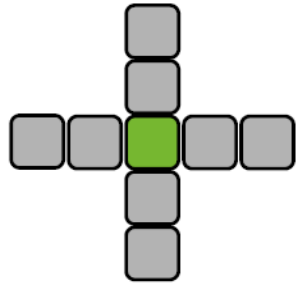
共有メモリ vs L1キャッシュ

```
index = ix + nx * iy;
res = coef[0] * in[index];
for (i = 0; i <= RADIUS; i++) {
    res += coef[i] * (in[index-i] +
                      in[index+i] +
                      in[index-i*nx] +
                      in[index+i*nx]);
}
out[index] = res;
```



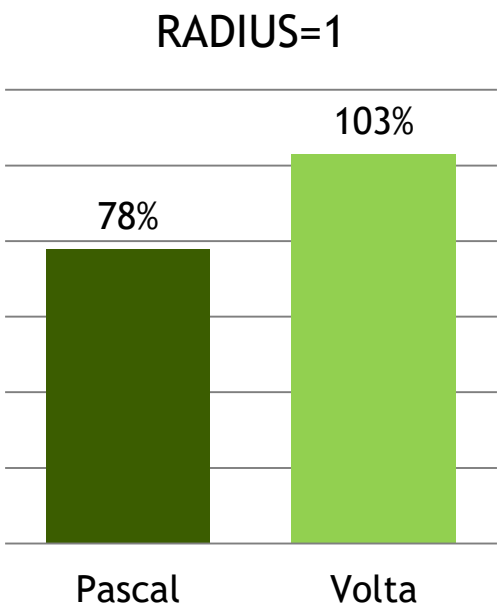
共有メモリ実装:

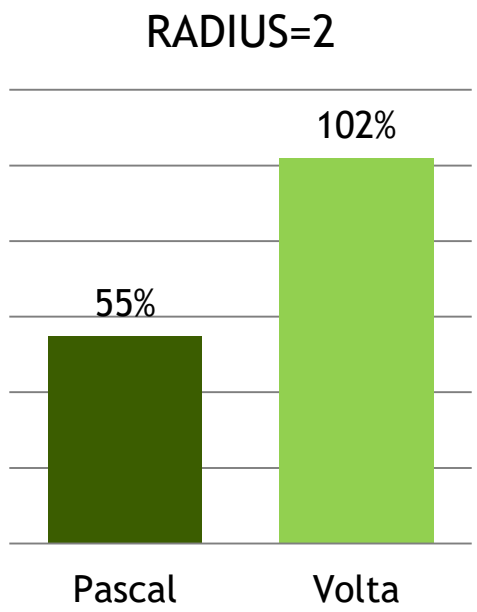
- スレッドブロック内のスレッドから参照される配列inの要素を、DRAMから読み出し、共有メモリに保存
- 配列inの要素は、共有メモリから読み出してステンシル計算を実行



2Dステンシル

共有メモリ実装に対する、L1キャッシュ実装の相対性能



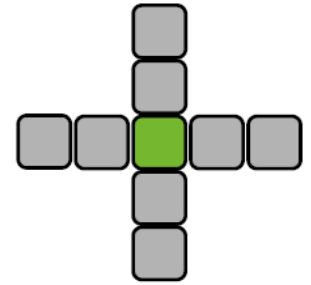


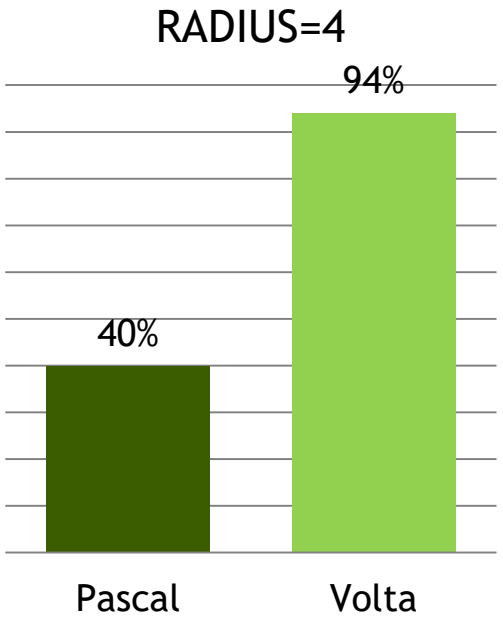
Pascalでは、常に共有メモリ実装が高速

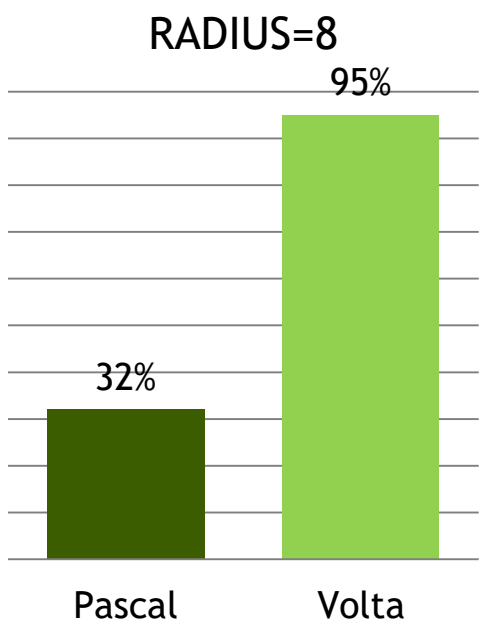
Voltaでは、半径が小さい場合は、L1キャッシュ実装の方が高速

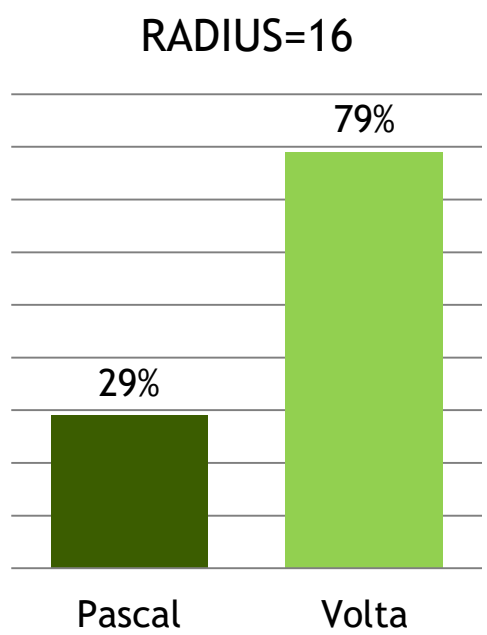
2Dステンシル

共有メモリ版に対する、L1キャッシュ版の相対性能









半径が大きくなると、Voltaでも、共有メモリ実装が高速

しかし、Pascalと比べると、共有メモリ実装とL1キャッシュ実装の性能差は少ない


VOLTAとTURING

VOLTA AND TURING

Pascalから進化、多くの機能を共有

Volta (cc70)

For HPC and Deep Learning

Tesla V100
(GV100)

Turing (cc75)

For Graphics and Deep Learning

QUADRO RTX6000
(TU102)

Tesla P100 (GP100)

QUADRO P6000 (GP102)


NVIDIA