GPU HOT TIPS: マルチGPUでプログラムを加速する方法

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

NVIDIA DGX-1V

8 x Tesa V100

NVLINK

2 x Intel Xeon

4 x EDR InfiniBand

3200 Watt

NVIDIA DGX-1V

8 x Tesla V100

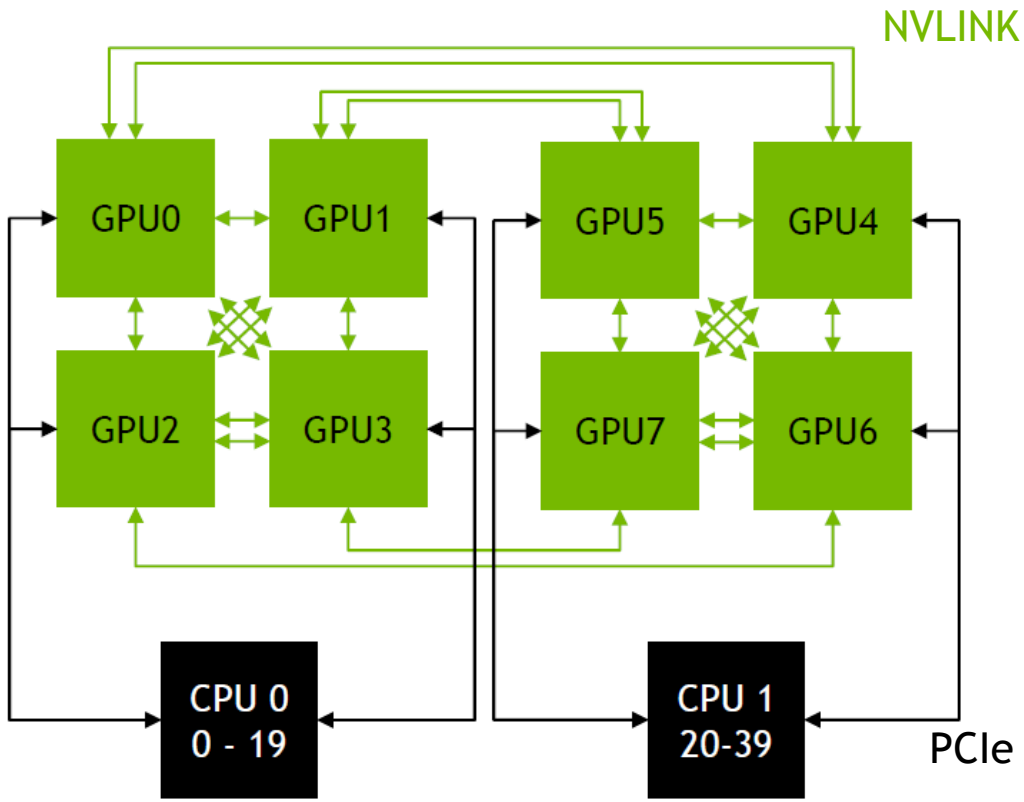
NVLINKで全結合された4 GPUブロック x 2

6 NVLINK lanes per GPU

25+25 GB/s per lane

隣接GPUメモリへのload/store/atomics

バルクデーテ転送ようの高速コピーエンジン



マルチGPU

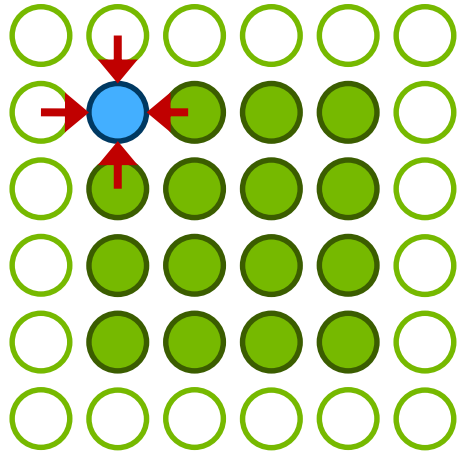
- シングルGPUでは時間のかかる問題を、もっと短い時間で解きたい
  - DLトレーニング、リアルタイム処理
- シングルGPUのメモリには収まらない、もっと大きな問題を扱いたい
  - 大規模シミュレーション
- マルチGPUノードがあるので、活用したい

サンプルプログラム: ヤコビ反復法

シングルGPU

収束条件を満たすまで、以下の計算を繰り返す

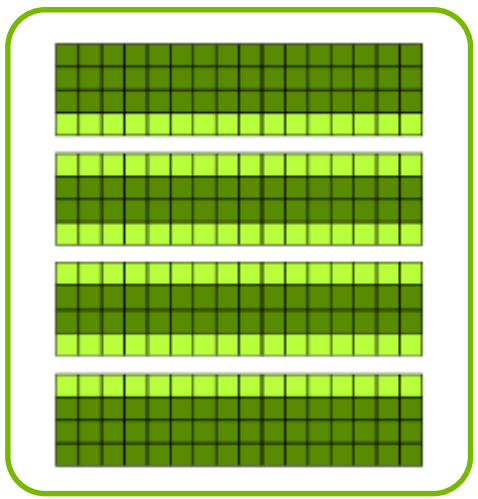
```
for ( iy = 1; iy < ny-1; iy++ )
    for ( ix = 1; ix < nx-1; ix++ )
        a_new[ ix + iy*nx ] = 0.25 * ( a[ ix + (iy-1)*nx ] + a[ ix + (iy+1)*nx ]
                                     + a[ (ix-1) + iy*nx ] + a[ (ix+1) + iy*nx ] );
```



マルチGPUで解く

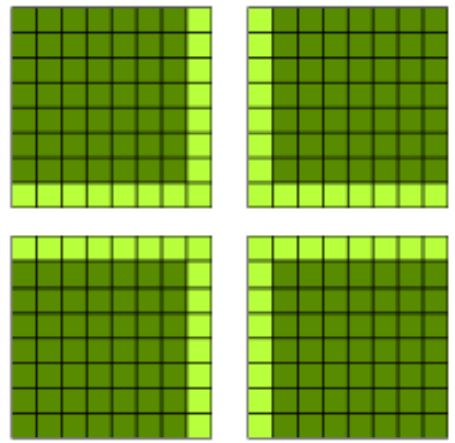
問題(2D領域)を分割して、各GPUに割り当てる

1D領域分割



- 通信相手が少ない
- 並列数が少ない場合に有効

2D領域分割



- 通信量が少ない
- 並列数が多い場合に有効

ヤコビ反復法

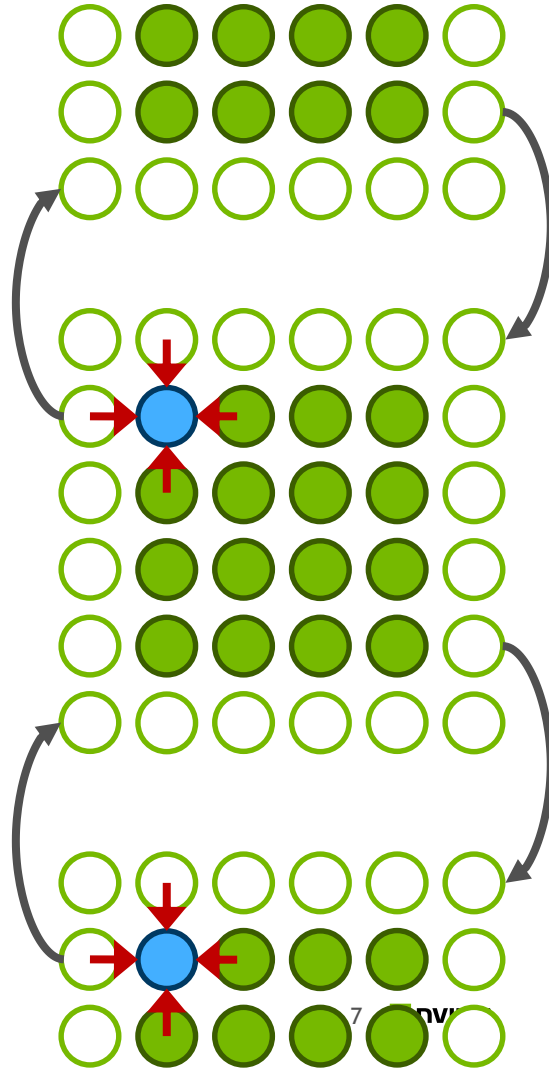
マルチGPU

収束条件を満たすまで、以下の計算と通信を繰り返す

計算: 自分が担当する領域に関して、以下の計算を行う

```
for ( iy = 1; iy < ny-1; iy++ )
    for ( ix = 1; ix < nx-1; ix++ )
        a_new[ ix + iy*nx ] = 0.25 * ( a[ ix + (iy-1)*nx ] + a[ ix + (iy+1)*nx ]
                                     + a[ (ix-1) + iy*nx ] + a[ (ix+1) + iy*nx ] );
```

通信: 更新した要素の境界部分を、隣接GPUに送る（受け取る）



マルチGPUを使う方法

シングル・スレッド

- 1つのスレッドが、複数GPUを管理

マルチ・スレッド

- 複数スレッドで、複数GPUを管理（1 GPU per 1スレッド）

マルチ・プロセス

- 複数プロセスで、複数GPUを管理（1 GPU per 1プロセス）

シングル・スレッドでマルチGPUを使う

GPUカーネル (ヤコビ反復法)

```
__global__ void kern_jacobi( ... ) {
    int ix = ...;
    int iy = ...;
    if ( out of range ) return;

    new_val = 0.25 * ( a[ ix + (iy-1)*nx ] + a[ ix + (iy+1)*nx ]
                     + a[ (ix-1) + iy*nx ] + a[ (ix+1) + iy*nx ] );

    a_new[ ix + iy*nx ] = new_val

    my_error = fabs( new_val - a[ ix + iy*nx ] );
    atomicMax( error, my_error );
}
```

各スレッドの担当箇所を決定
領域内か確認

担当要素の計算

計算結果をwrite

収束条件の計算
(最大差分)

シングルGPUの場合

```
while ( *error > tol ) {
      cudaMemset( error_d, 0, … );
      kern_jacobi<<< ... >>>( a_new, a, error_d, nx, ny );
      cudaMemcpy( error, error_d, … );
      swap( a_new, a );
}
```

最大誤差の初期化

GPUカーネルの実行

最大誤差をCPUに回収

ポインタの入れ替え

マルチGPUの場合

```
while ( error > tol ) {
    for ( dev_id = 0; dev_id < num_devs; dev_id++ ) {
        cudaSetDevice( dev_id );
        cudaMemsetAsync( error_d[dev_id], 0, … );
        kern_jacobi<<<blocks, threads, …>>>( a_new[dev_id], a[dev_id], error_d[dev_id],
                                             nx, iy_start[dev_id], iy_end
        cudaMemcpyAsync( error_h[dev_id], error_d[dev_id], … );

        prev_id = (dev_id - 1) % num_devs;
        cudaMemcpyAsync( a_new[prev_id] + iy_end[prev_id]*nx,
                         a_new[dev_id] + iy_start[dev_id]*nx, … );
        next_id = (dev_id + 1) % num_devs;
        cudaMemcpyAsync( a_new[next_id],
                         a_new[dev_id] + (iy_end[dev_id]-1)*nx, … )
    }

    error = 0.0;
    for ( dev_id = 0; dev_id < num_devs; dev_id++ ) {
        cudaSetDevice( dev_id );  cudaDeviceSynchronize();
        error = max(error, *(error_h[dev_id]));
        swap( a_new[dev_id], a[dev_id] );
    }
}
```

複数GPU、それぞれに指示
cudaSetDeviceでGPU指定

GPUカーネル実行
非同期API使用、
CPUがGPUの実行完了
を待たないようにする

通信 (後述)

各GPUの処理完了を待つ
cudaDeviceSynchronize

最大誤差をCPUに回収

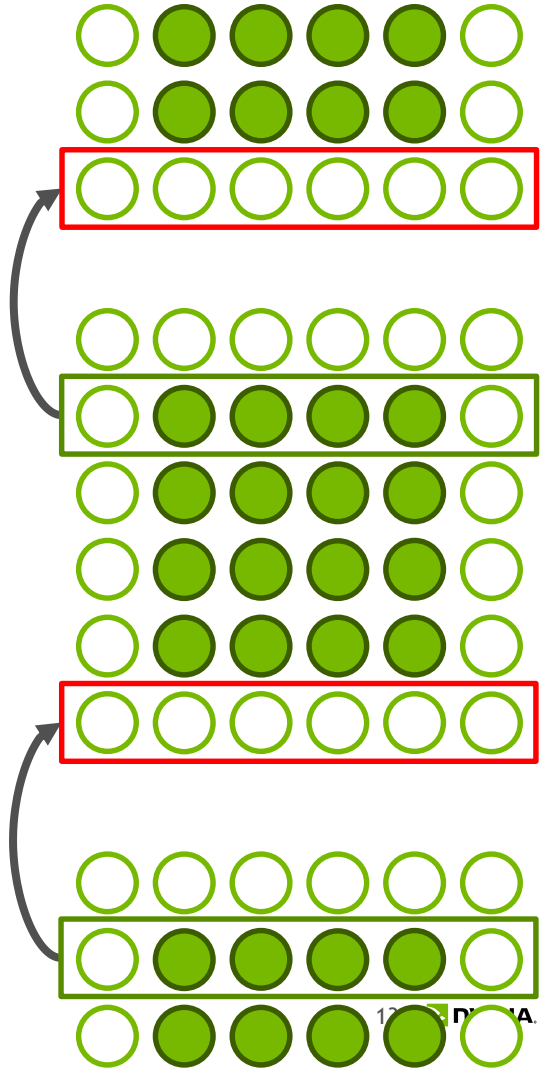
通信: 境界交換

```
cudaMemcpyAsync(
    a_new[prev_id] + iy_end[prev_id]*nx,
    a_new[dev_id] + iy_start[dev_id]*nx,
    … );
```

コピー先

コピー元

(*) 前のGPUが、次のサイクルに使うデータを送る

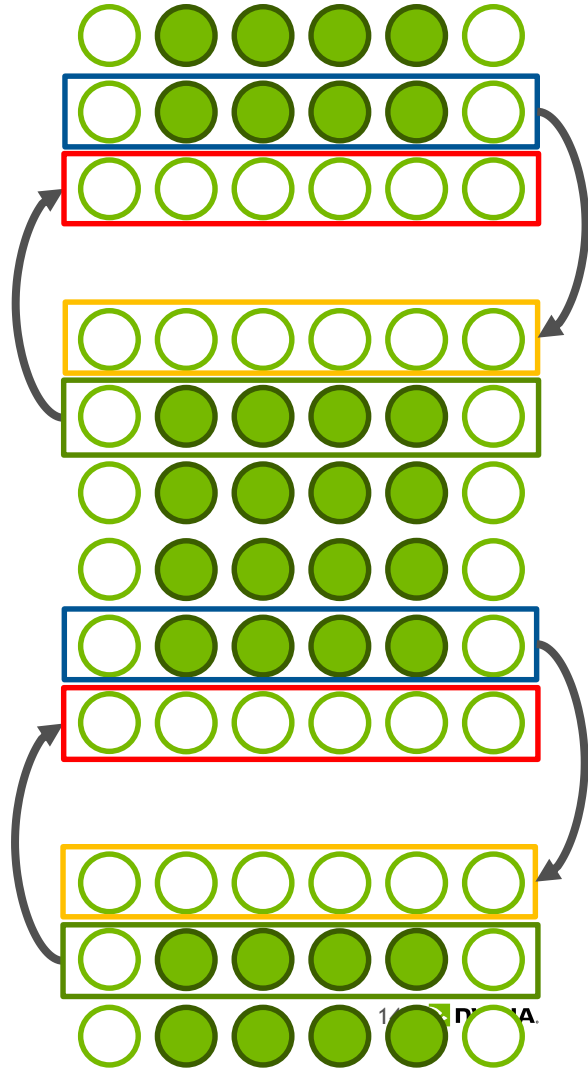


通信: 境界交換

```
cudaMemcpyAsync(
    a_new[prev_id] + iy_end[prev_id]*nx,
    a_new[dev_id] + iy_start[dev_id]*nx,
    … );
```

(*) 次のGPUが、次のサイクルに使うデータを送る

```
cudaMemcpyAsync(
    a_new[next_id],          コピー先
    a_new[dev_id] + (iy_end[dev_id]-1)*nx,   コピー元
    … );
```



並列性能の、性能指標と測定方法

T_s … 逐次実行時間 (シングルGPU)

T_p … 並列実行時間 (マルチGPU)

P … プロセス数 (GPU数)

$S = T_s/T_p$ … スピードアップ (理想はP)

$E = S/P$ … 並列化効率 (理想は1.0)

ストロング・スケーリング

- 問題サイズを固定
- プロセス数が増えるほど、プロセスあたり担当領域が小さくなる

ウィーク・スケーリング

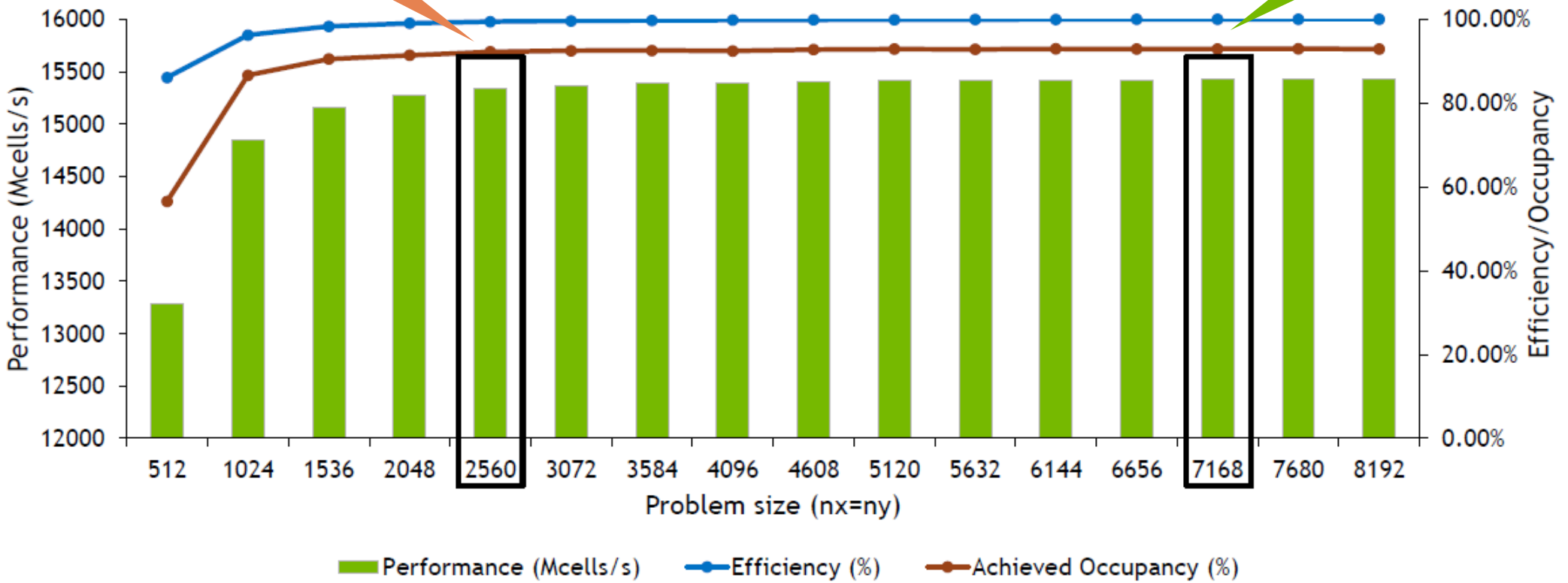
- プロセス数に比例して問題サイズを大きくする
- プロセスあたりの問題サイズは一定

シングルGPU性能

問題サイズと性能

8 GPUのときの
各GPUの問題サイズ

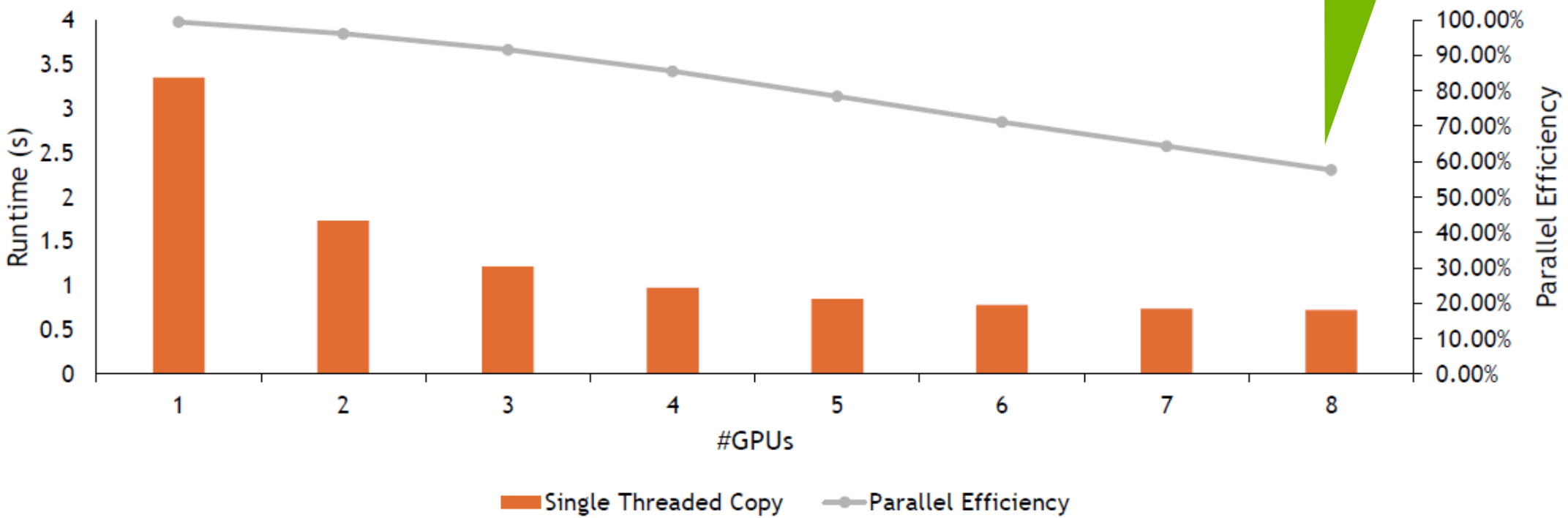
問題サイズは
7168を使用



マルチGPU (シングル・スレッド)

DGX-1V, 1000反復

8 GPUのときの
並列化効率は
6割弱

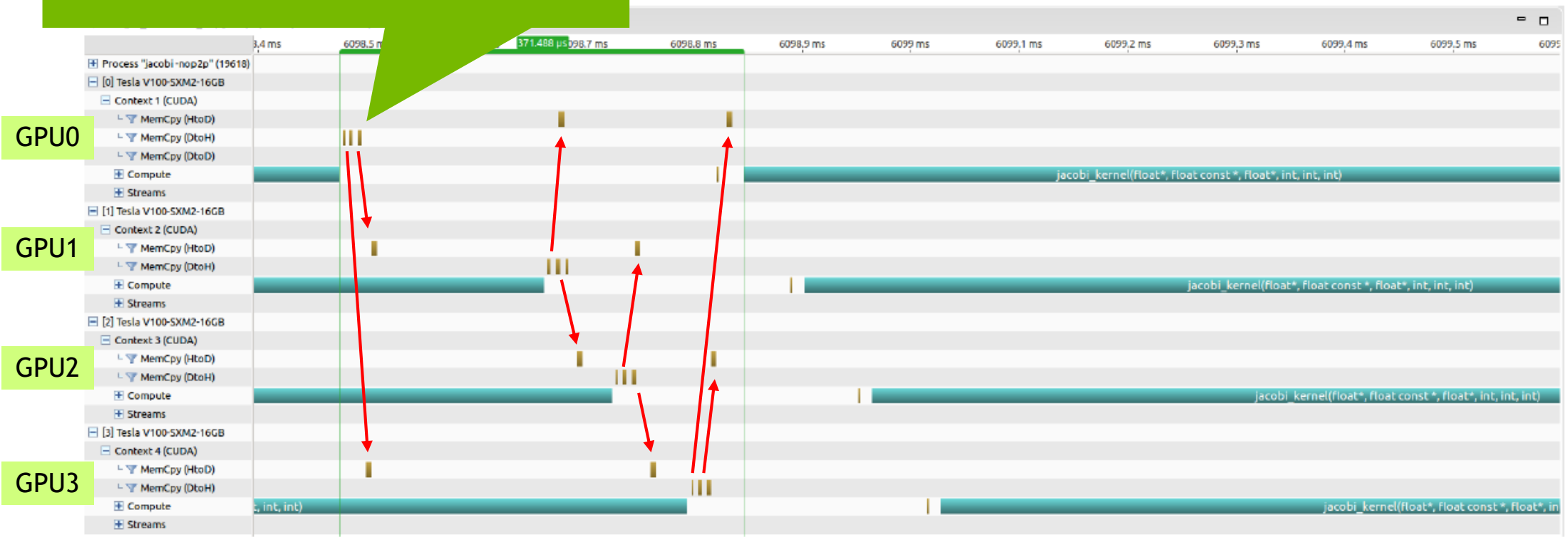


PU (シングル・スレッド)

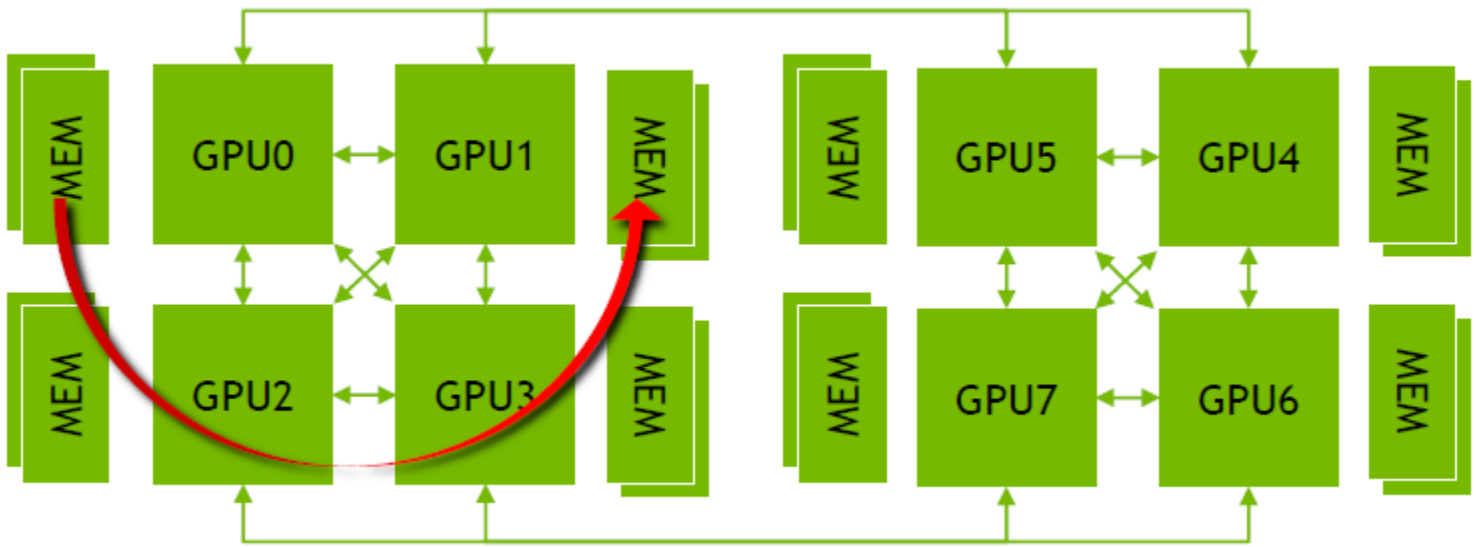
NVVP Timeline

MemCpy(DtoH)とMemCpy(HtoD)が使われている

GPU間の通信(コピー)が、ホストメモリ経由になっている



PEER-TO-PEER(P2P) メモリコピー



ホスト(CPU)を介さないで、デバイス(GPU)間で、直接データ転送

GPU間がNVLINK接続なら、NVLINK使用 (そうでなければ、PCIe使用)

P2Pの設定

自GPUを選択
(dev_id)

```
for ( dev_id = 0; dev_id < num_devs; dev_id++ ) {
    cudaSetDevice( dev_id );

    prev_id = (dev_id - 1) % num_devs;
    cudaDeviceCanAccessPeer( &canAccessPeer, dev_id, prev_id );
    if ( canAccessPeer )
        cudaDeviceEnablePeerAccess( prev_id, 0 );

    next_id = (dev_id + 1) % num_devs;
    cudaDeviceCanAccessPeer( &canAccessPeer, dev_id, next_id );
    if ( canAccessPeer )
        cudaDeviceEnablePeerAccess( next_id, 0 );
}
```

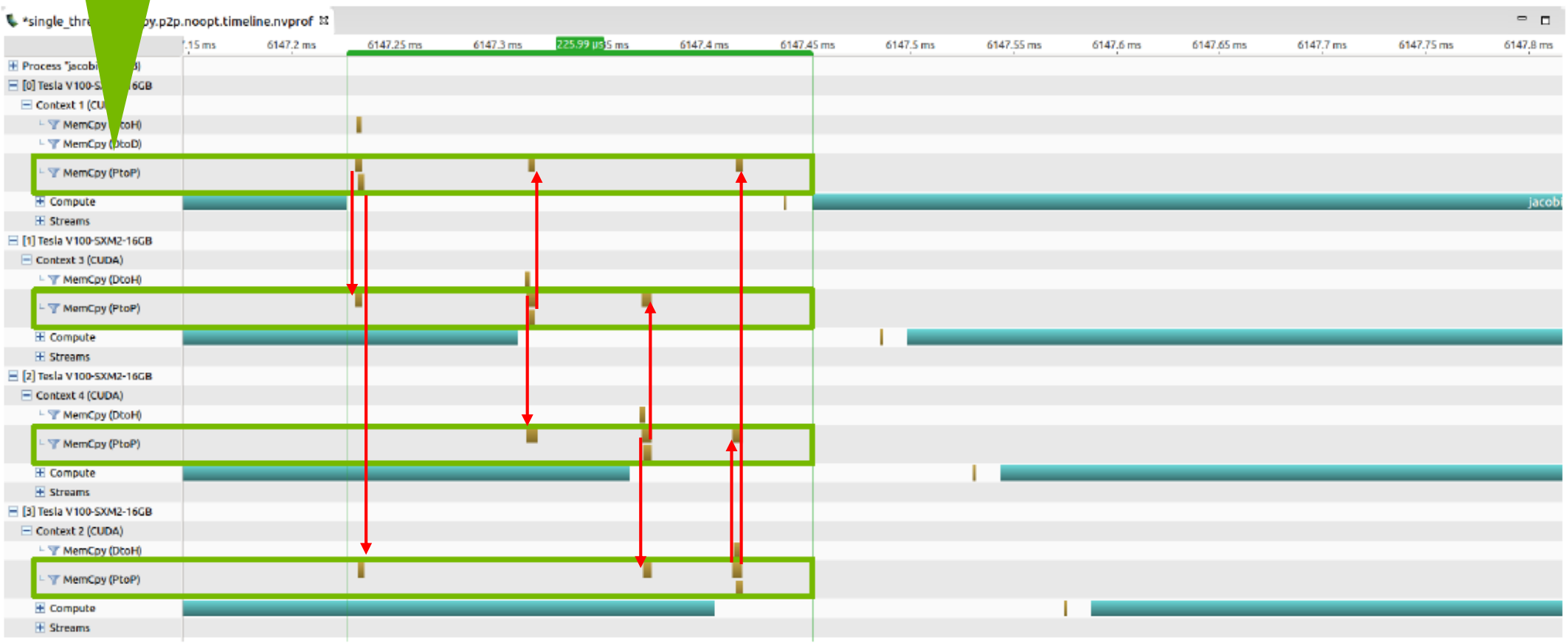
自GPUから、
別GPU(prev_id)への
P2P可否を確認

自GPUから、
別GPU(prev_id)への
P2Pを可能に

MemCpy(PtoP)
が使われている

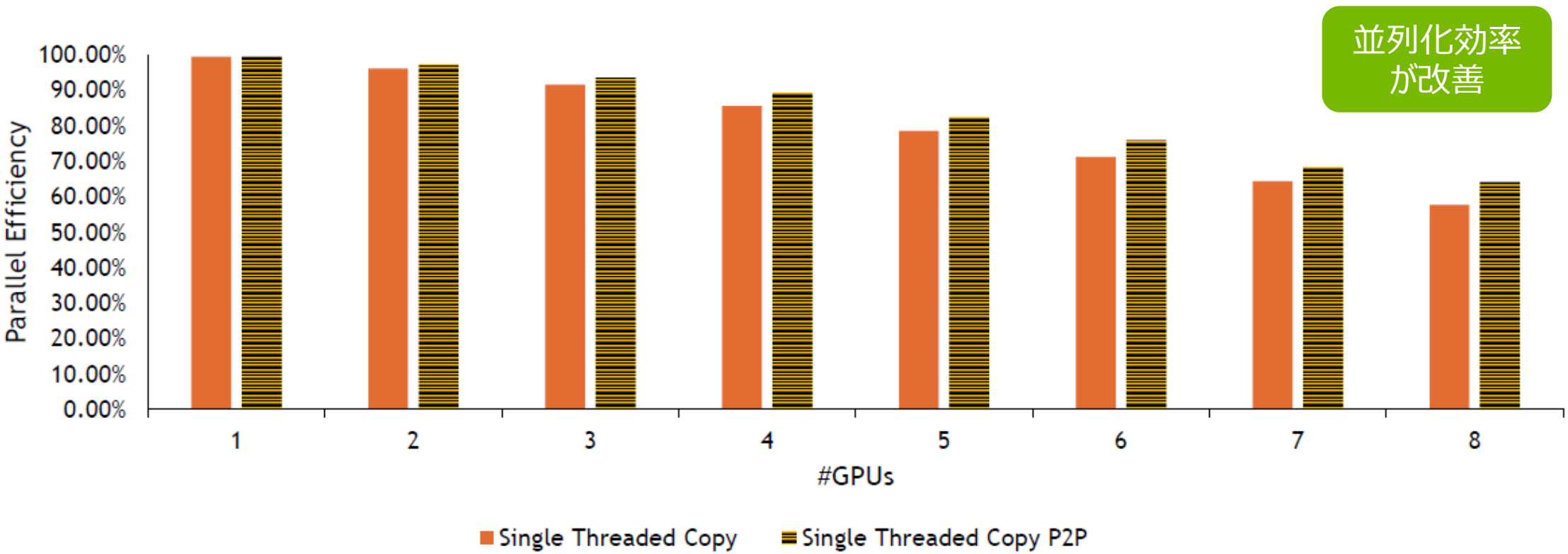
P2Pメモリコピーを Enabled

NVVP Timeline



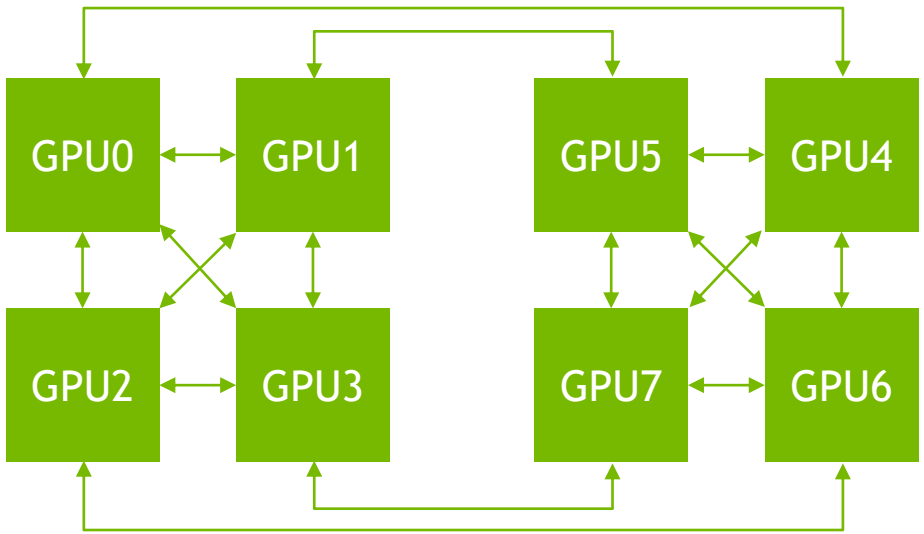
マルチGPU (シングルスレッド)

DGX-1V, 1000反復



NVLINKは使われているのか?

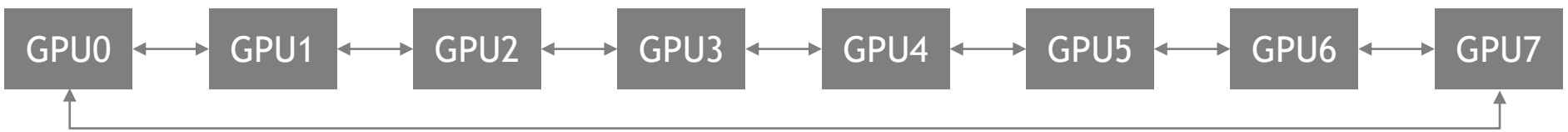
NVLINK接続形態



NVLINKにルーティング機能は無い

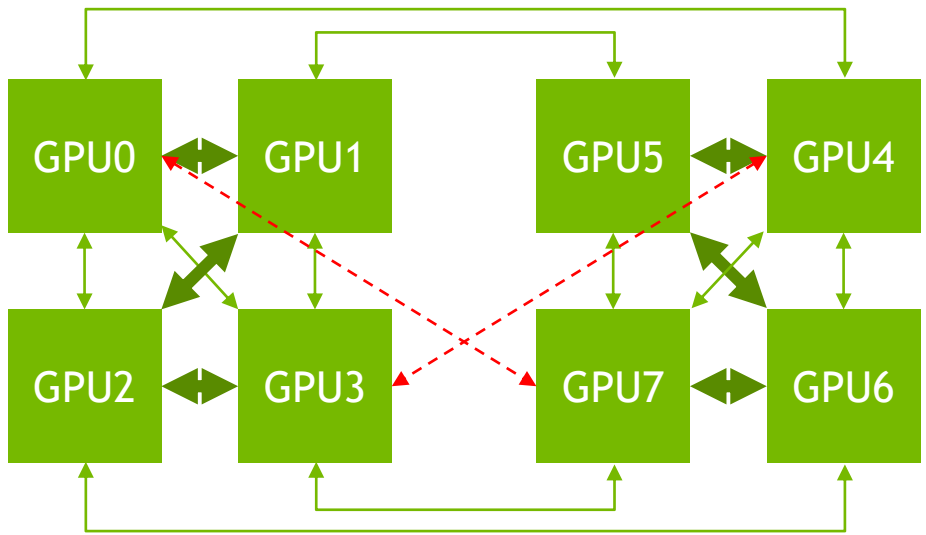
直接接続ではないGPU間のデータコピーに、NVLINKは使用されない

ヤコビ反復法の通信

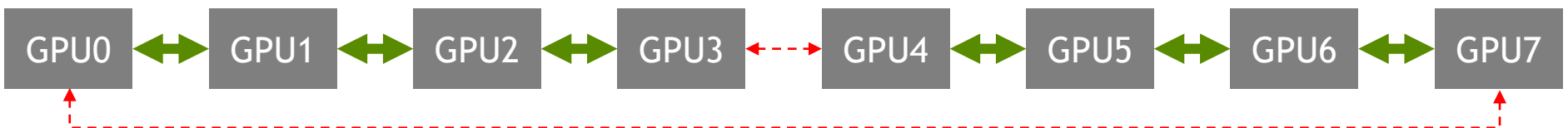


NVLINKは使われているのか?

NVLINK接続形態



ヤコビ反復法の通信

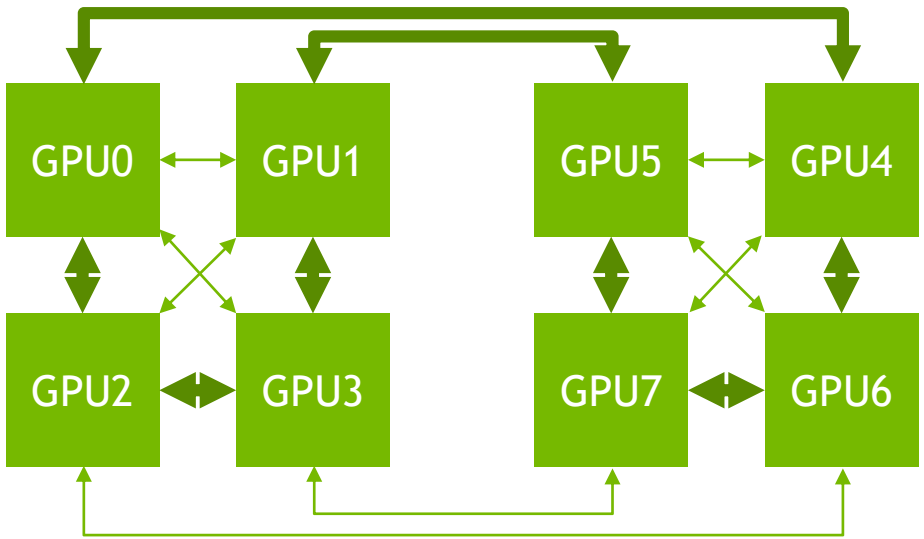


GPU3とGPU4、GPU0とGPU7、ここにはPCIが使われる

GPU AFFINITY

```
export CUDA_VISIBLE_DEVICES="0,2,3,1,5,7,6,4"
```

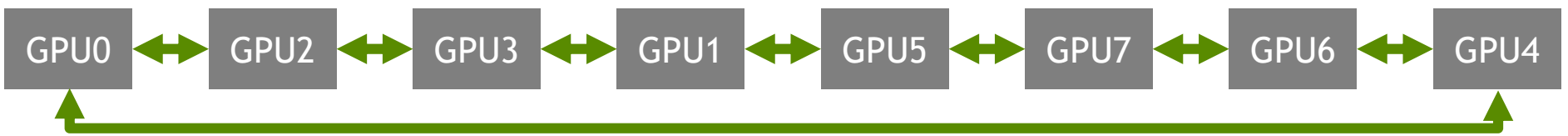
NVLINK接続形態



CUDA_VISIBLE_DEVICES

- そのプロセスが認識できるGPU、その順番を設定可能

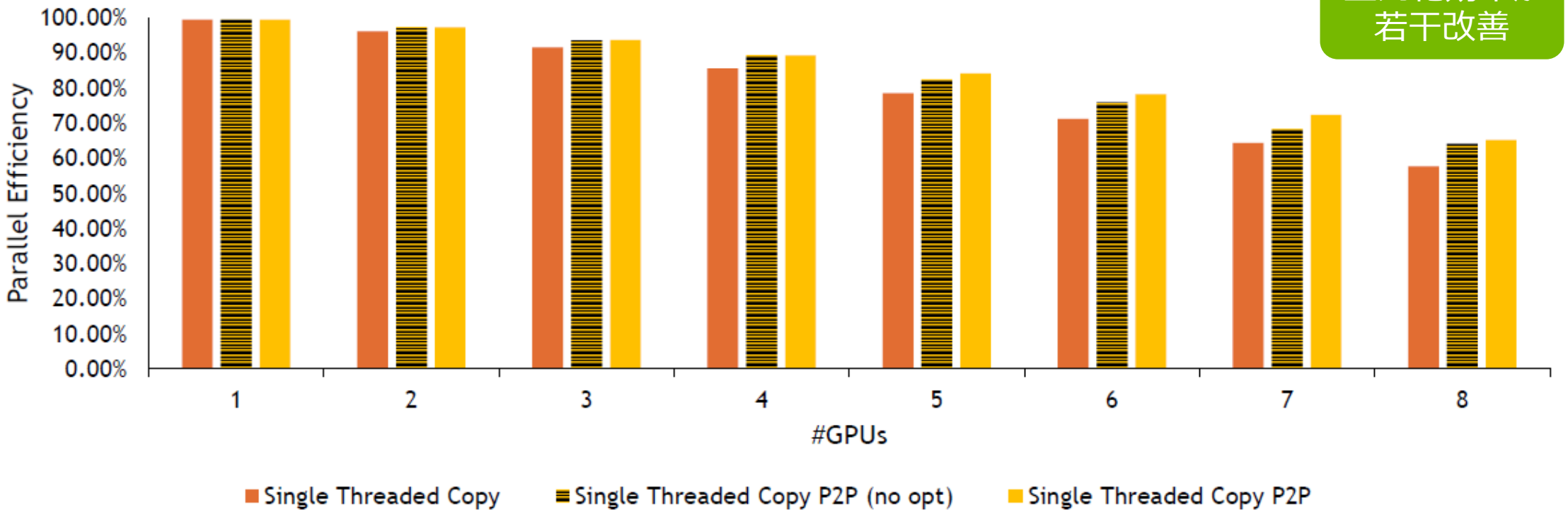
ヤコビ反復法の通信



マルチGPU (シングル・スレッド)

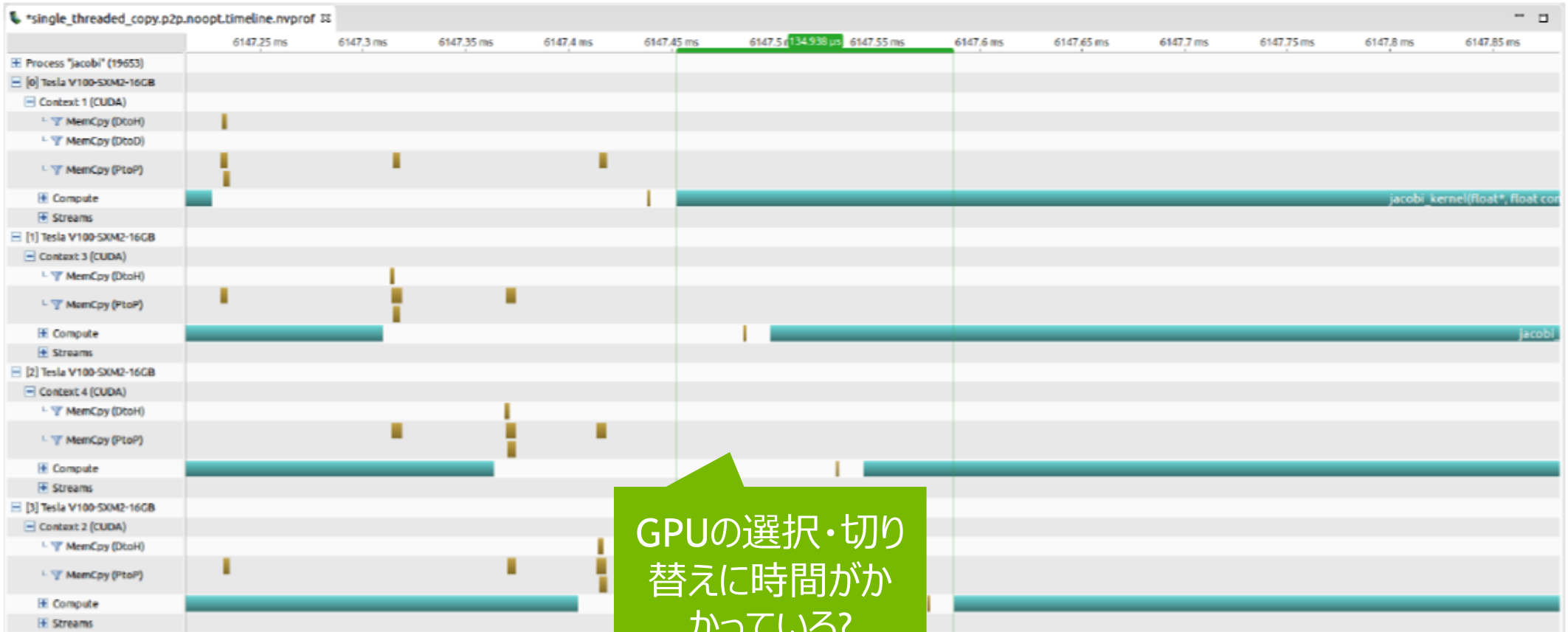
DGX-1V, 1000反復

並列化効率が
若干改善



マルチGPU (シングル・スレッド + P2Pコピー)

NVVP Timeline



マルチ・スレッドによるマルチGPU使用

マルチ・スレッドによる、マルチGPU活用

OpenMP

GPU数の取得

スレッド数の指定

スレッド番号の取得

マルチスレッド区間

```
num_devs = 0;
cudaGetDeviceCount( &num_devs );

#pragma omp parallel num_threads( num_devs )
{
        dev_id = omp_get_thread_num();
        cudaSetDevice( dev_id );

        ...

}
```

シングル・スレッド

```
while ( error > tol ) {
    for ( dev_id = 0; dev_id < num_devs; dev_id++ ) {
        cudaSetDevice( dev_id );
        cudaMemsetAsync( error_d[dev_id], 0, … );
        kern_jacobi<<<blocks, threads, …>>>( a_new[dev_id], a[dev_id], er
                                             nx, iy_start[dev_id], iy_
        cudaMemcpyAsync( error_h[dev_id], error_d[dev_id], … );

        prev_id = (dev_id – 1) % num_devs;
        cudaMemcpyAsync( a_new[prev_id] + iy_end[prev_id]*nx,
                         a_new[dev_id] + iy_start[dev_id]*nx, … );
        next_id = (dev_id + 1) % num_devs;
        cudaMemcpyAsync( a_new[next_id],
                         a_new[dev_id] + (iy_end[dev_id]-1)*nx, … );
    }

    error = 0.0;
    for ( dev_id = 0; dev_id < num_devs; dev_id++ ) {
        cudaSetDevice( dev_id );  cudaDeviceSynchronize();
        error = max(error, *(error_h[dev_id]));
        swap( a_new[dev_id], a[dev_id] );
    }
}
```

複数GPU、それぞれに指示
cudaSetDeviceでGPU指定

GPUカーネル実行
非同期API使用、
ここで待たないようにする

通信

各GPUの処理完了を待つ
cudaDeviceSynchronize

最大誤差をCPUに回収

マルチ・スレッド (OPENMP)

```
while ( error > tol ) {
    error = 0.0
#pragma omp parallel num_threads(num_devs) reduction(max:error)
    {
        dev_id = omp_get_thread_num();
        cudaSetDevice( dev_id );
        cudaMemsetAsync( error_d[dev_id], 0, … );
        kern_jacobi<<<blocks, threads, …>>>( a_new[dev_id], a[dev_id], error_d[dev_id],
                                             nx, iy_start[dev_id], iy_end[dev_id] );
        cudaMemcpyAsync( error_h[dev_id], error_d[dev_id], … );

        prev_id = (dev_id - 1) % num_devs;
        cudaMemcpyPeerAsync( a_new[prev_id] + iy_end[prev_id]*nx,
                             a_new[dev_id] + iy_start[dev_id]*nx, … );
        next_id = (dev_id + 1) % num_devs;
        cudaMemcpyPeerAsync( a_new[next_id],
                             a_new[dev_id] + (iy_end[dev_id]-1)*nx, … );

        cudaDeviceSynchronize();
        error = max(error, *(error_h[dev_id]));
        swap( a_new[dev_id], a[dev_id] );
    }
}
```

スレッド間の
最大誤差の縮約

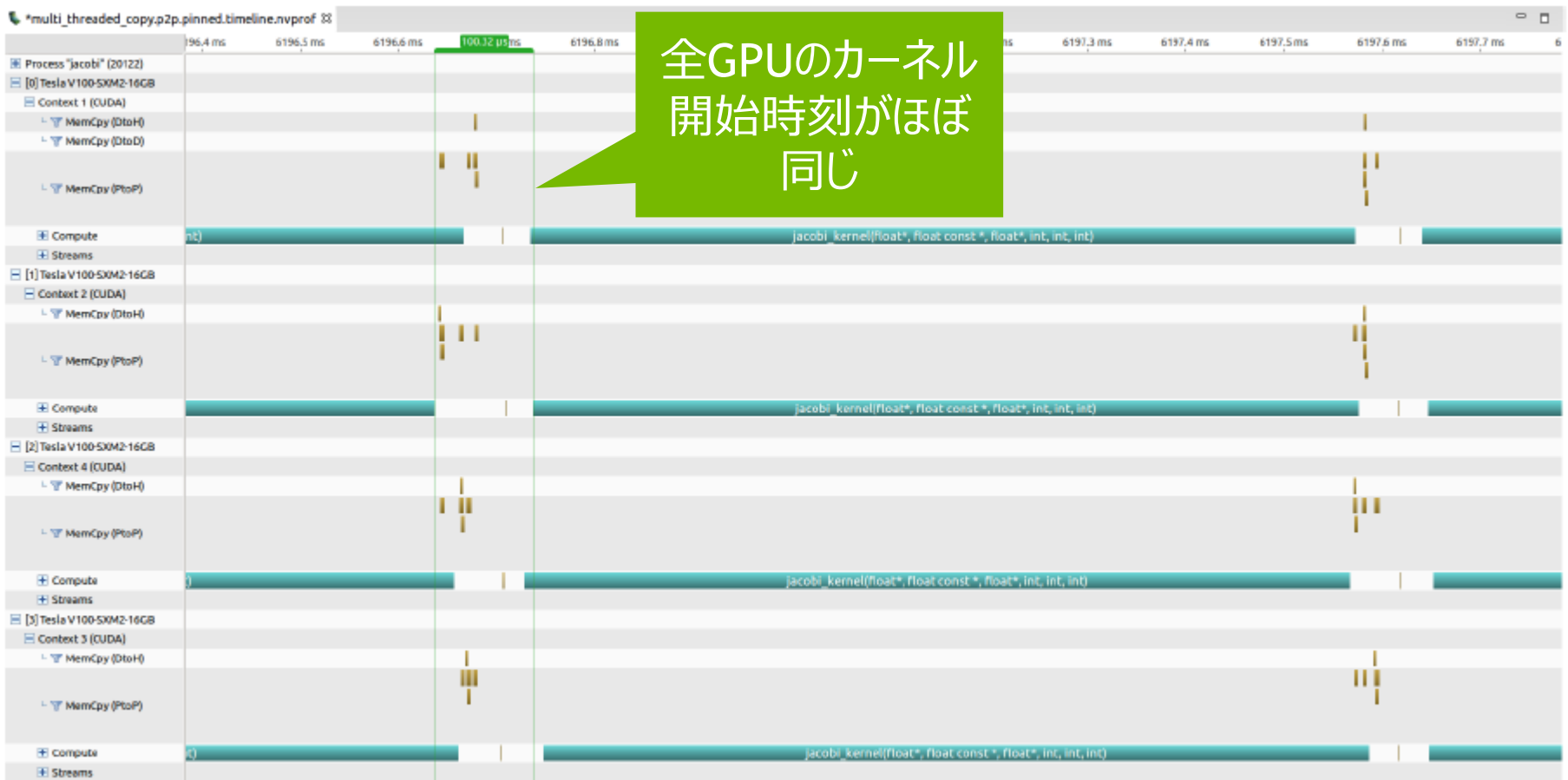
GPUカーネル実行
非同期API使用、
ここで待たないようにする

通信

最大誤差をCPUに回収

マルチ・スレッド + P2Pコピー

NVVP Timeline

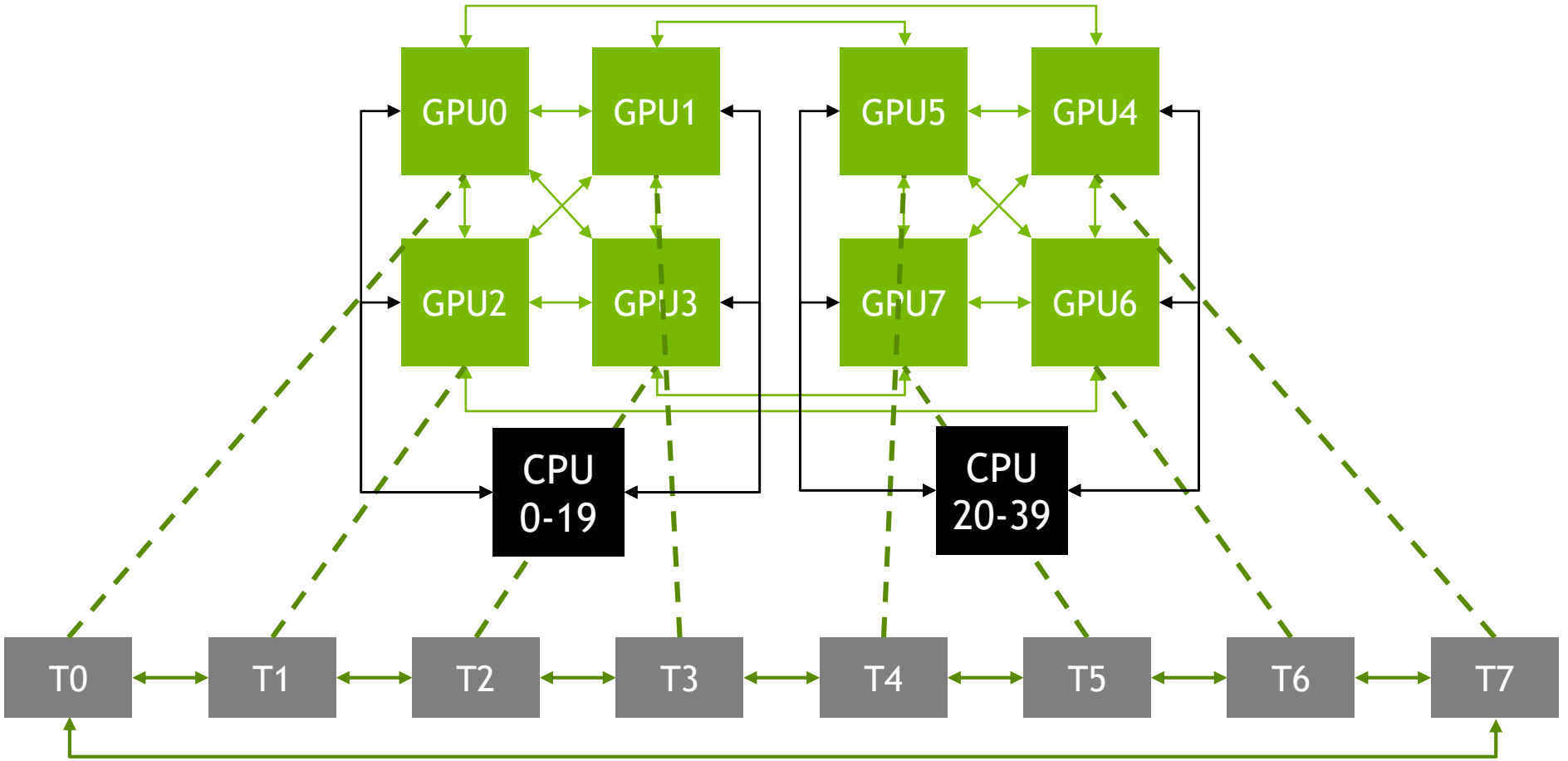


マルチ・スレッド + P2Pコピー

DGX-1V, 1000反復

並列化効率がかなり改善

スレッドは、どのCPU上にいるの?



GPU/CPU AFFINITY

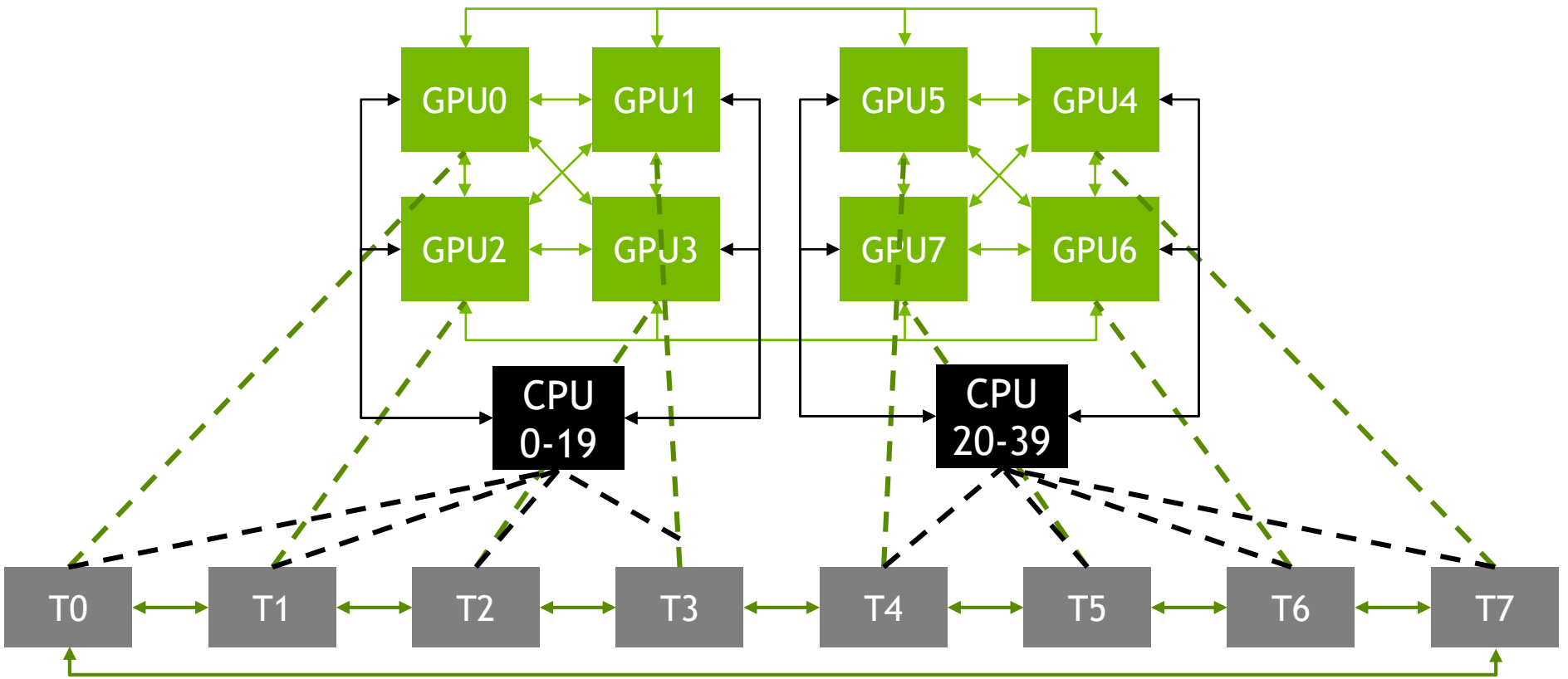
nvidia-smi topo -m

```
$ nvidia-smi topo -m
```

	GPU0	GPU1	GPU2	GPU3	GPU4	GPU5	GPU6	GPU7	mlx5_0	mlx5_2	mlx5_1	mlx5_3	CPU Affinity	
GPU0	X	NV1	NV1	NV2	NV2	SOC	SOC	SOC	PIX	SOC	PHB	SOC	**0-19**	CPU 0
GPU1	NV1	X	NV2	NV1	SOC	NV2	SOC	SOC	PIX	SOC	PHB	SOC	**0-19**	
GPU2	NV1	NV2	X	NV2	SOC	SOC	NV1	SOC	PHB	SOC	PIX	SOC	**0-19**	
GPU3	NV2	NV1	NV2	X	SOC	SOC	SOC	NV1	PHB	SOC	PIX	SOC	**0-19**	
GPU4	NV2	SOC	SOC	SOC	X	NV1	NV1	NV2	SOC	PIX	SOC	PHB	**20-39**	CPU 1
GPU5	SOC	NV2	SOC	SOC	NV1	X	NV2	NV1	SOC	PIX	SOC	PHB	**20-39**	
GPU6	SOC	SOC	NV1	SOC	NV1	NV2	X	NV2	SOC	PHB	SOC	PIX	**20-39**	
GPU7	SOC	SOC	SOC	NV1	NV2	NV1	NV2	X	SOC	PHB	SOC	PIX	**20-39**	
mlx5_0	PIX	PIX	PHB	PHB	SOC	SOC	SOC	SOC	X	SOC	PHB	SOC		
mlx5_2	SOC	SOC	SOC	SOC	PIX	PIX	PHB	PHB	SOC	X	SOC	PHB		
mlx5_1	PHB	PHB	PIX	PIX	SOC	SOC	SOC	SOC	PHB	SOC	X	SOC		
mlx5_3	SOC	SOC	SOC	SOC	PHB	PHB	PIX	PIX	SOC	PHB	SOC	X		

GPU/CPU AFFINITY

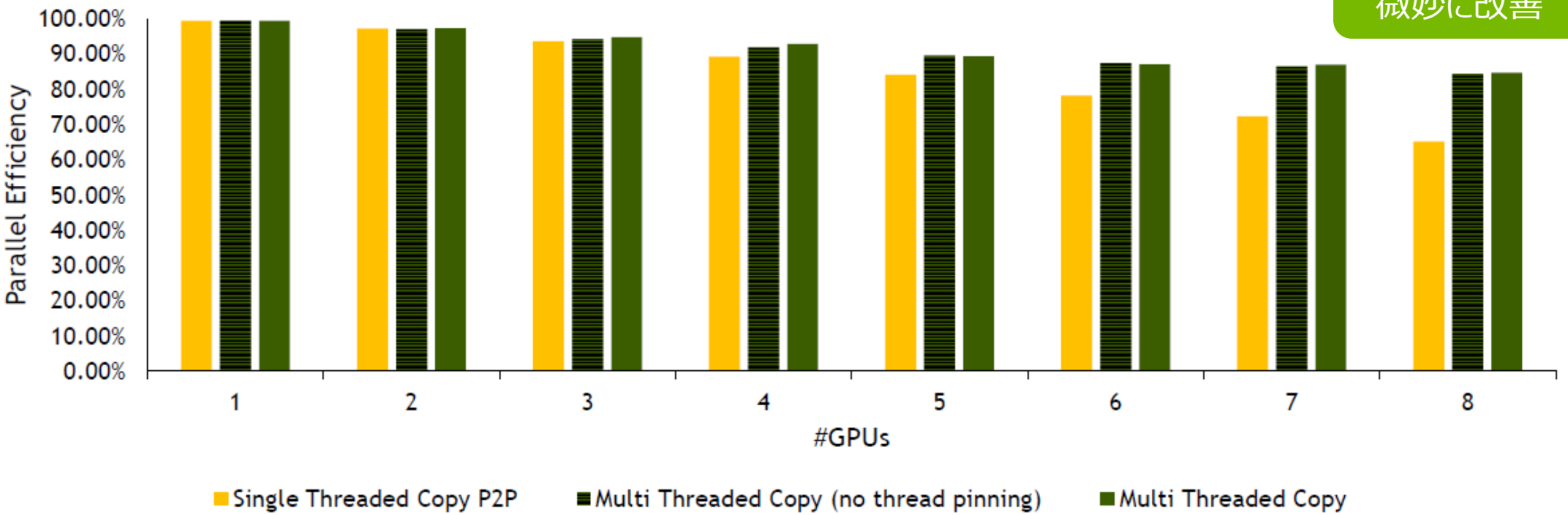
```
export CUDA_VISIBLE_DEVICES="0,2,3,1,5,7,6,4"
export OMP_PROC_BIND=TRUE
export OMP_PLACES="{0},{1},{2},{3},{20},{21},{22},{23}"
```



マルチスレッド + P2Pコピー

DGX-1V, 1000反復

並列化効率は
微妙に改善

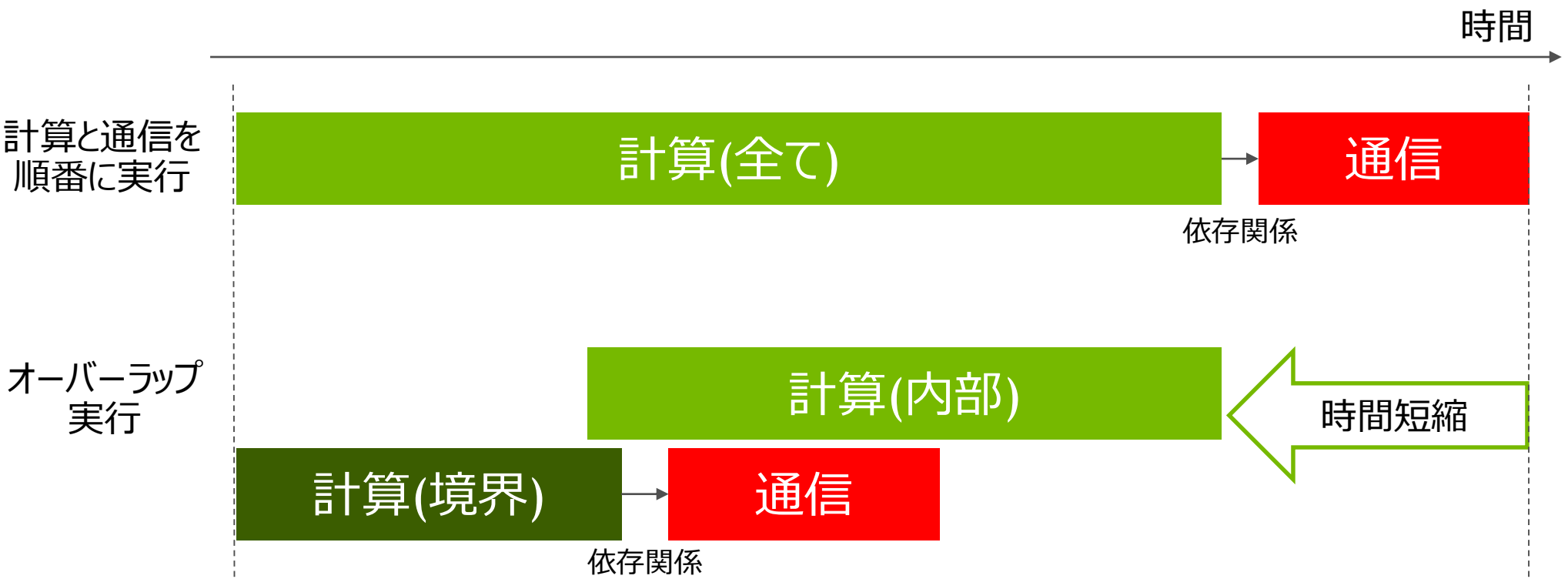


マルチスレッド + P2Pコピー

NVVP Timeline

計算と通信のオーバーラップ

通信が必要な、境界部分の計算を、先にする



計算と通信のオーバーラップ

CUDAストリームで、実行順序を制御

```
// (内部計算)
kern_jacobi<<<blocks, threads, 0, stream_comp>>>(
    a_new[dev_id], a[dev_id], error_d[dev_id], nx, iy_start[dev_id]+1, iy_end[dev_id]-1 );

// (境界計算)
kern_jacobi<<<blocks, threads, 0, stream_prev>>>(
    a_new[dev_id], a[dev_id], error_d[dev_id], nx, iy_start[dev_id], iy_start[dev_id]+1 );
kern_jacobi<<<blocks, threads, 0, stream_next>>>(
    a_new[dev_id], a[dev_id], error_d[dev_id], nx, iy_end[dev_id]-1, iy_end[dev_id] );

// (通信)
cudaMemcpyPeerAsync( a_new[prev_id] + iy_end[prev_id]*nx,
                     a_new[dev_id] + iy_start[dev_id]*nx, …, stream_prev );
cudaMemcpyPeerAsync( a_new[next_id],
                     a_new[dev_id] + (iy_end[dev_id]-1)*nx, …, stream_next );
```

内部と境界部を、別カーネルで計算

CUDAストリームで依存性を記述

計算と通信のオーバーラップ

Priority CUDAストリーム

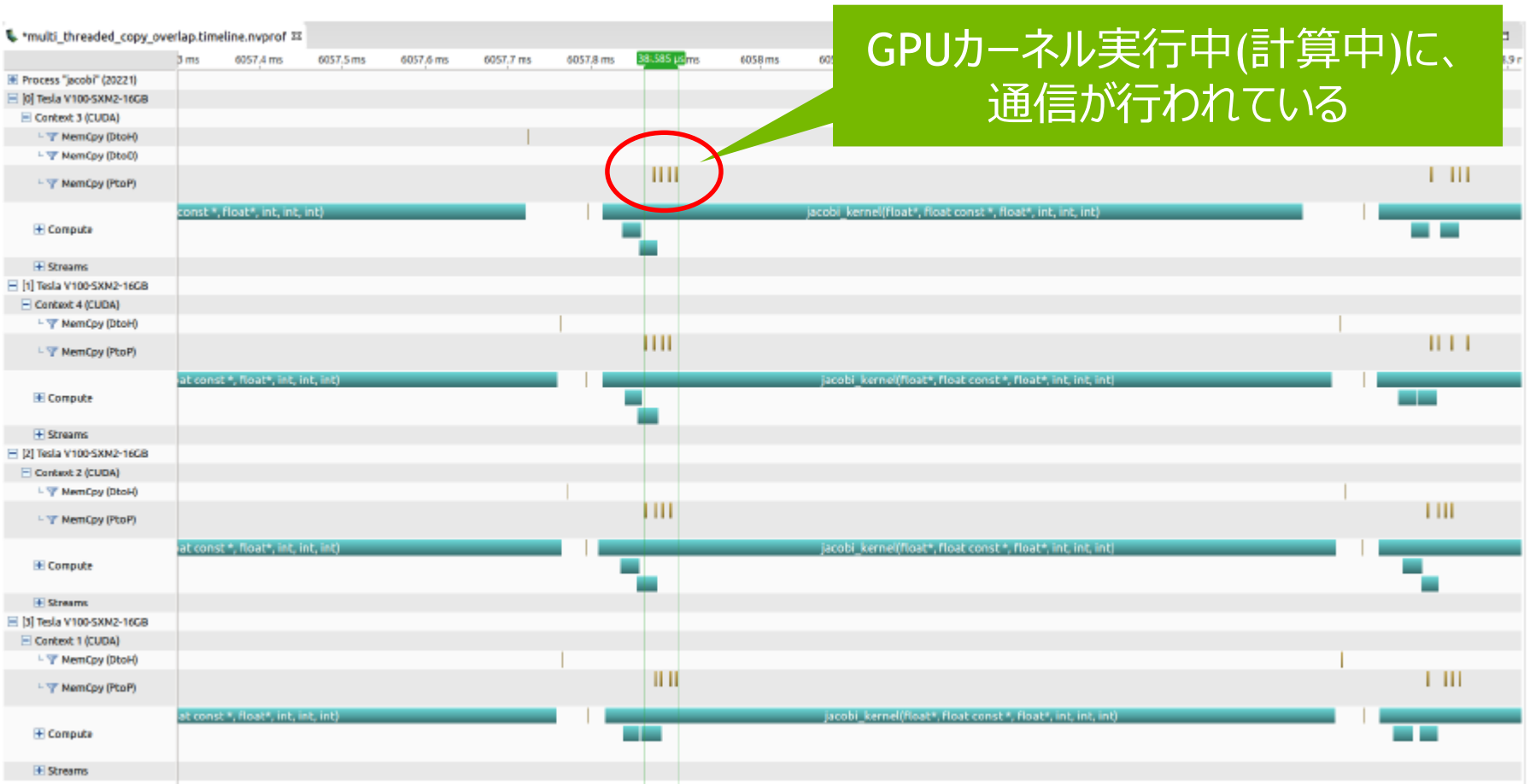
```
cudaDeviceGetStreamPriorityRange( &low_priority, &high_priority );

cudaStreamCreateWithPriority( &stream_comp, ..., low_priority );

cudaStreamCreateWithPriority( &stream_prev, ..., high_priority );
cudaStreamCreateWithPriority( &stream_next, ..., high_priority );
```

Priorityの高いCUDAストリームに投入されたカーネルは、優先的に実行される

→ 境界領域の計算に、Priorityの高いCUDAストリームを使用

マルチ・スレッド + 計算・通信オーバーラップ

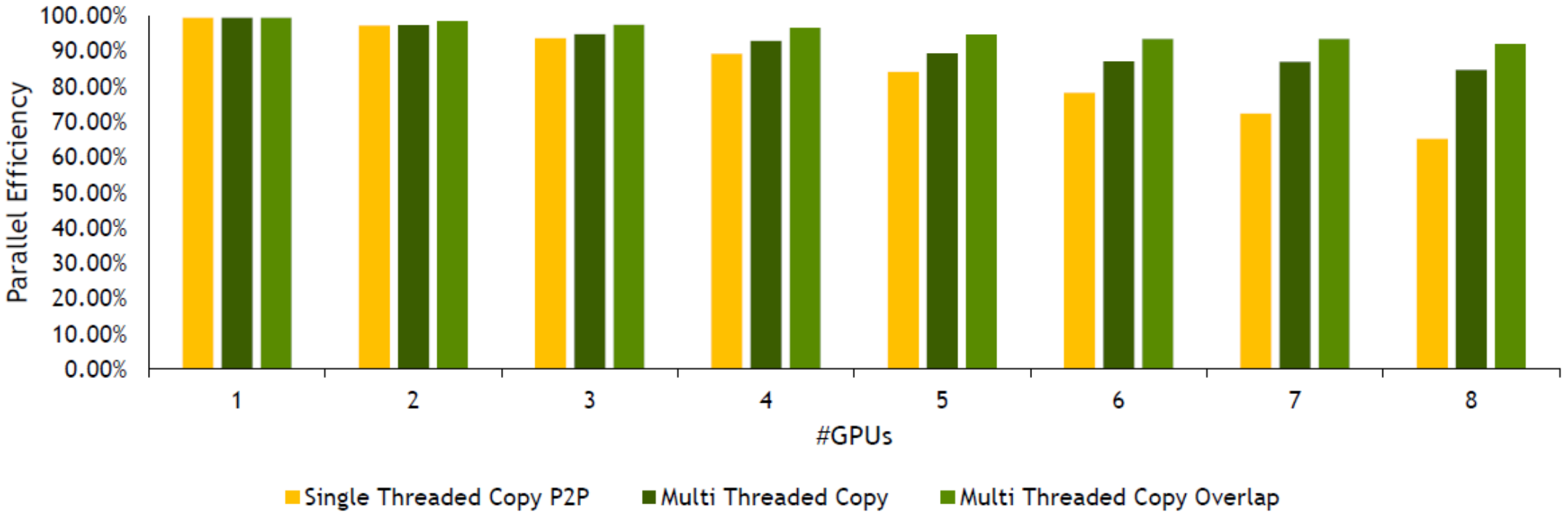
NVVP Timeline



マルチ・スレッド + 計算・通信オーバーラップ

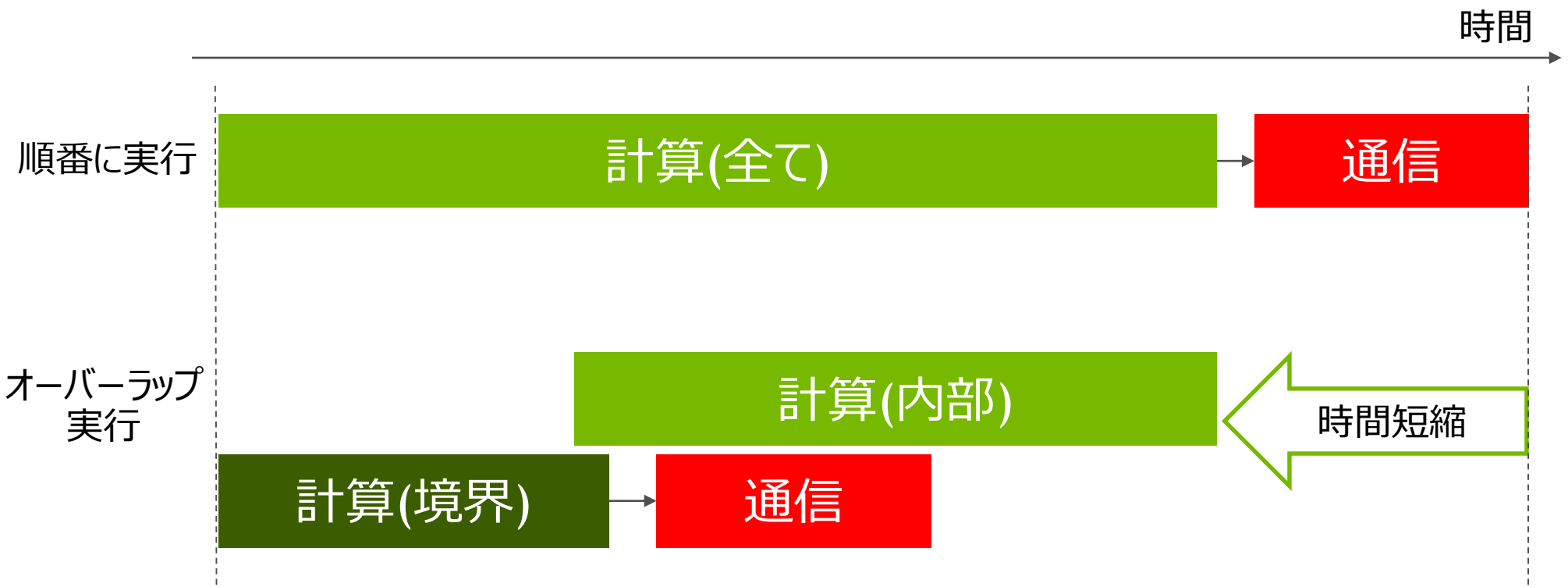
DGX-1V, 1000反復

並列化効率は
かなり改善



明示的な通信は必要なのか

通信(memcpy)はCPUが起動



P2Pアクセス

GPUカーネルから、隣接GPUメモリに直接アクセス

GPUカーネル

```
__global__ void kern_jacobi( ... ) {
    int ix = ...;
    int iy = ...;
    if ( out of range ) return;
    new_val = 0.25 * ( a[ ix + (iy-1)*nx ] + a[ ix + (iy+1)*nx ]
                     + a[ (ix-1) + iy*nx ] + a[ (ix+1) + iy*nx ] );
    a_new[ ix + iy*nx ] = new_val




    my_error = fabs( new_val - a[ ix + iy*nx ] );
    atomicMax( error, my_error );
}
```

各スレッドの担当箇所を決定
領域内か確認

担当要素の計算

計算結果をwrite

収束条件の計算(最大差分)

GPUカーネル
P2Pアクセス

```
__global__ void kern_jacobi_p2p( ... ) {
    int ix = ...;
    int iy = ...;
    if ( out of range ) return;
    new_val = 0.25 * ( a[ ix + (iy-1)*nx ] + a[ ix + (iy+1)*nx ]
                     + a[ (ix-1) + iy*nx ] + a[ (ix+1) + iy*nx ] );
    a_new[ ix + iy*nx ] = new_val

    if ( iy == iy_start ) a_new_prev[ ix + iy_end*nx ] = new_val;
    if ( iy == iy_end-1 ) a_new_next[ ix             ] = new_val;

    my_error = fabs( new_val - a[ ix + iy*nx ] );
    atomicMax( error, my_error );
}
```

各スレッドの担当箇所を決定
領域内か確認

担当要素の計算

計算結果をwrite

隣接GPUメモリに
直接書き込み

収束条件の計算(最大差分)

マルチス・レッド + P2Pアクセス

```
while ( error > tol ) {
    cudaMemsetAsync( error_d[dev_id], 0, …, stream_comp );
    kern_jacobi_p2p<<<blocks, threads, …, stream_comp>>>(
        a_new[dev_id], a[dev_id], error_d[dev_id],
        nx, iy_start[dev_id], iy_end[dev_id],
        a_new[prev_id], iy_end[prev_id],
        a_new[next_id], iy_start[next_id] );
    cudaMemcpyAsync( error_h[dev_id], error_d[dev_id], …, stream_comp );

    cudaDeviceSynchronize();
#pragma omp barrier
    (... reduce error over threads...)
}
```

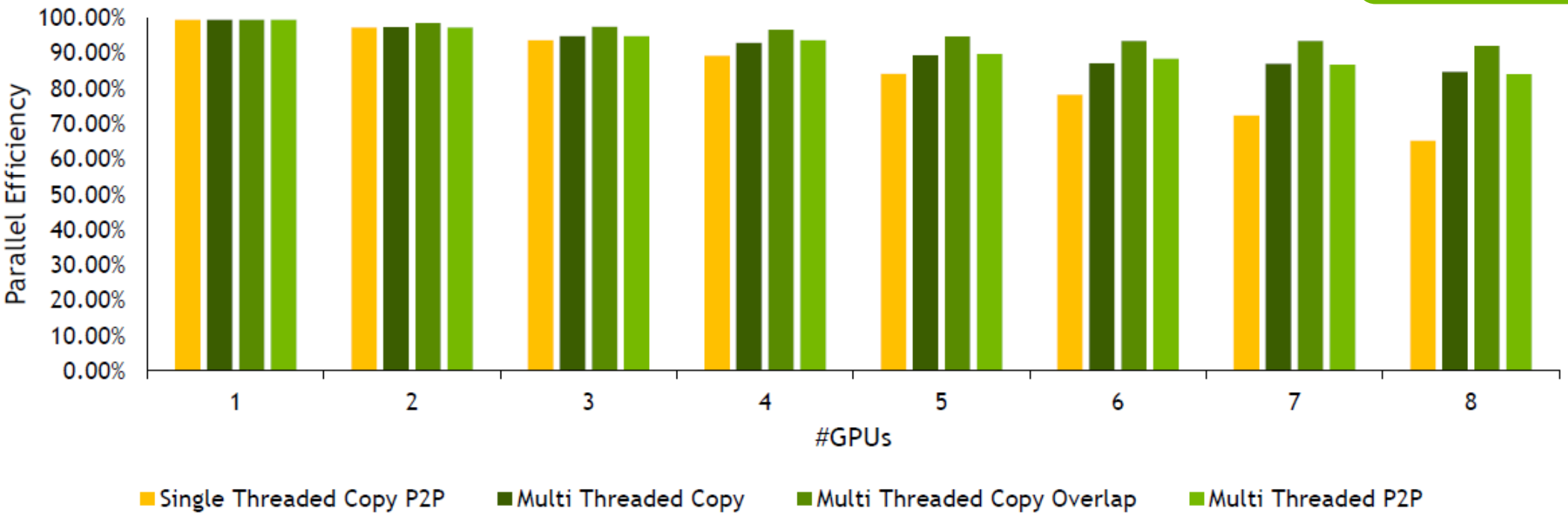
P2Pアクセス用のカーネルを投入

明示的な通信は不要

マルチ・スレッド + P2Pアクセス

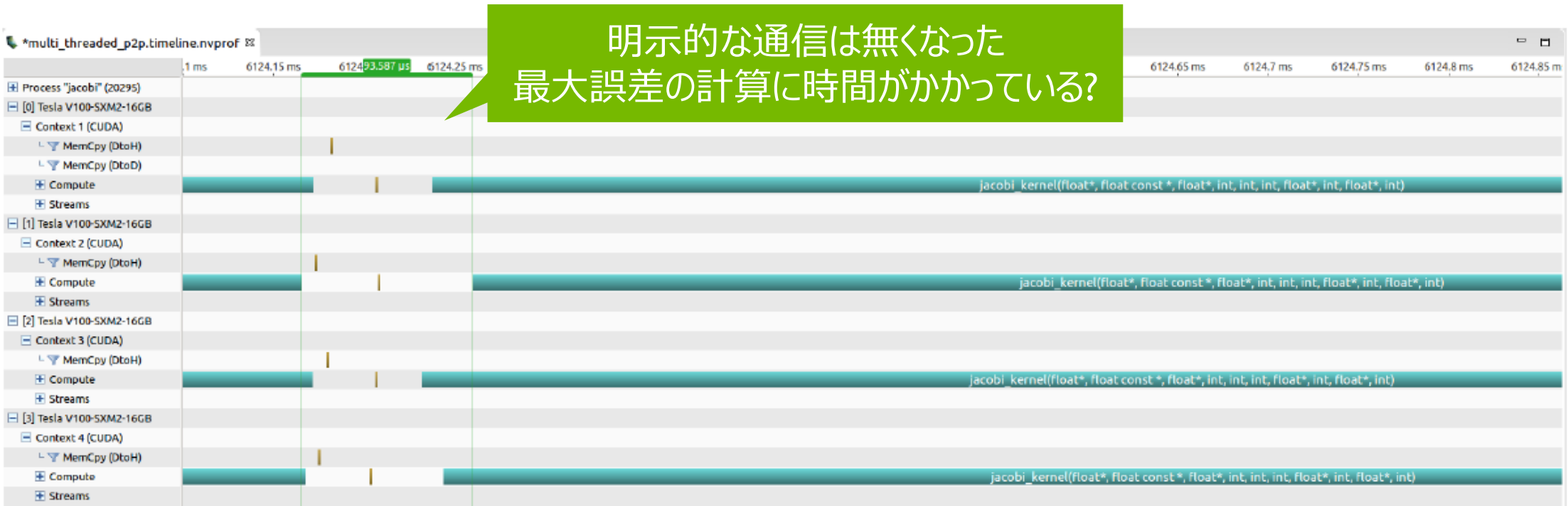
DGX-1V, 1000反復

並列化効率は低下?



マルチ・スレッド + P2Pアクセス

NVVP Timeline



マルチス・レッド + P2Pアクセス

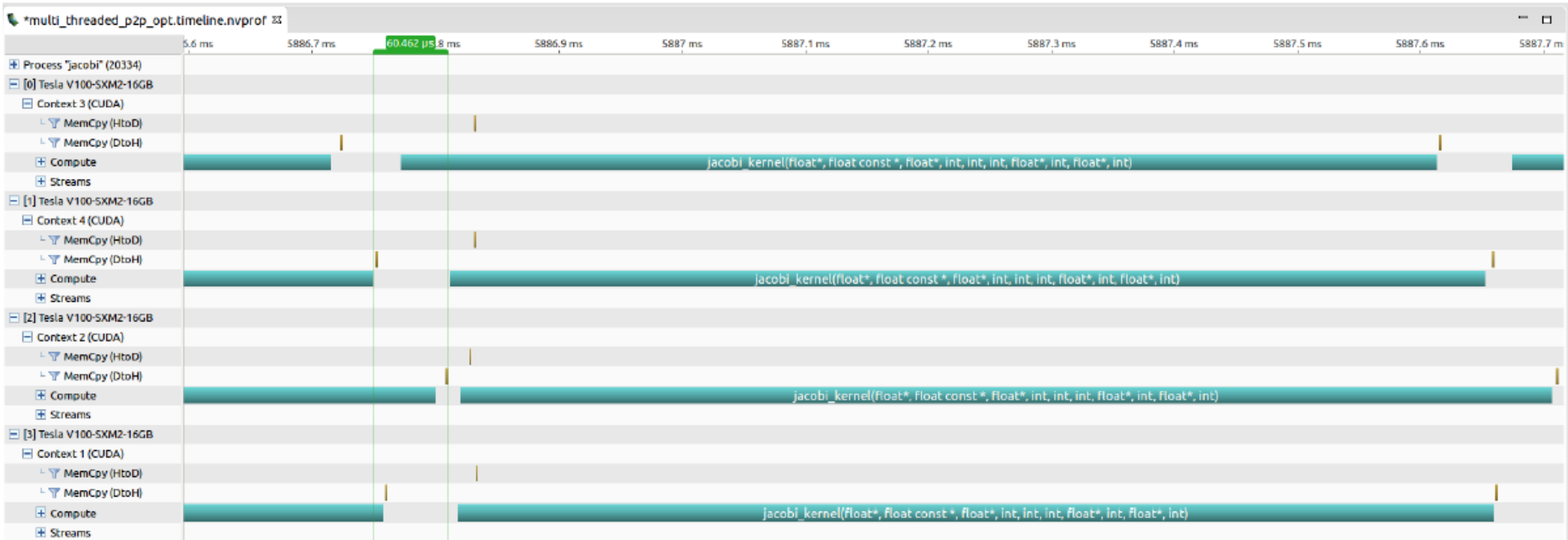
収束判定を1サイクル遅延

```
while ( error > tol ) {
    cudaMemsetAsync( error_d[dev_id], 0, …, stream_comp );
    kern_jacobi_p2p<<<blocks, threads, …, stream_comp>>>(
        a_new[dev_id], a[dev_id], error_d[dev_id],
        nx, iy_start[dev_id], iy_end[dev_id],
        a_new[prev_id], iy_end[prev_id],
        a_new[next_id], iy_start[next_id] );
    cudaMemcpyAsync( error_h[dev_id], error_d[dev_id], …, stream_comp );

#pragma omp barrier
    (... reduce error over threads...)
}
```

GPUカーネル実行中に
前サイクルの誤差を計算

マルチ・スレッド + P2Pアクセス，遅延判定

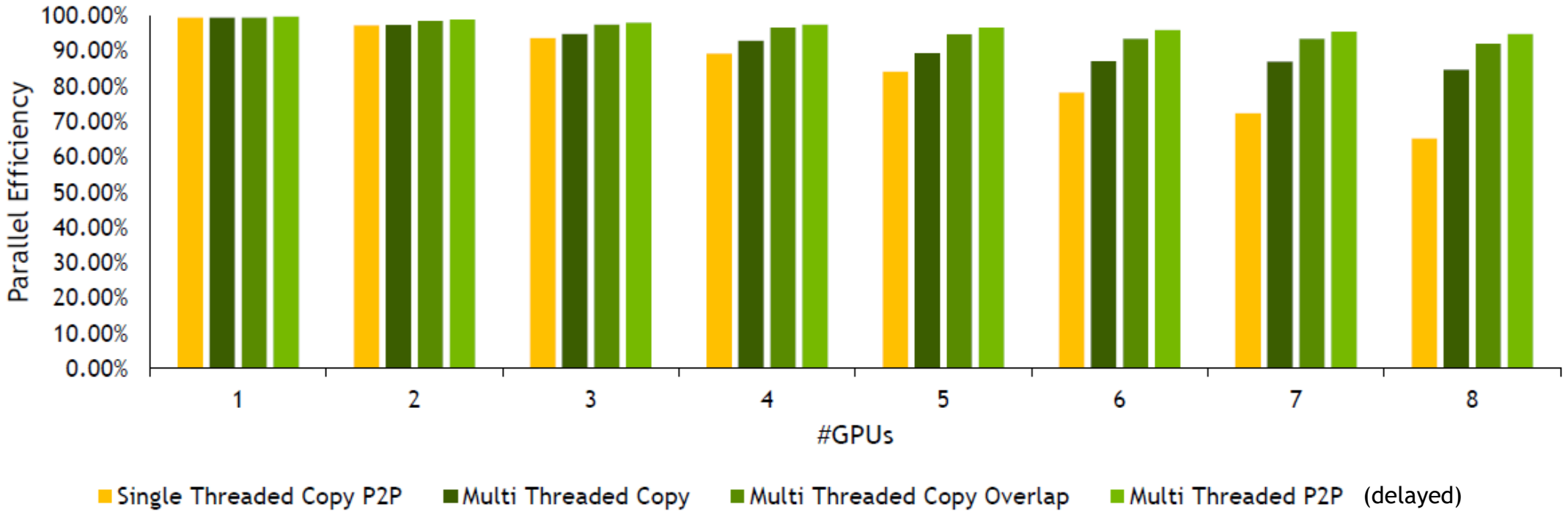
NVVP Timeline



マルチ・スレッド + P2Pアクセス & 遅延判定

DGX-1V, 1000反復

並列化効率は改善



マルチ・スレッドでマルチGPU

DGX1V: 8 GPUまでヤコビ反復法がスケールすることを確認

問題: マルチノードの用途に拡張できない

マルチ・プロセスで、マルチGPUを活用する

MPI (MESSAGE PASSING INTERFACE)

- プロセス並列で、最も一般的な方法 (HPCでは)
  - プロセス間のデータ交換(通信)のAPIを規定
  - 1対1: MPI_Send, MPI_Recv, …
  - 集団: MPI_Bcast, MPI_Gather, MPI_Reduce, …
- 複数の実装、複数の言語サポート
  - MPICH, MVAPICH, OpenMPI, …
  - C/C++, Fortran, Python, …

MPIプログラム

コンパイル
```
$ mpicc –o myapp myapp.c
```

実行
```
$ mpirun –np 4 myapp
```

```
#include <mpi.h>

    MPI_Init( ... );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    ...
    (MPIでプロセス間のデータ交換)
    ...
    MPI_Finalize( );
```

MPIライブラリ開始

プロセス数

プロセスID (0 … size-1)

MPIライブラリ終了

MPIプログラム

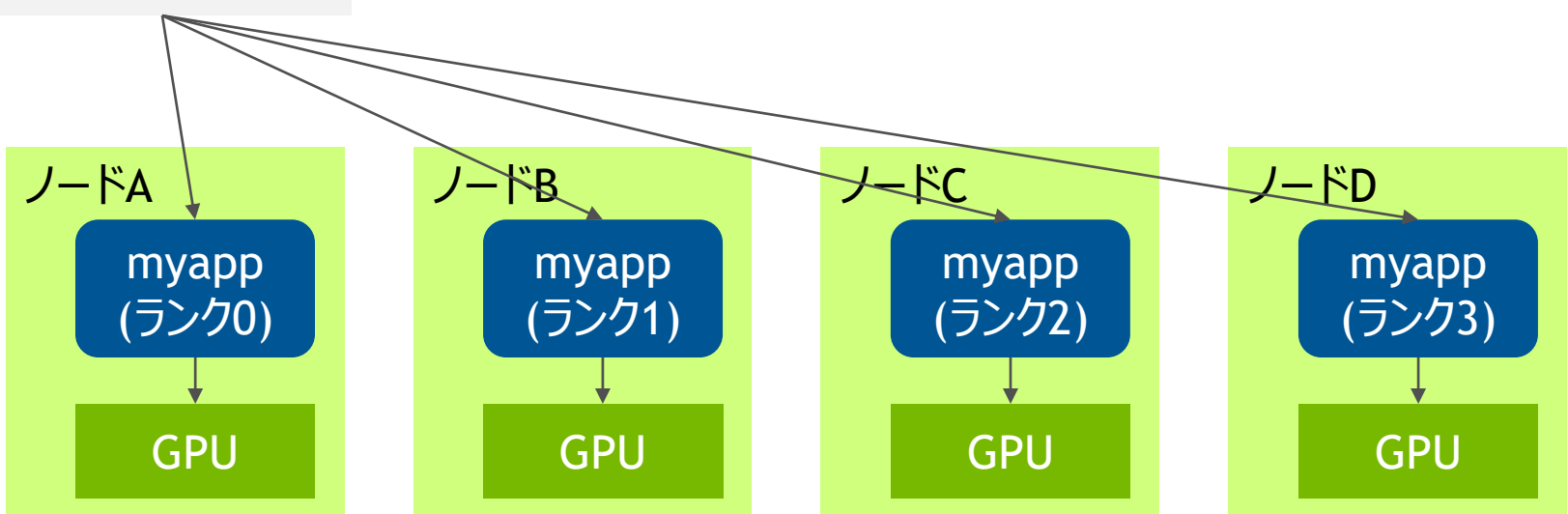
コンパイル

```
$ mpicc –o myapp myapp.c
```

実行

```
$ mpirun –np 4 myapp
```

マルチ・ノード



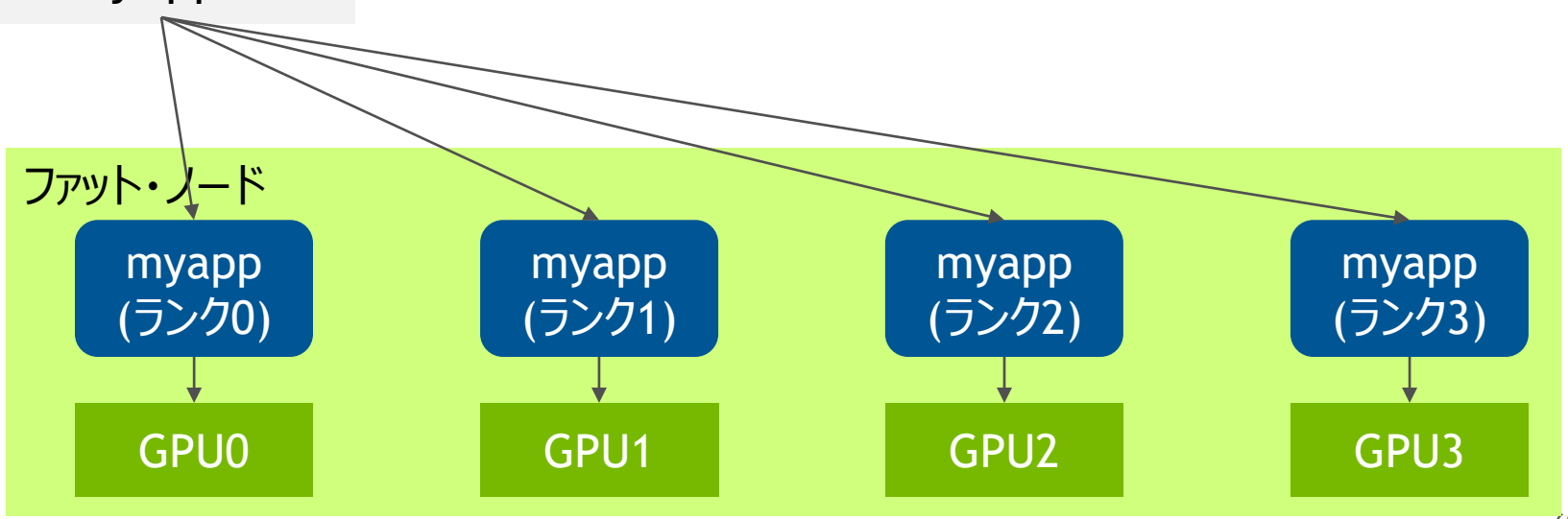
MPIプログラム

コンパイル

```
$ mpicc -o myapp myapp.c
```

実行

```
$ mpirun -np 4 myapp
```



マルチ・プロセス (MPI)

```
cudaSetDevice( rank );
prev_rank = (rank - 1) % size;
next_rank = (rank + 1) % size;

while ( error > tol ) {
    cudaMemsetAsync( error_d, 0, … );
    kern_jacobi<<<blocks, threads, …>>>( a_new, a, error_d, nx, iy_start, iy_end );
    cudaMemcpyAsync( error_h, error_d, … );
    cudaDeviceSynchronize( );

    MPI_Sendrecv( a_new +  iy_start *nx, nx, MPI_FLOAT, prev_rank, ...,
                  a_new +  iy_end   *nx, nx, MPI_FLOAT, next_rank, ...,
                  MPI_COMM_WORLD, ... );
    MPI_Sendrecv( a_new + (iy_end-1)*nx, nx, MPI_FLOAT, next_rank, ...,
                  a_new                , nx, MPI_FLOAT, prev_rank, ...,
                  MPI_COMM_WORLD, ... );

    error = 0.0
    MPI_Allreduce( error_h, &error, 1, MPI_FLOAT, MPI_MAX, MPI_COMM_WORLD );
    swap( a_new, a );
}
```

GPUカーネル実行

MPI通信 (後述)

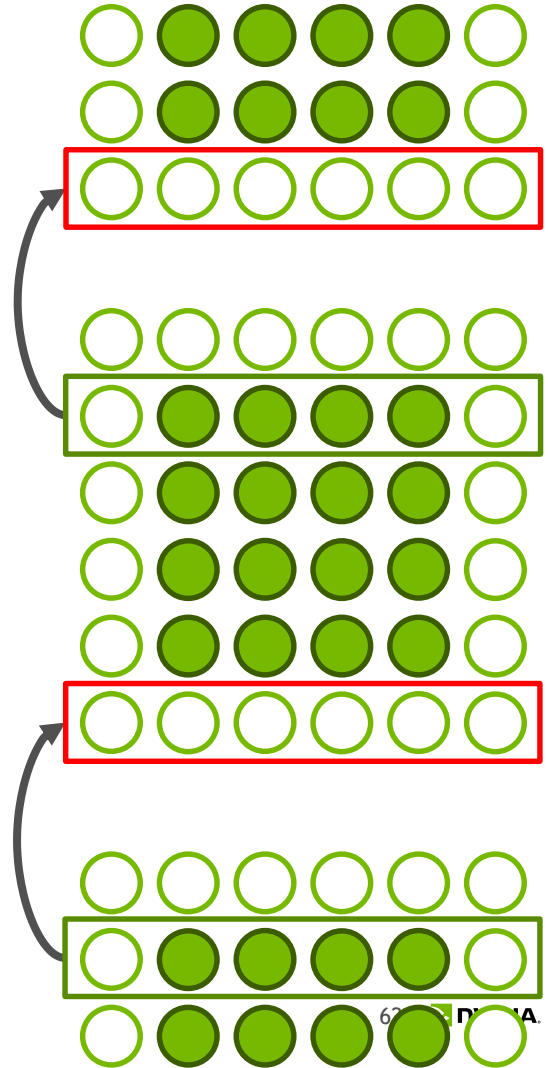
プロセス間の
最大誤差取得

MPI通信: 境界交換

```
MPI_Sendrecv(
  a_new +  iy_start *nx, nx, MPI_FLOAT, prev_rank, ...,
  a_new +  iy_end   *nx, nx, MPI_FLOAT, next_rank, ...,
  MPI_COMM_WORLD, ... );
```

送信

受信



MPI通信: 境界交換

```
MPI_Sendrecv(
    a_new +  iy_start *nx, nx, MPI_FLOAT, prev_rank, ...,
    a_new +  iy_end   *nx, nx, MPI_FLOAT, next_rank, ...,
    MPI_COMM_WORLD, ... );
```

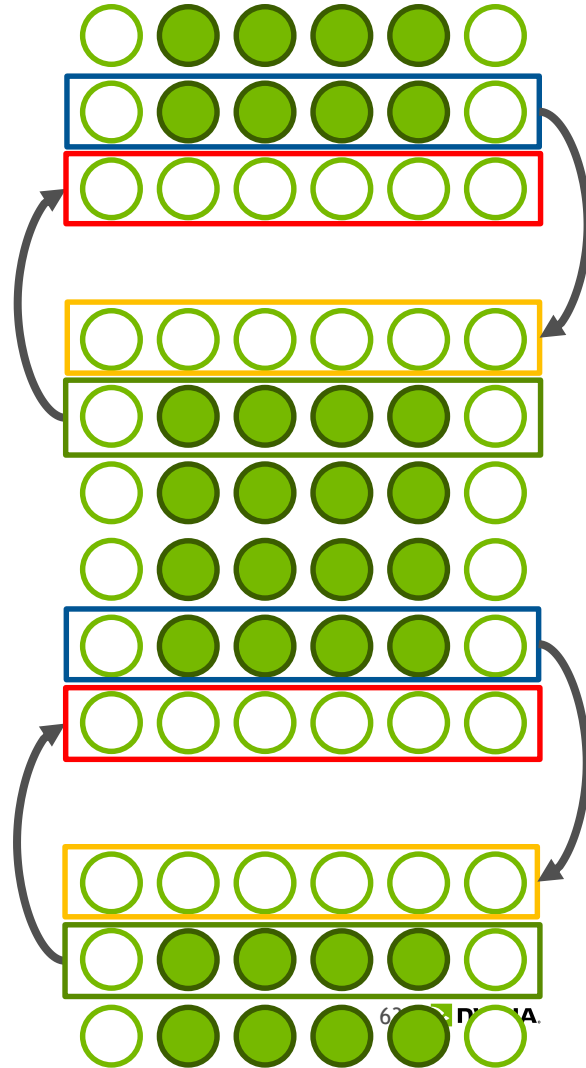
送信

受信

```
MPI_Sendrecv(
    a_new +  (iy_end-1) *nx, nx, MPI_FLOAT, next_rank, ...,
    a_new                  , nx, MPI_FLOAT, prev_rank, ...,
    MPI_COMM_WORLD, ... );
```

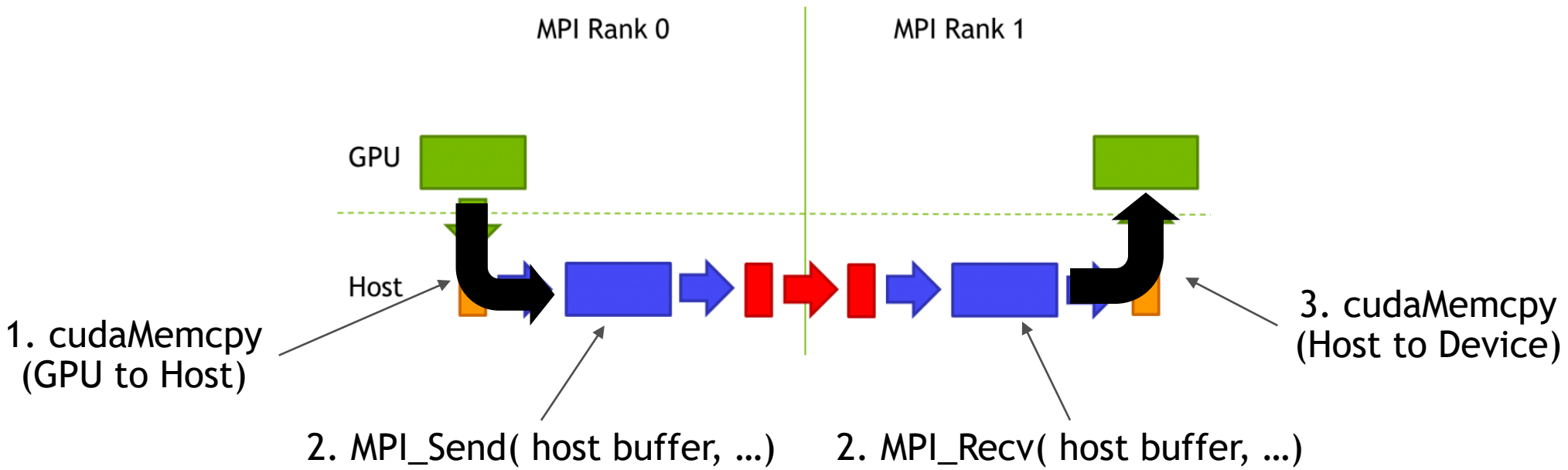
送信

受信



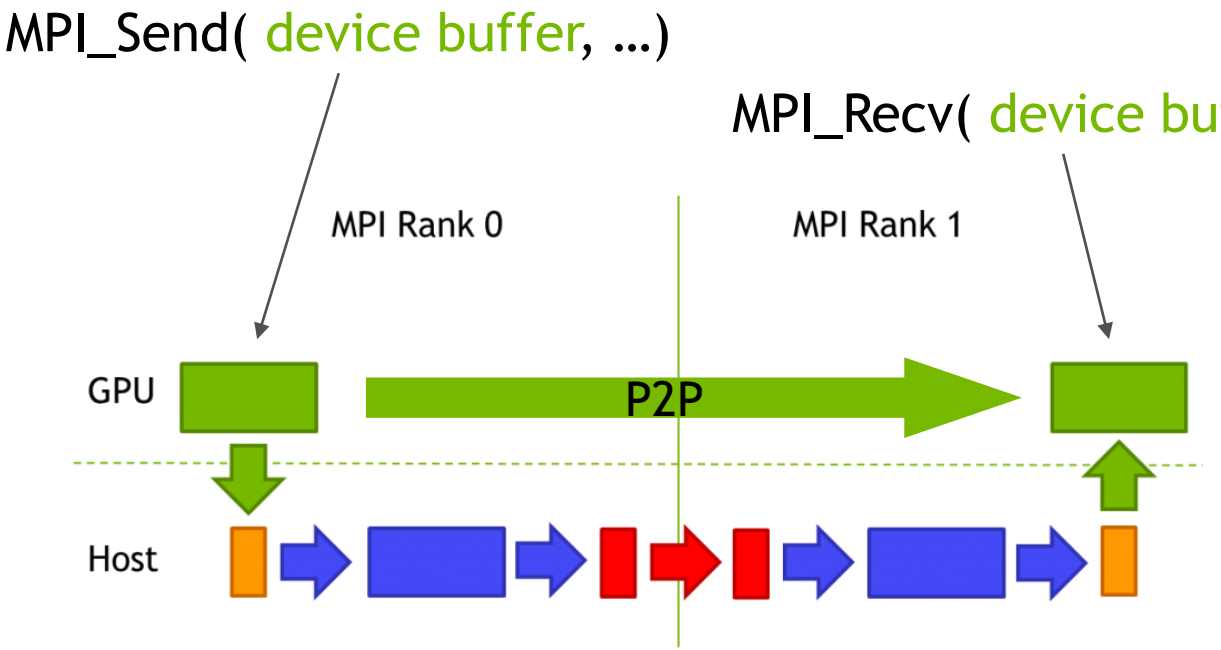
通常のMPI

通常のMPIは、GPUメモリを扱えない
MPI通信の前後で、cudaMemcpyでGPU・Host間のメモリコピー



CUDA AWARE MPI

MPIの送信・受信バッファに、GPUメモリを指定できる

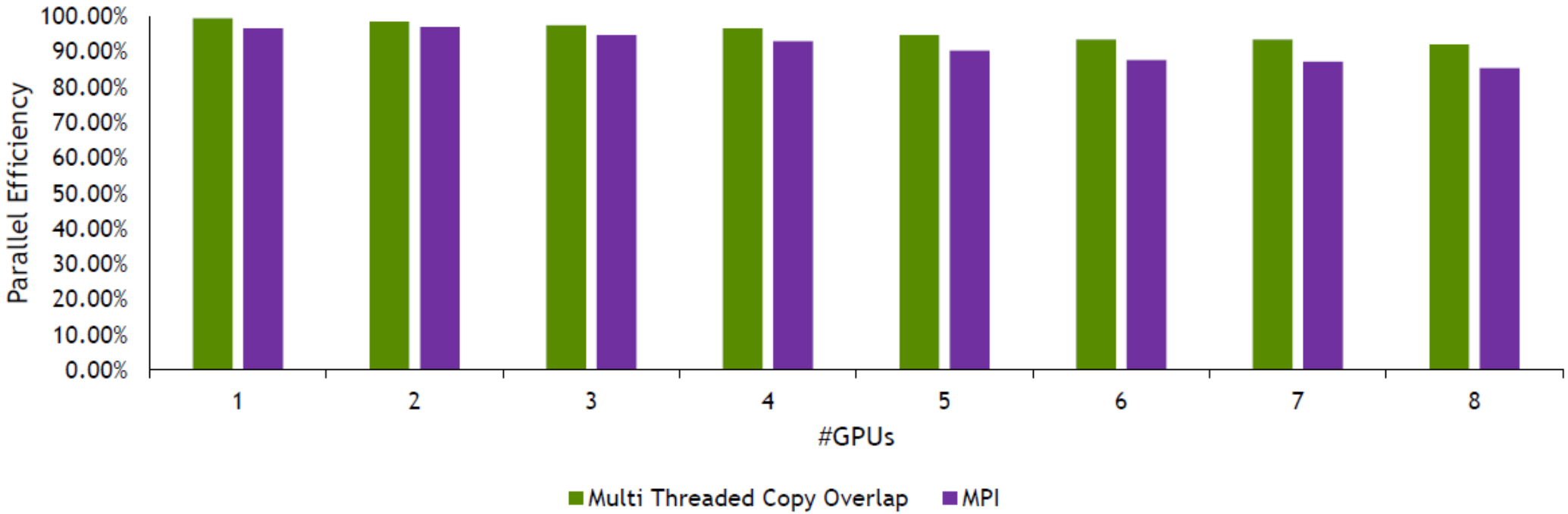


MVAPICHとOpenMPIがサポート

- CPUとGPU間のデータコピーはMPIライブラリが実施
- GPU間でP2Pが使えるなら、直接データ転送

マルチ・プロセス (CUDA AWARE MPI)

DGX-1V, 1000反復



マルチ・プロセス (CUDA AWARE MPI)

NVVP Timeline

計算と通信のオーバーラップ

MPI + CUDAストリーム

```
// (内部計算)
kern_jacobi<<<..., stream_comp>>>( a_new, a, error_d, nx, iy_start+1, iy_end-1 );

// (境界計算)
kern_jacobi<<<..., stream_prev>>>( a_new, a, error_d, nx, iy_start, iy_start+1 );
kern_jacobi<<<..., stream_next>>>( a_new, a, error_d, nx, iy_end-1, iy_end );

// (境界通信)
cudaStreamSynchronize( stream_prev );
MPI_Sendrecv( a_new +  iy_start *nx, nx, MPI_FLOAT, prev_rank, ...,
              a_new +  iy_end   *nx, nx, MPI_FLOAT, next_rank, ...,
              MPI_COMM_WORLD, ... );
cudaStreamSynchronize( stream_next );
MPI_Sendrecv( a_new + (iy_end-1)*nx, nx, MPI_FLOAT, next_rank, ...,
              a_new                , nx, MPI_FLOAT, prev_rank, ...,
              MPI_COMM_WORLD, ... );
```

内部と境界部を、別カーネルで計算

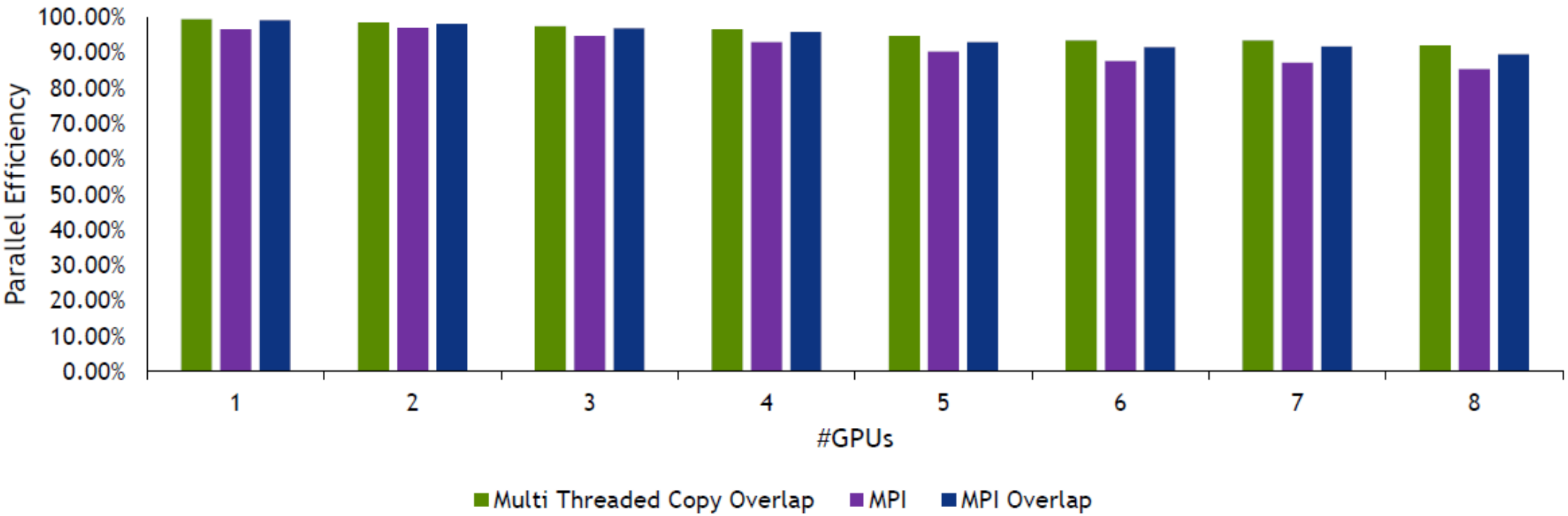
CUDAストリームで依存性を記述

マルチ・プロセス + 計算・通信オーバーラップ

NVVP Timeline

マルチ・プロセス + 計算・通信オーバーラップ

DGX-1V, 1000反復



マルチ・プロセス

計算は、プログラミング言語 (C/C++, Fortran, Python) で記述

通信は、通信ライブラリのAPI (MPI) で記述

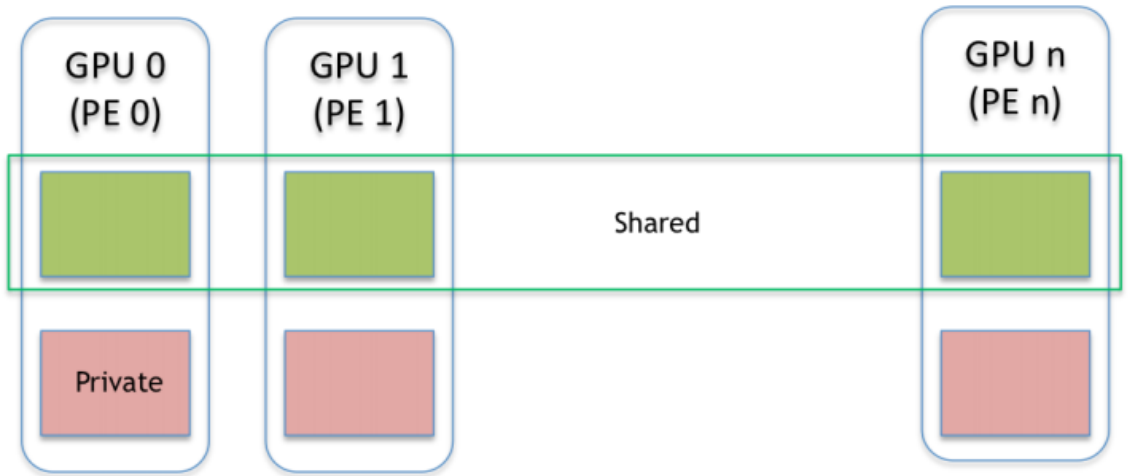
計算と通信は、別々に記述するしかない?

NVSHMEM

Partitioned Global Address Space (PGAS) ライブラリ

OpenSHMEM

- 別プロセスからアクセス可能な空間
- リモートメモリにアクセスするAPI
  - 単方向: shmem_put, shmem_get
  - 集合: shmem_broadcast



NVSHMEMは、OpenSHMEMのGPU実装

- GPUカーネルから、リモートメモリにアクセス可能 (現在は、ノード内に限定)
- MPIとの併用可能

NVSHMEMプログラム

```
#include <shmem.h>
#include <shmemx.h>

    shmemx_init_attr_t  attr;
    attr.mpi_comm = MPI_COMM_WORLD;
    shmemx_init_attr( SHMEMX_INIT_WITH_MPI_COMM, &attr );
    npes = shmem_n_pes();
    mype = shmem_my_pe();
    ...
    (NVSHMEMでプロセス間のデータ交換)
    ...
```

プロセス数

プロセスID
(0 … npes-1)

マルチ・プロセス (NVSHMEM)

```
cudaSetDevice( mype );
prev_pe = (mype - 1) % npes;
next_pe = (mype + 1) % npes;

a     = (float*) shmem_malloc( nx * (my_ny+2) * sizeof(float) );
a_new = (float*) shmem_malloc( nx * (my_ny+2) * sizeof(float) );

while ( error > tol ) {
    ...
    kern_jacobi<<< ..., stream >>>(
        a_new, a, error_d, nx, iy_start, iy_end, ... );
    shmem_barrier_all_stream( stream );
    ...
    swap( a_new, a );
}
```

GPUカーネル

NVSHMEM

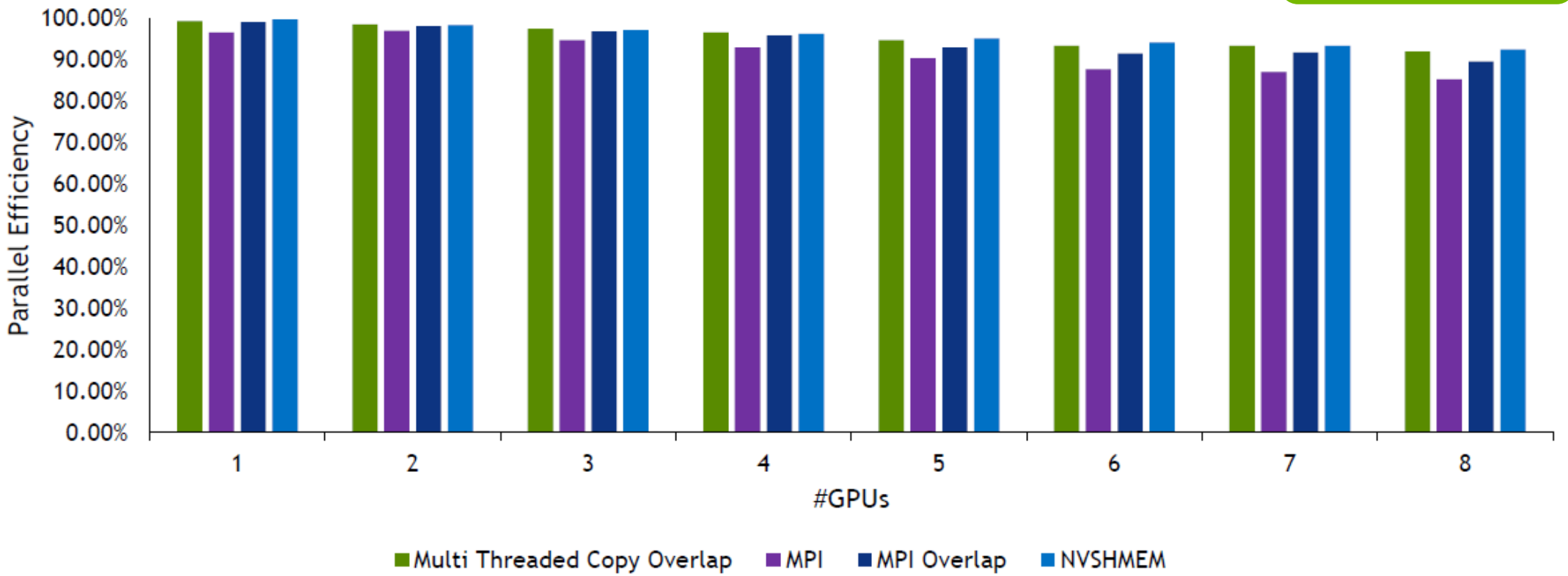
```
__global__ void kern_jacobi( ... ) {
    int ix = ...;
    int iy = ...;
    if ( out of range ) return;
    new_val = 0.25 * ( a[ ix + (iy-1)*nx ] + a[ ix + (iy+1)*nx ]
                     + a[ (ix-1) + iy*nx ] + a[ (ix+1) + iy*nx ] );
    a_new[ ix + iy*nx ] = new_val;
    if ( iy == iy_start )
       shmem_float_p( a_new + ix + iy_end*ny, new_val, prev_pe );
    if ( iy == iy_end-1 )
       shmem_float_p( a_new + ix             , new_val, next_pe );
    my_error = fabs( new_val - a[ ix + iy*nx ] );
    atomicMax( error, my_error );
}
```

隣接GPUメモリに
直接書き込み

マルチ・プロセス (NVSHMEM)

DGX-1V, 1000反復

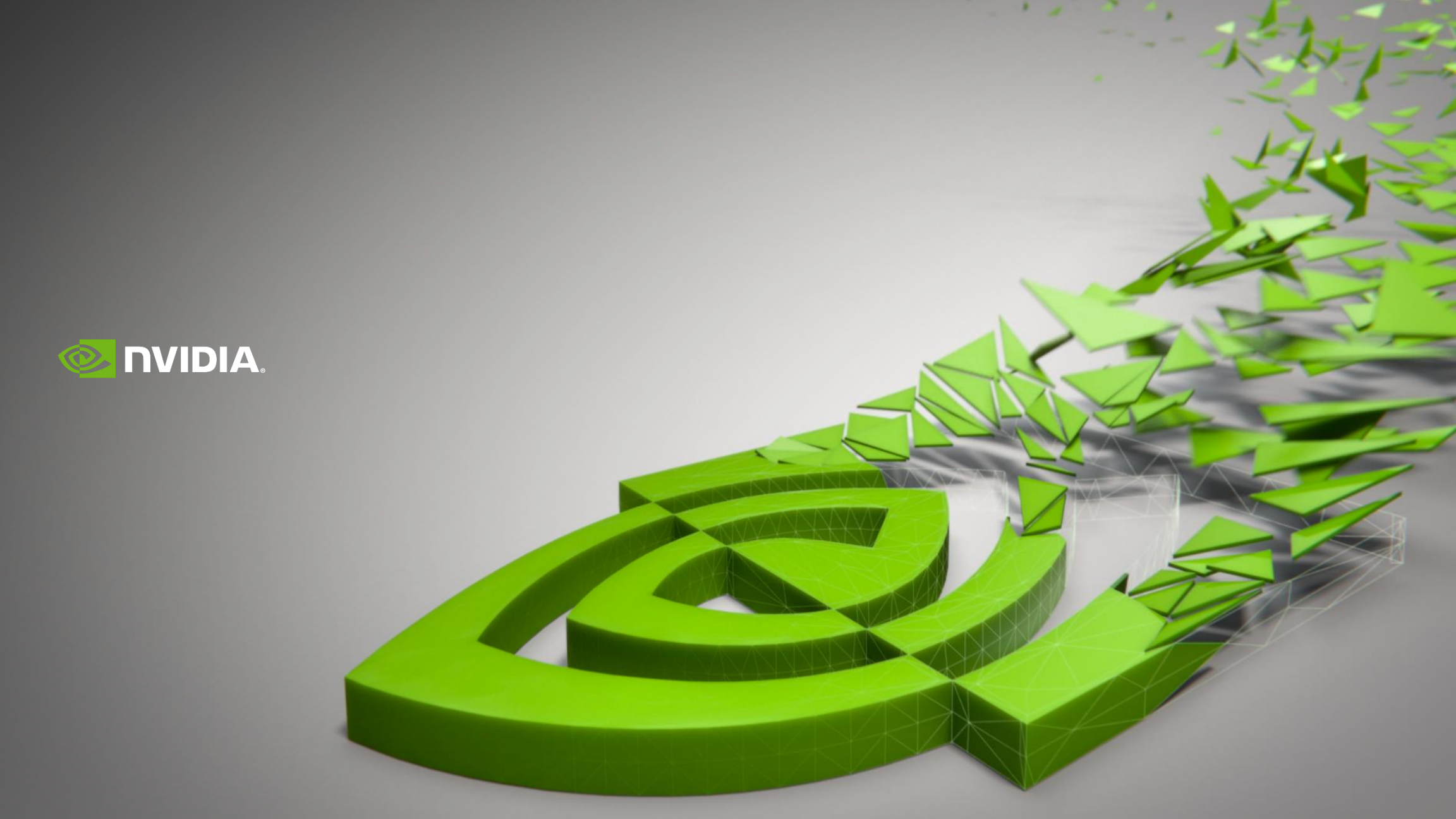
マルチ・スレッドと
同レベルの性能



まとめ

	データ交換(通信)	P2P	マルチノード
シングル・スレッド	Memcpy	○	X
マルチ・スレッド	Memcpy	○	X
	P2Pアクセス	必須	X
マルチ・プロセス	MPI	○	可能
	NVSHMEM	必須	可能(*)

(*) 現時点ではNVSHMEMはノード内限定


NVIDIA