VOLTA GPU 架构和性能优化

Xiaonan Tian

PGI

AGENDA

Volta V100 Overview

GPU Programming Models: CUDA and OpenACC

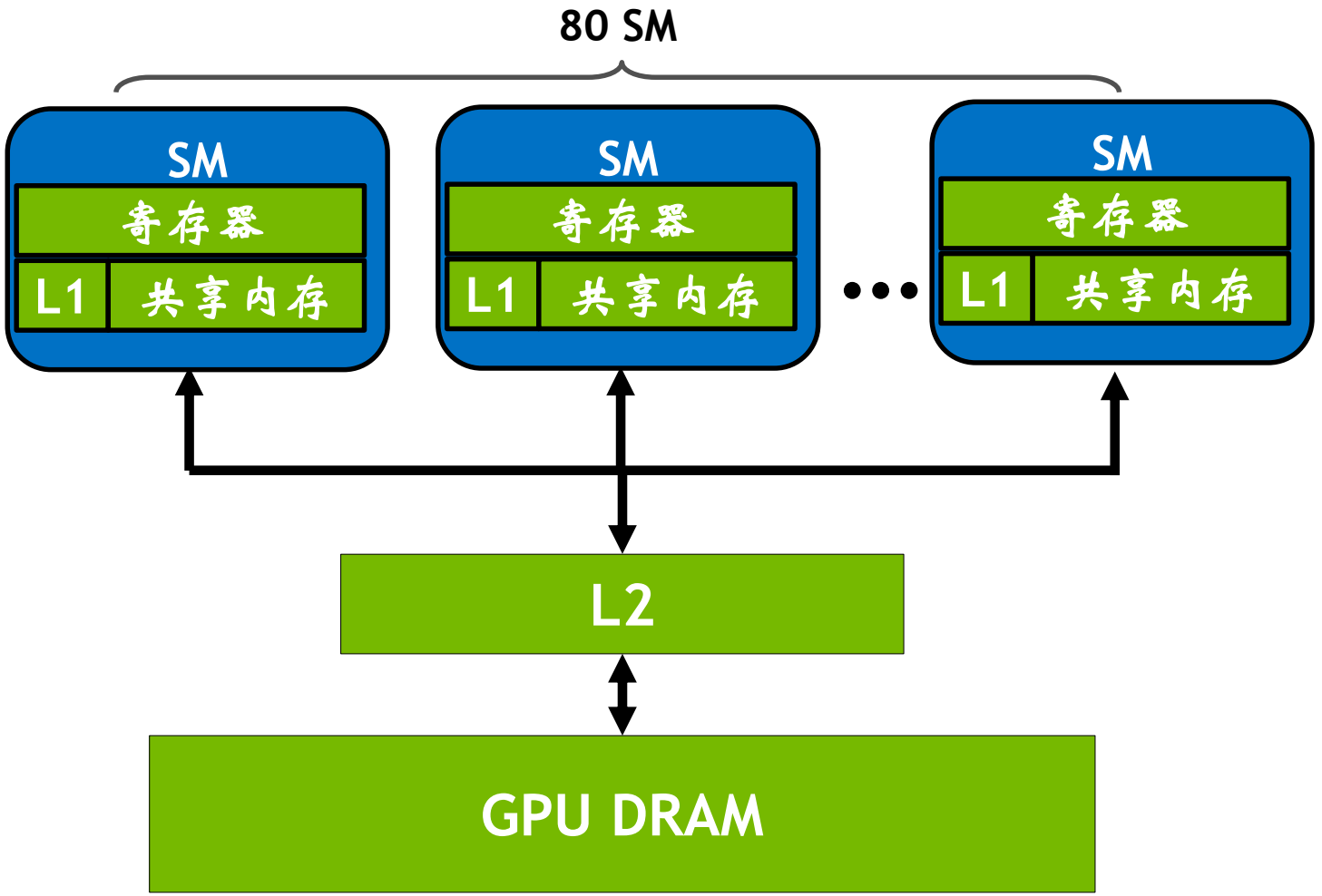
CUDA Optimization Tips

Asynchronous Execution: Stream and Concurrency

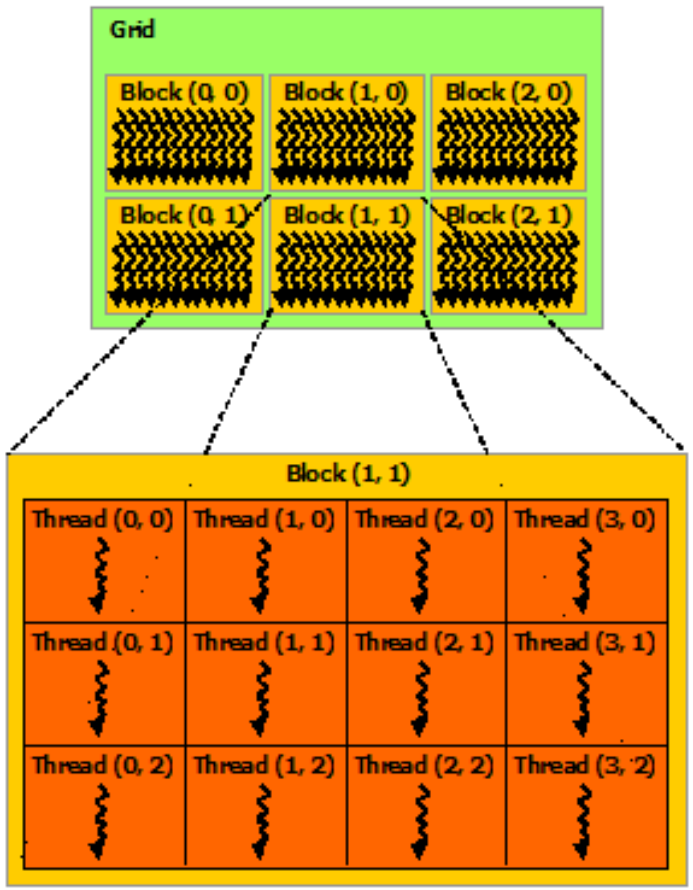
Multiple GPUs Programming

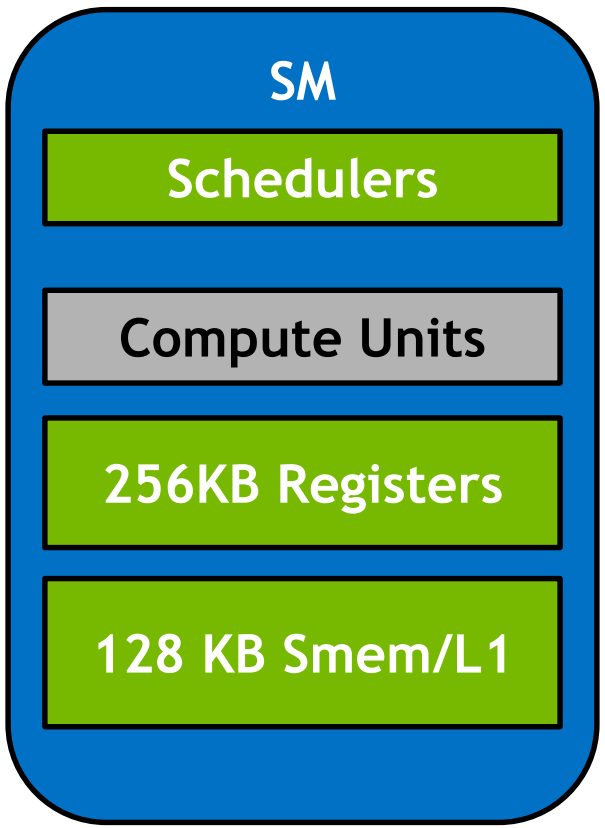
New Features in Volta GPU

VOLTA V100



Threads and SM





AGENDA

Volta V100 Overview

GPU Programming Models: CUDA and OpenACC

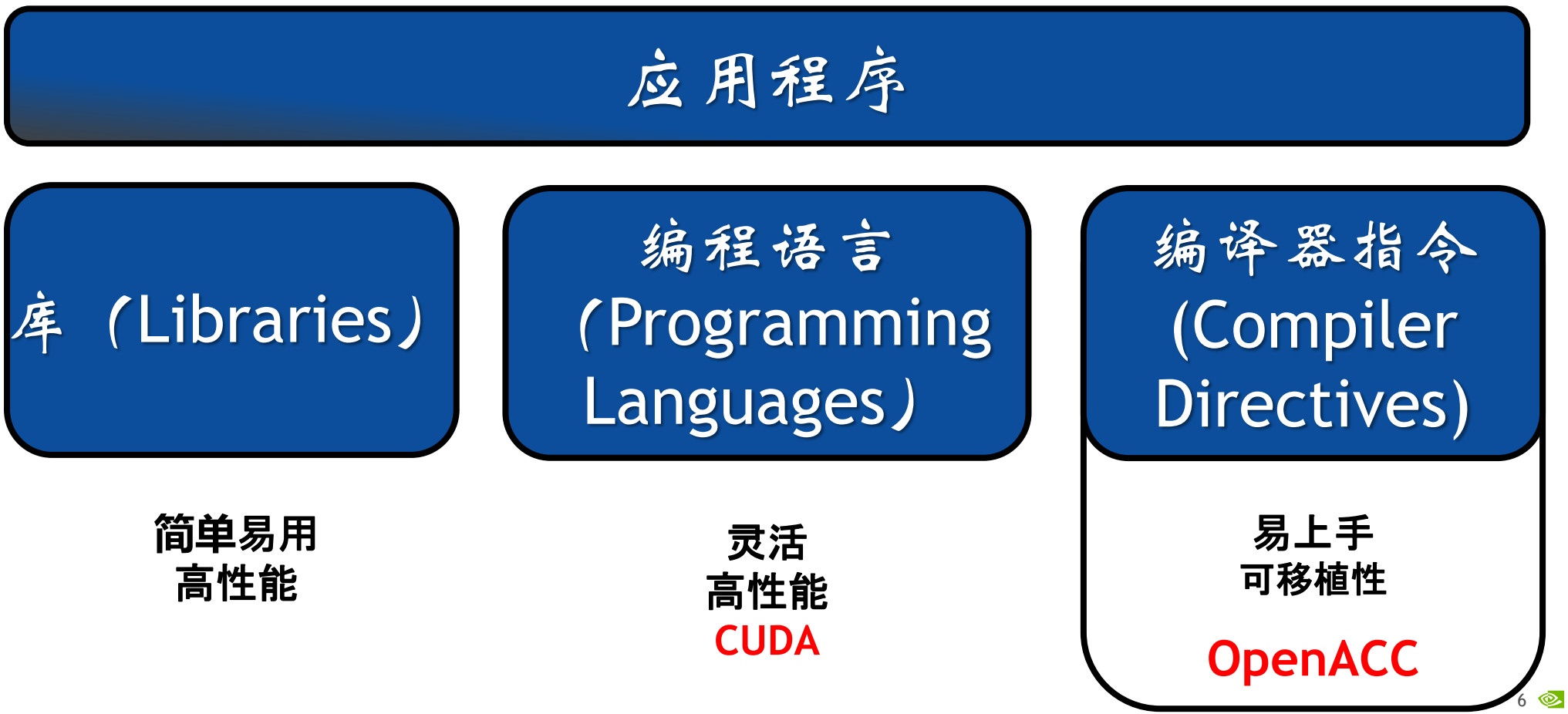
CUDA Optimization Tips

Asynchronous Execution: Stream and Concurrency

Multiple GPUs Programming

New Features in Volta GPU

GPU 编程模型

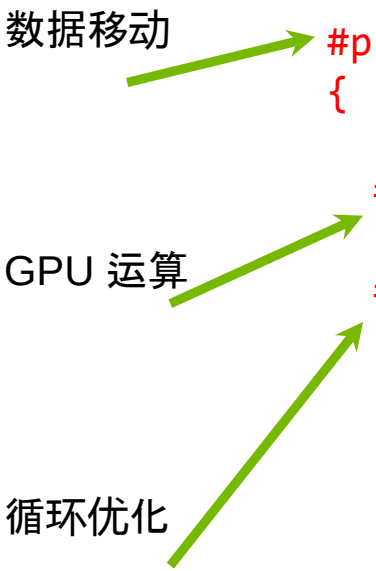


CUDA 示例

```
__global__ void
vectorAdd(const float *A, const float *B,
                  float *C, int numElements)
{
  int i = blockDim.x * blockIdx.x + threadIdx.x;
  if (i < numElements) {
     C[i] = A[i] + B[i];
  }
}

int main(void)
{
  …
  cudaMalloc((void **)&d_A, size);
  cudaMalloc((void **)&d_B, size);
  cudaMalloc((void **)&d_C, size);
  …
  cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
  cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
  //启动GPU 内核代码
  vectorAdd<<<20, 256>>>(d_A, d_B, d_C, numElements);
  //拷贝数据到到CPU内存
  cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
  cudaFree(d_A);
  cudaFree(d_B);
  cudaFree(d_C);
}
```

OpenACC 示例

OpenACC 指令




```
#pragma acc data copyin(a,b) copyout(c)
{
  ...
  #pragma acc parallel
  {
  #pragma acc loop gang vector
      for (i = 0; i < n; ++i) {
          c[i] = a[i] + b[i];
          ...
      }
  }
  ...
}
```

OpenACC
Directives for Accelerators

渐进式

单个源代码

可交互性

可移植性

支持GPU，CPU（x86和Power PC）

AGENDA

Volta V100 Overview

GPU Programming Models: CUDA and OpenACC

CUDA Optimization Tips

Asynchronous Execution: Stream and Concurrency

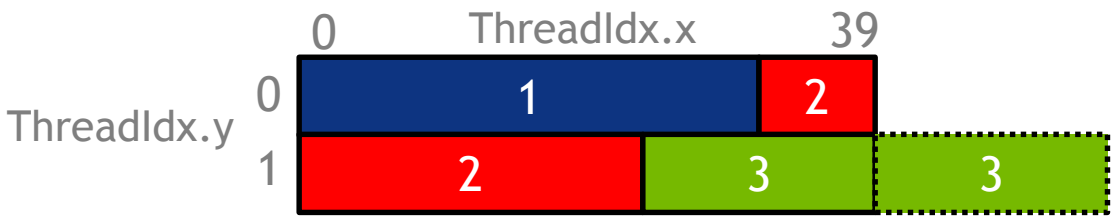
Multiple GPUs Programming

New Features in Volta GPU

CUDA Optimization Tips

- Thread Diverge
- Data Locality Optimization
  - ✓ Global memory Access: Coalesced Memory Access
  - ✓ Shared Memory Access: Bank Conflict
  - ✓ Constant Memory Access: Unified Access
  - ✓ Take Advantage of GPU Registers

Thread Diverge



Instructions, time

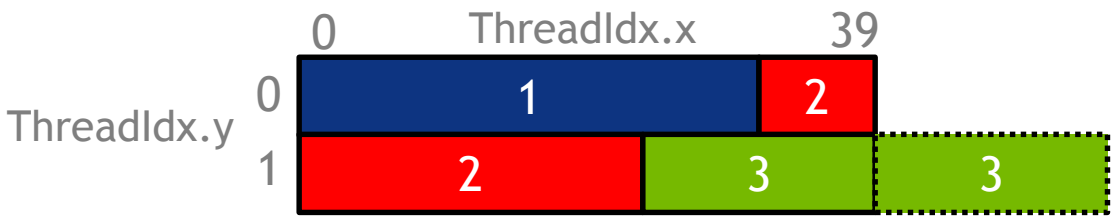
```
A;
if(threadIdx.y==0)
  B;
else
  C;
D;
```

Warp 1: 0 … 31

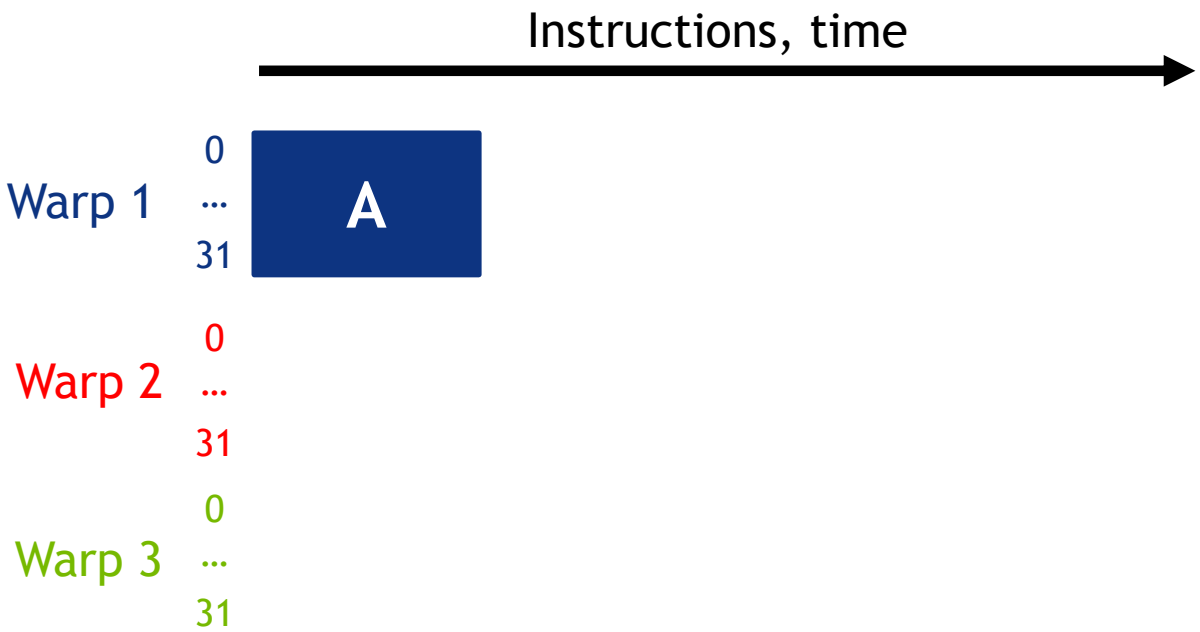
Warp 2: 0 … 31

Warp 3: 0 … 31

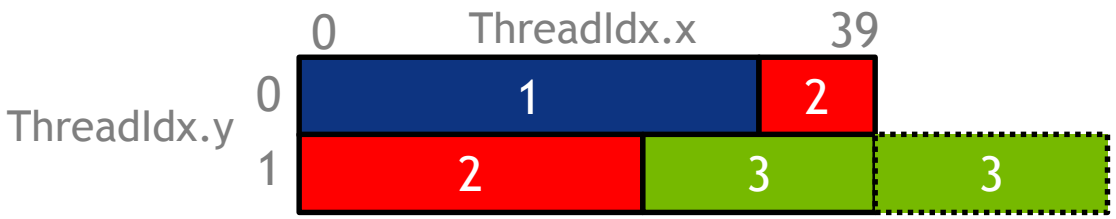
Thread Diverge



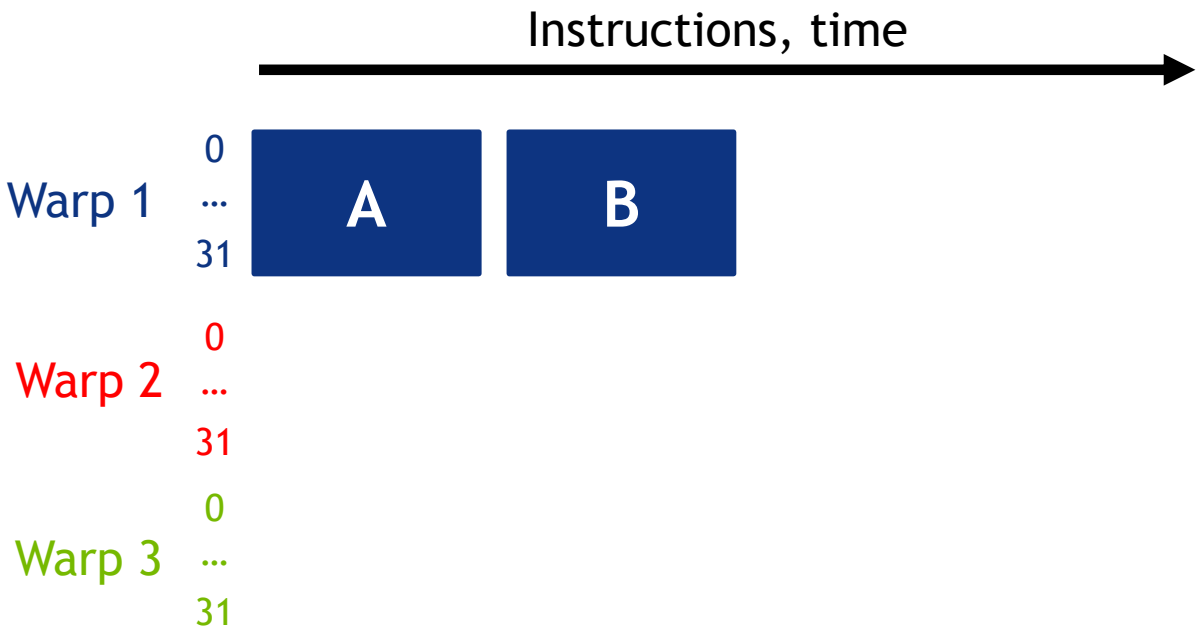

```
A;
if(threadIdx.y==0)
  B;
else
  C;
D;
```



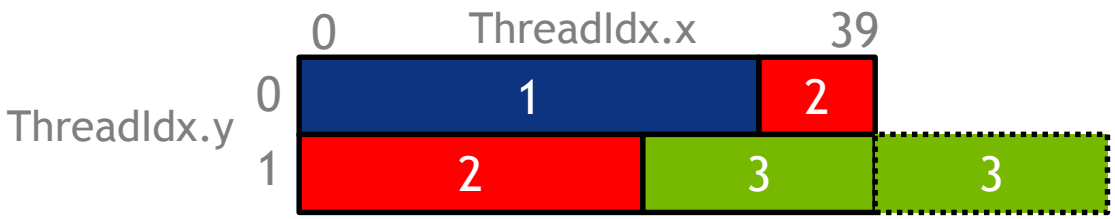
Thread Diverge



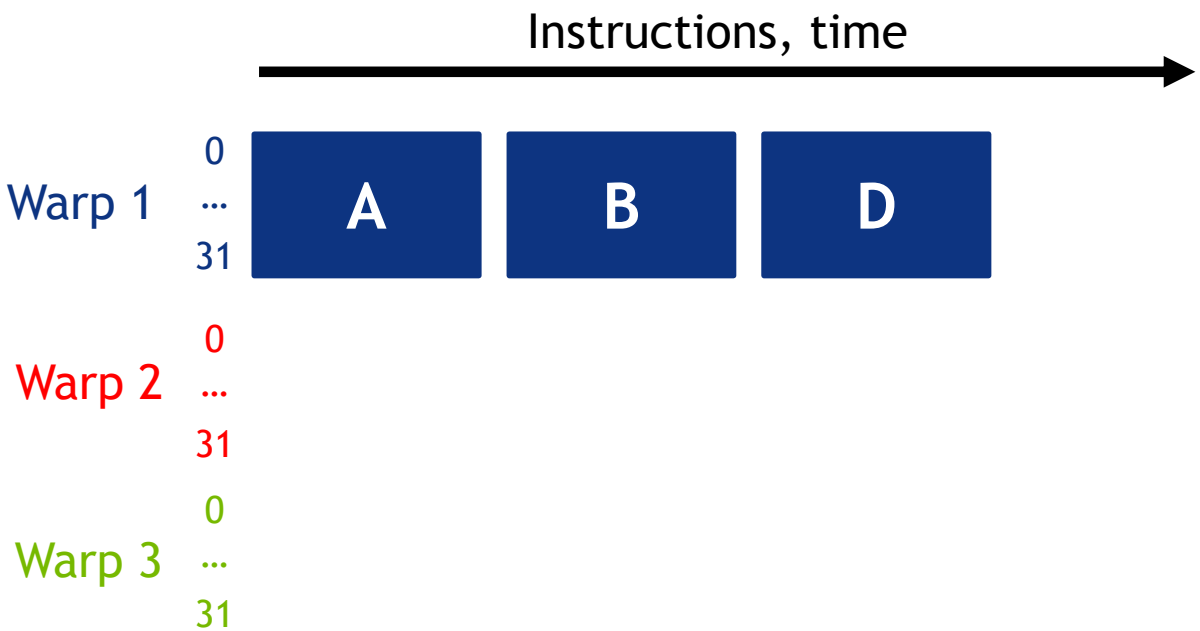

```
A;
if(threadIdx.y==0)
  B;
else
  C;
D;
```



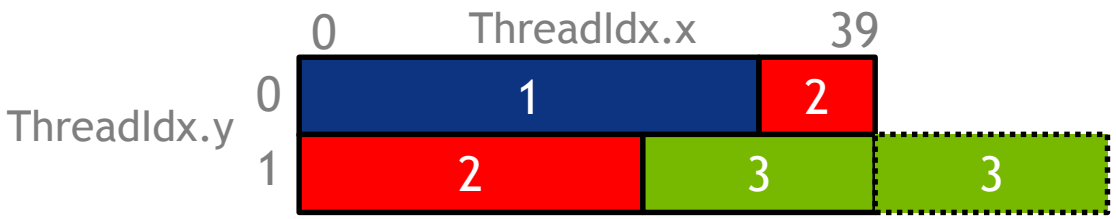
Thread Diverge



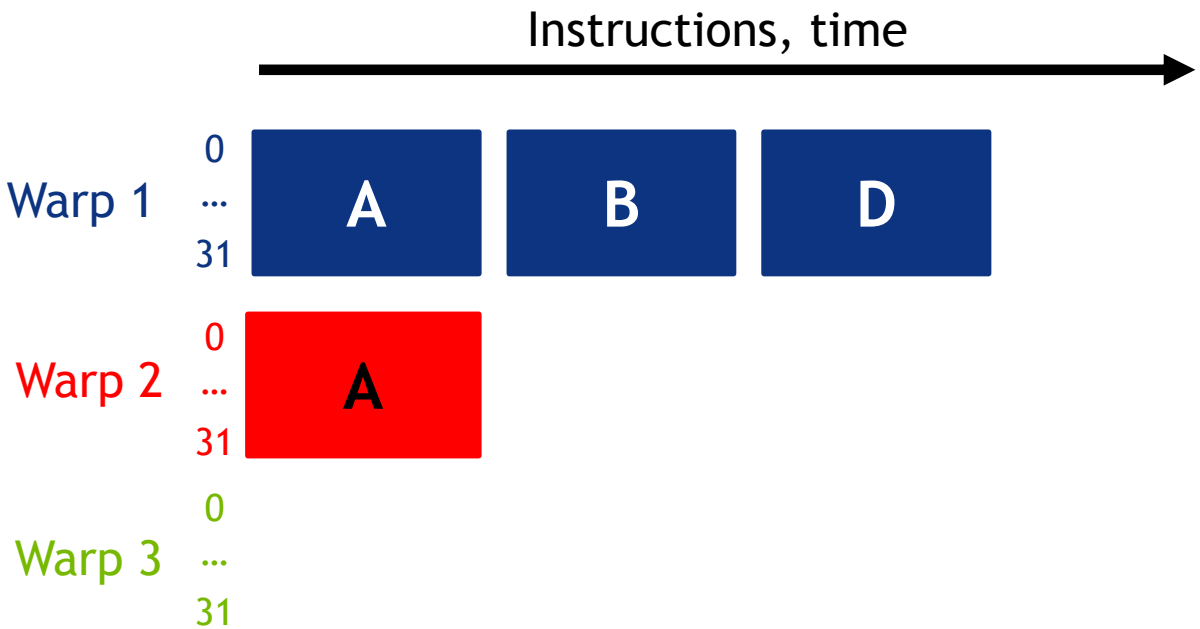

```
A;
if(threadIdx.y==0)
  B;
else
  C;
D;
```



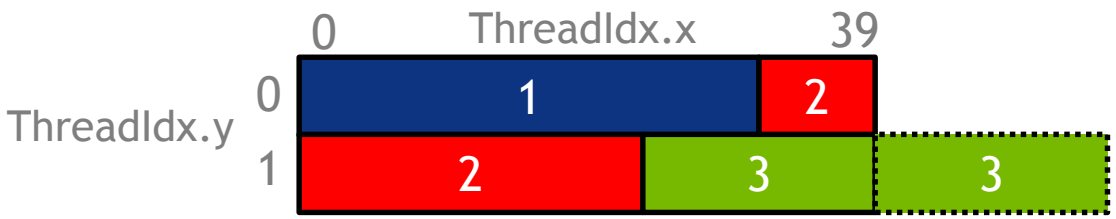
Thread Diverge



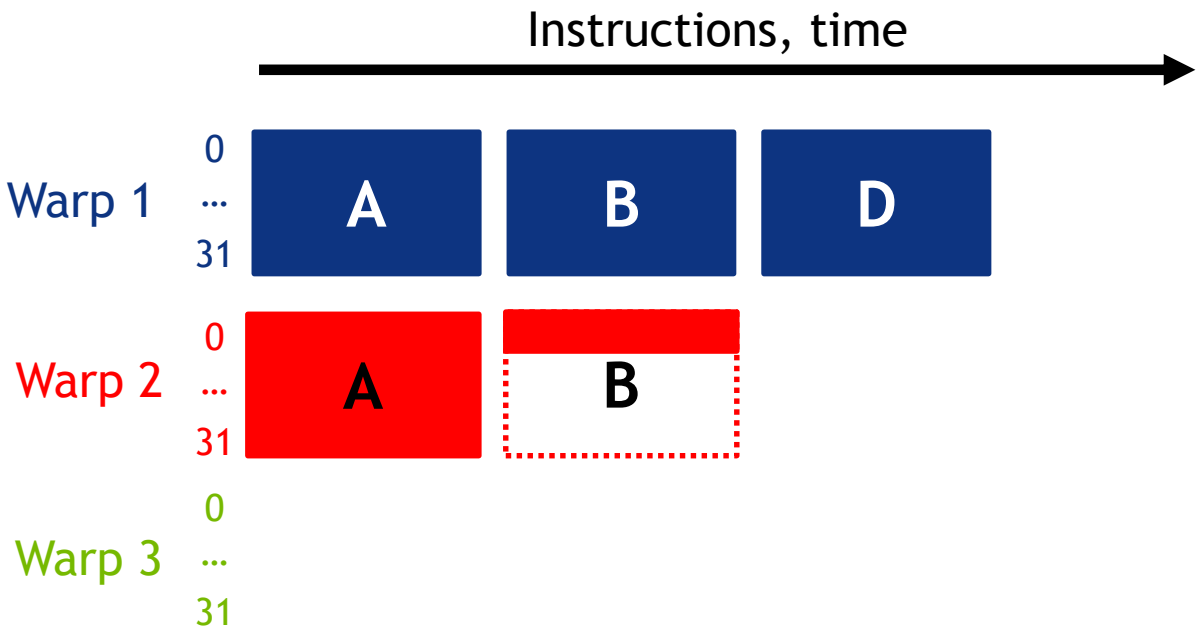

```
A;
if(threadIdx.y==0)
  B;
else
  C;
D;
```



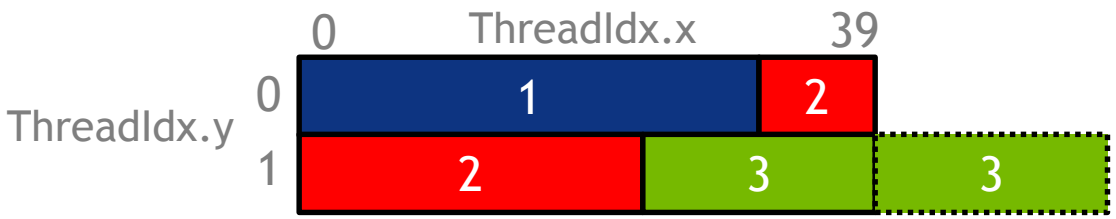
Thread Diverge



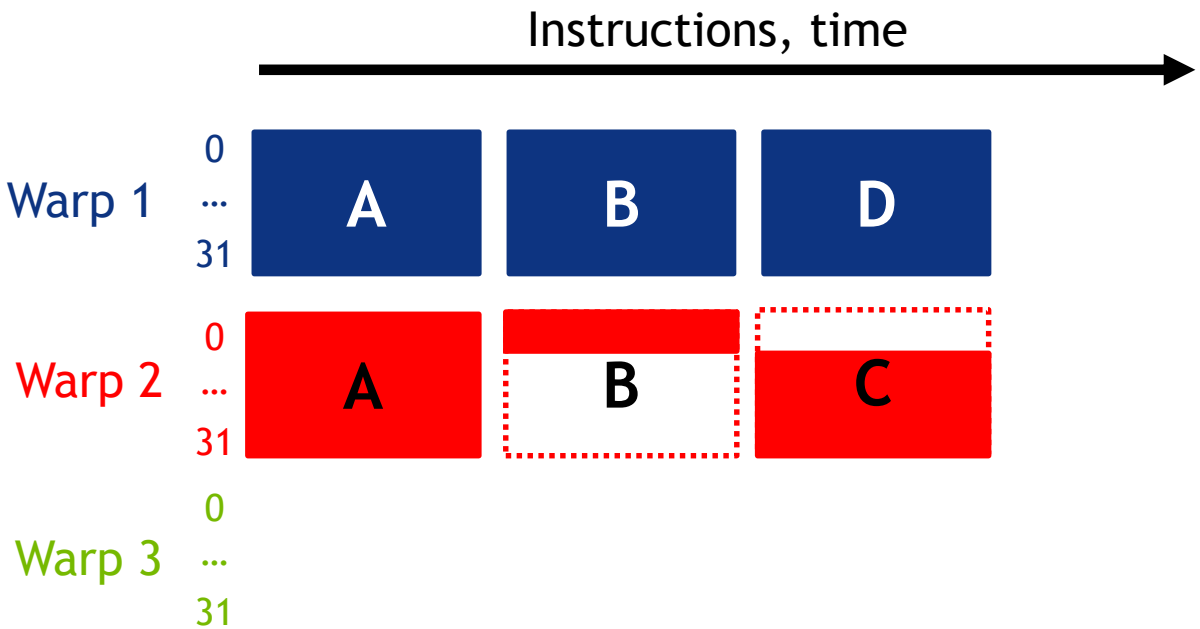

```
A;
if(threadIdx.y==0)
  B;
else
  C;
D;
```



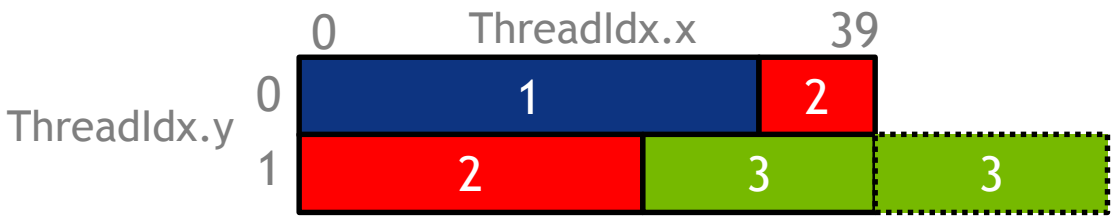
Thread Diverge



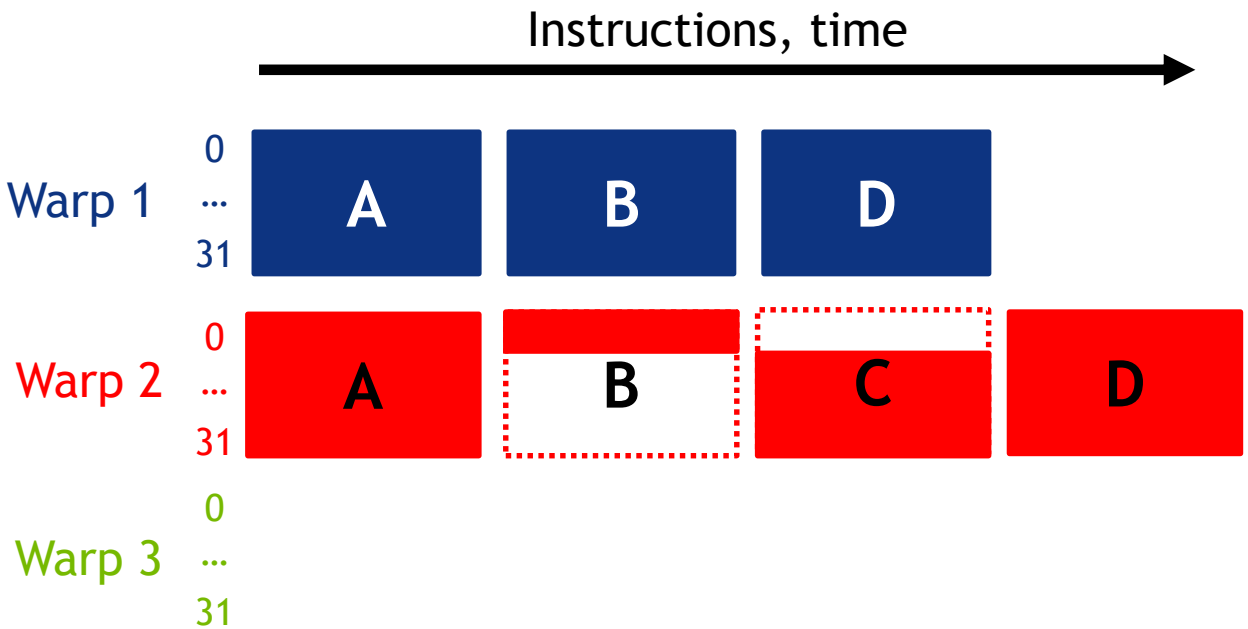

```
A;
if(threadIdx.y==0)
  B;
else
  C;
D;
```



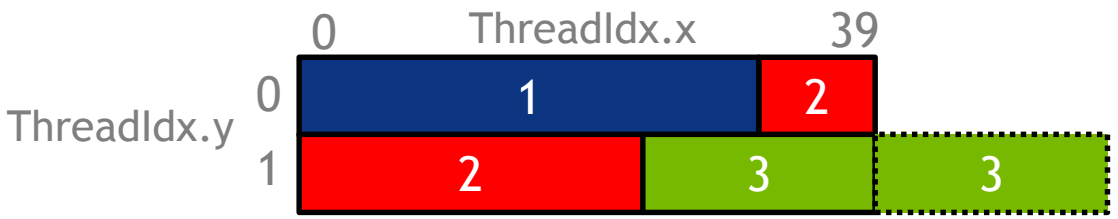
Thread Diverge



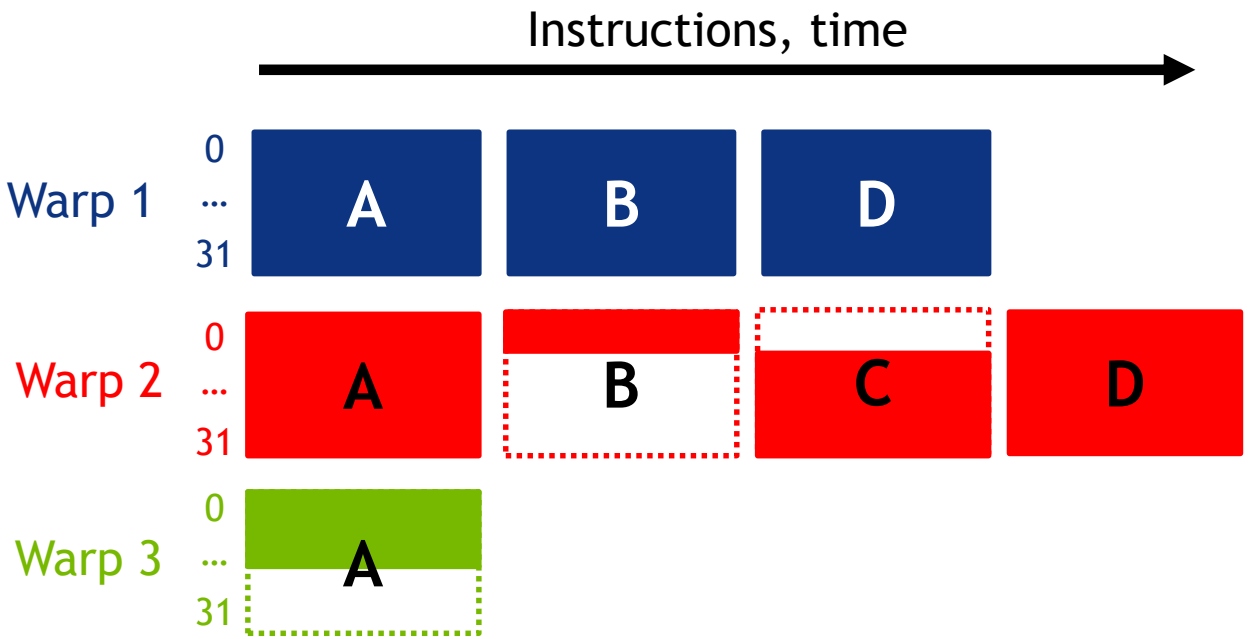

```
A;
if(threadIdx.y==0)
  B;
else
  C;
D;
```



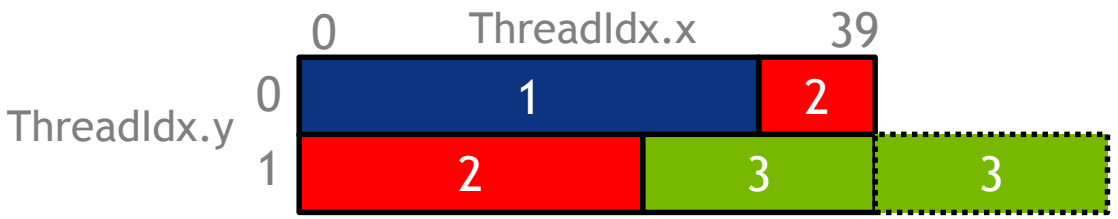
Thread Diverge



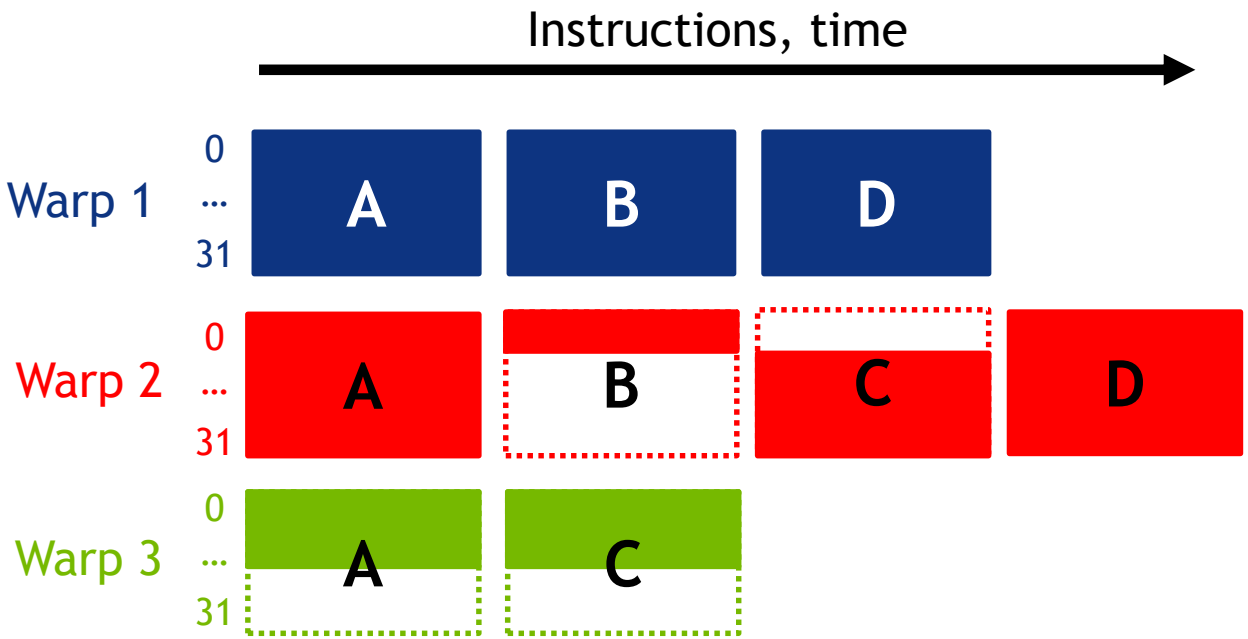

```
A;
if(threadIdx.y==0)
  B;
else
  C;
D;
```



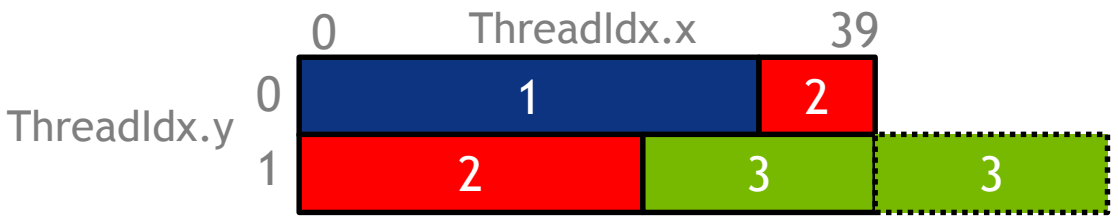
Thread Diverge



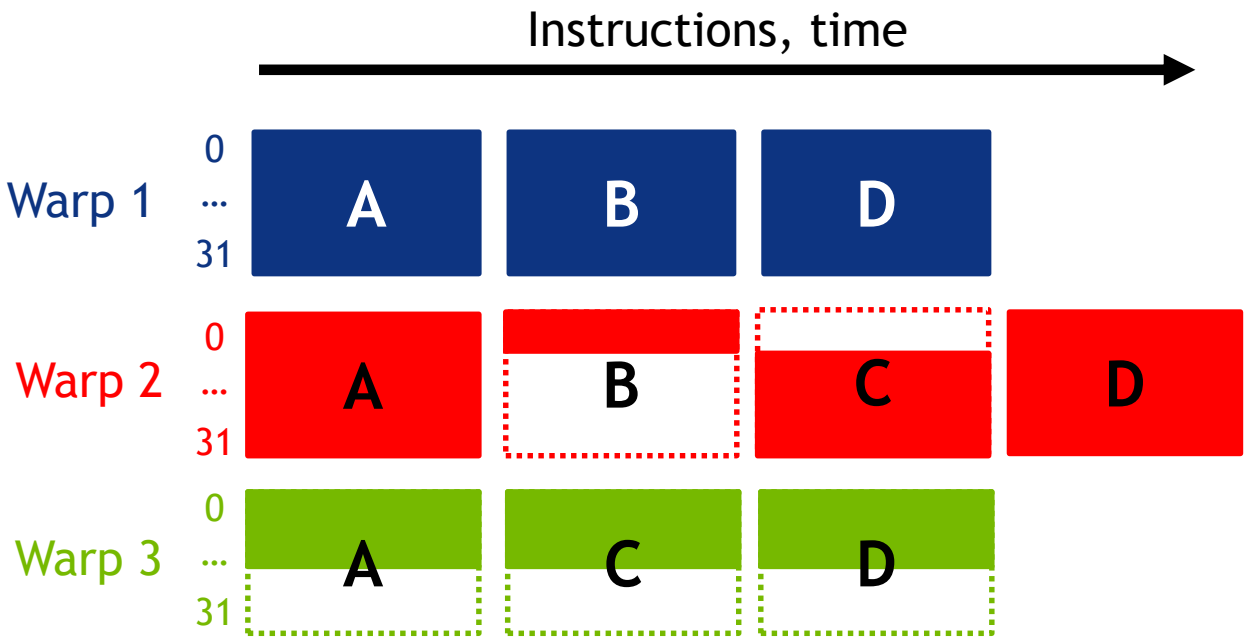

```
A;
if(threadIdx.y==0)
  B;
else
  C;
D;
```



Thread Diverge




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

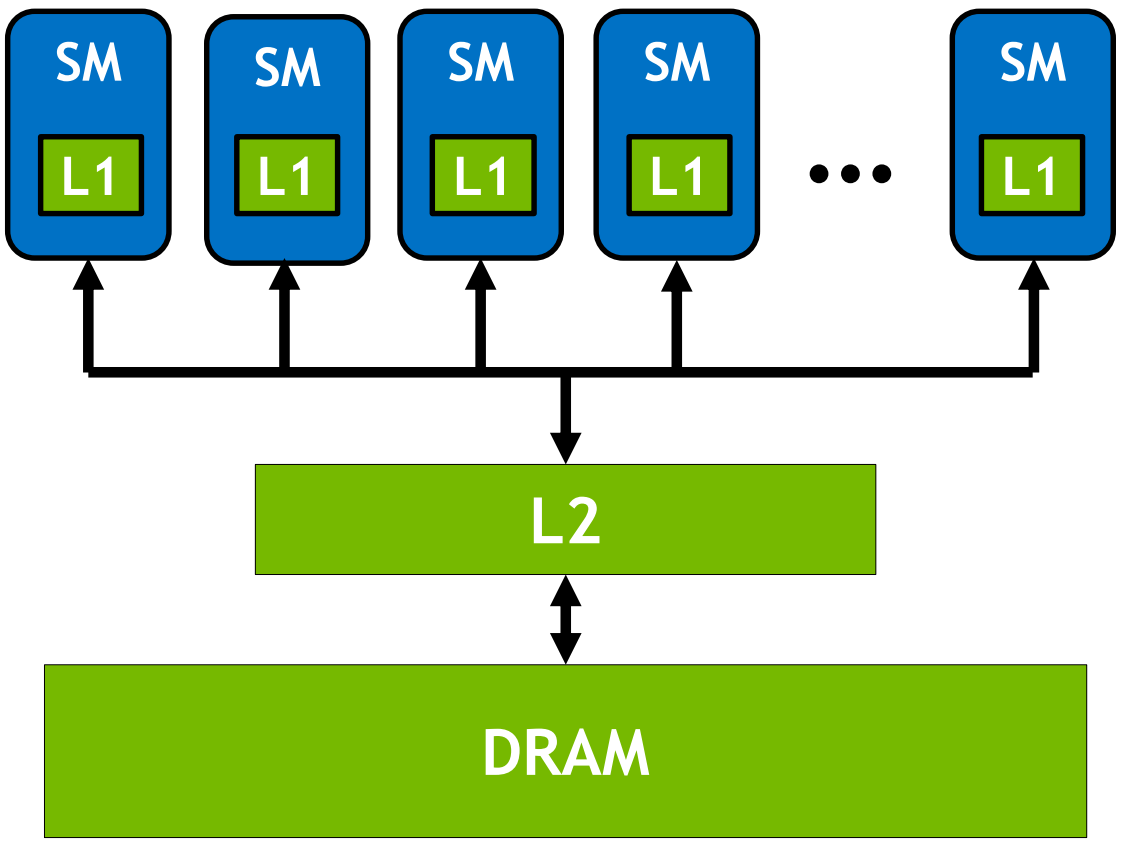


优化建议

- 尽可能减少线程分叉
- 不同的warp执行不同的路径是不影响性能的

Volta Memory System

V100



80 Streaming Multiprocessors
256KB register file (20 MB)

Unified Shared Mem / L1 Cache
128KB, Variable split
(~10MB Total, 14 TB/s)

6 MB L2 Cache
(2.5TB/s Read, 1.6TB/s Write)

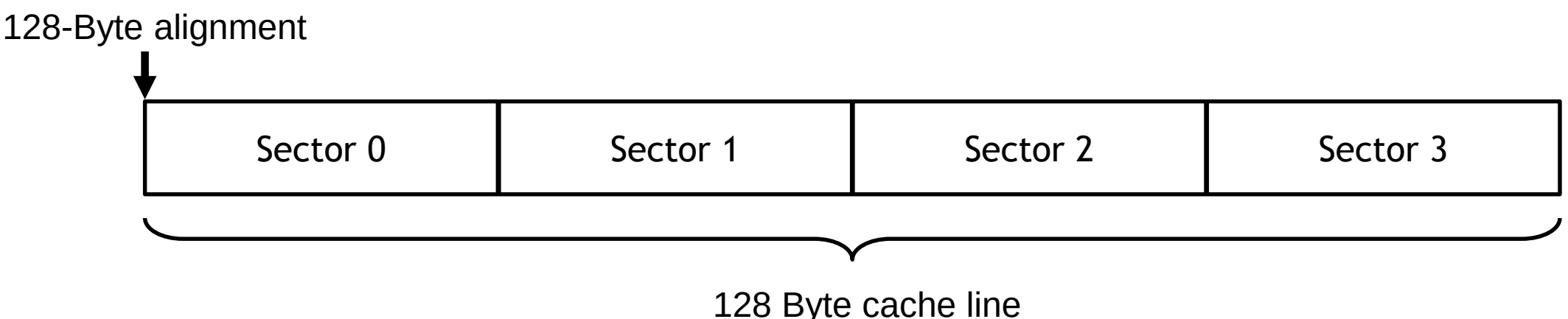
16/32 GB HBM2 (900 GB/s)
“Free” ECC.

Cache Lines & Sectors

Moving data between L1, L2, DRAM

Memory access granularity = **32 Bytes = 1 sector**
(32B for Maxwell, Pascal, Volta. Kepler and before: variable, 32B or 128B, depending on architecture, access type, caching / non-caching options)

A **cache line is 128 Bytes**, made of **4 sectors.**
Cache ”management” granularity = 1 cache line

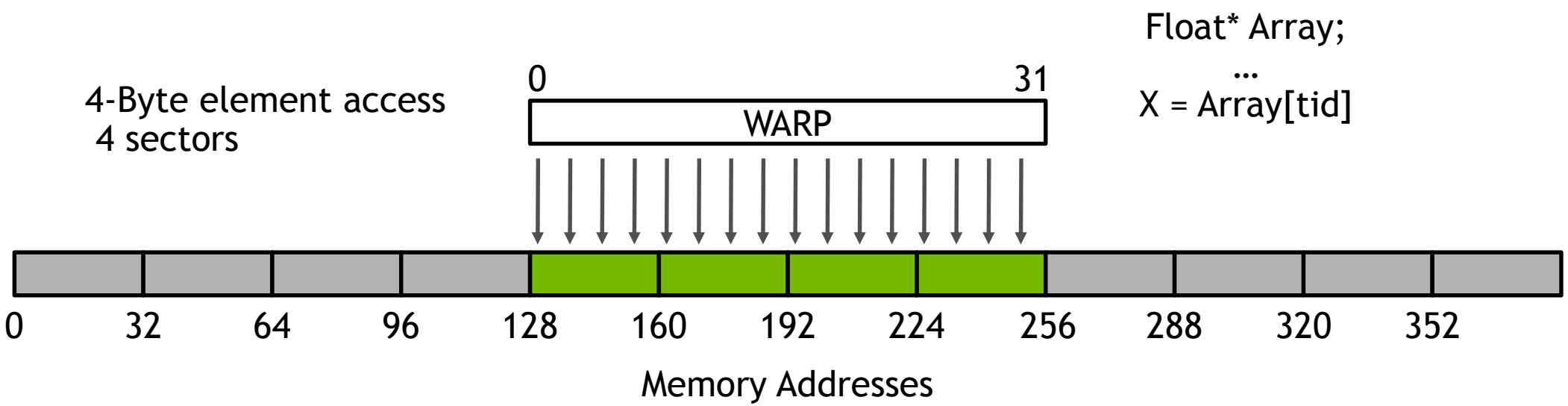


Access Patterns

Warps and Sectors

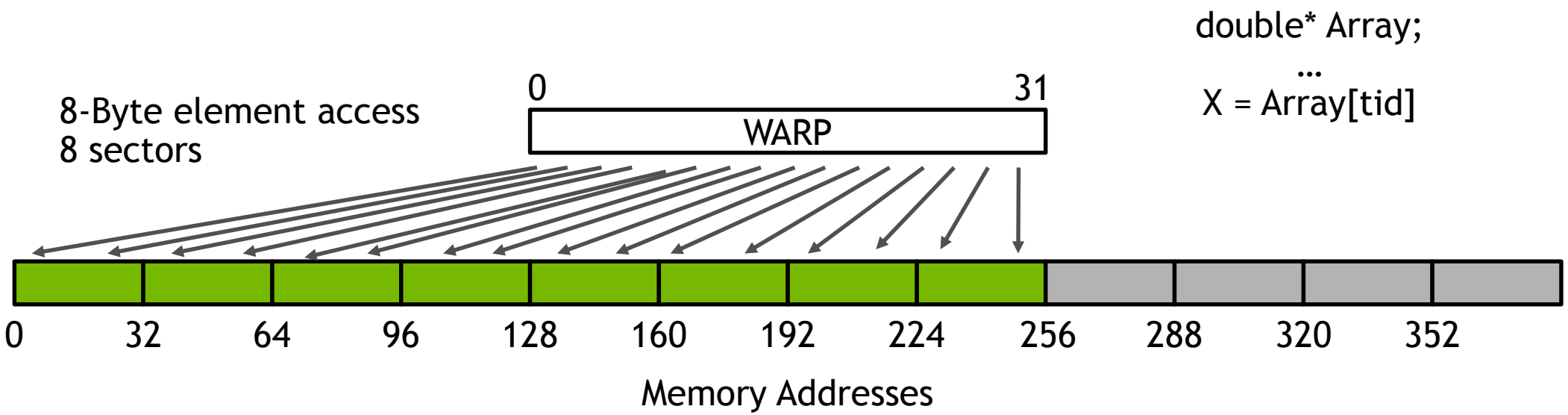
For each warp: How many **sectors** needed?

Depends on addresses, active threads, access size.
Natural element sizes = 1B, 2B, 4B, 8B, 16B.



Access Patterns

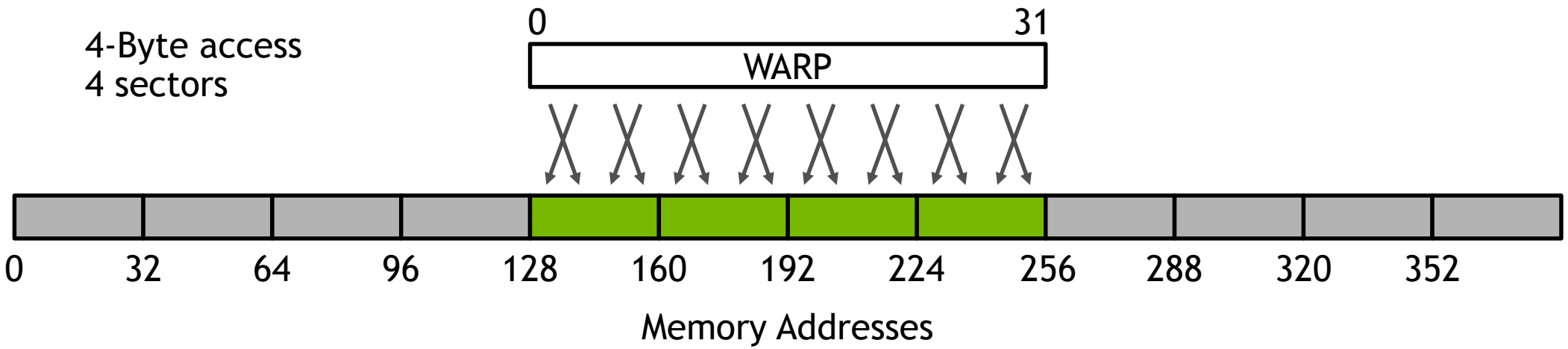
Warps and Sectors



Examples of 8-byte elements: long long, int2, double, float2

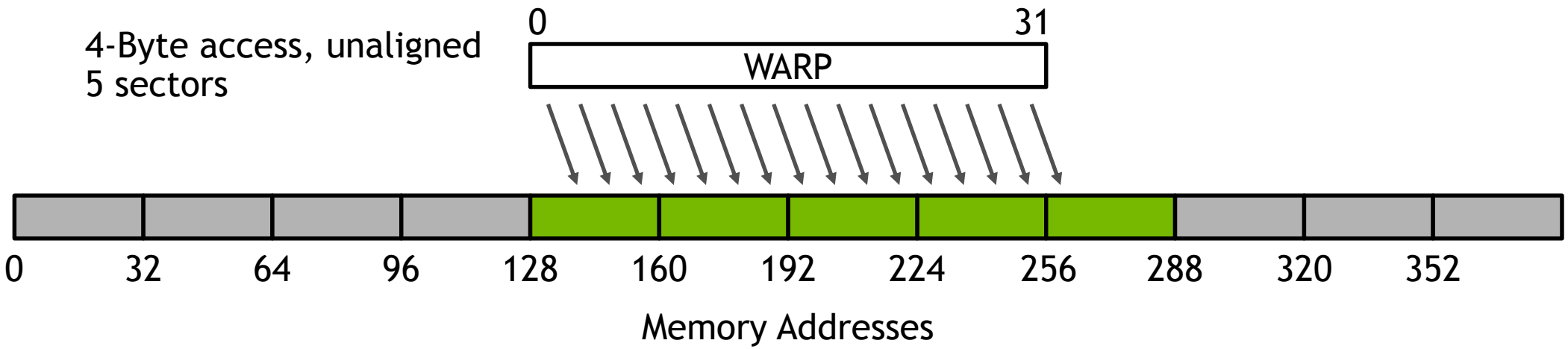
Access Patterns

Warps and Sectors



Access Patterns

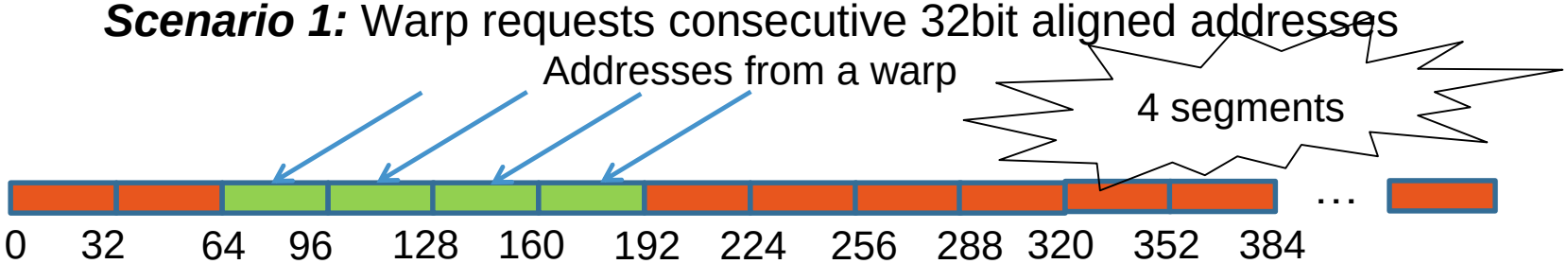
Warps and Sectors



128 bytes requested, 160 bytes read (80% efficiency)

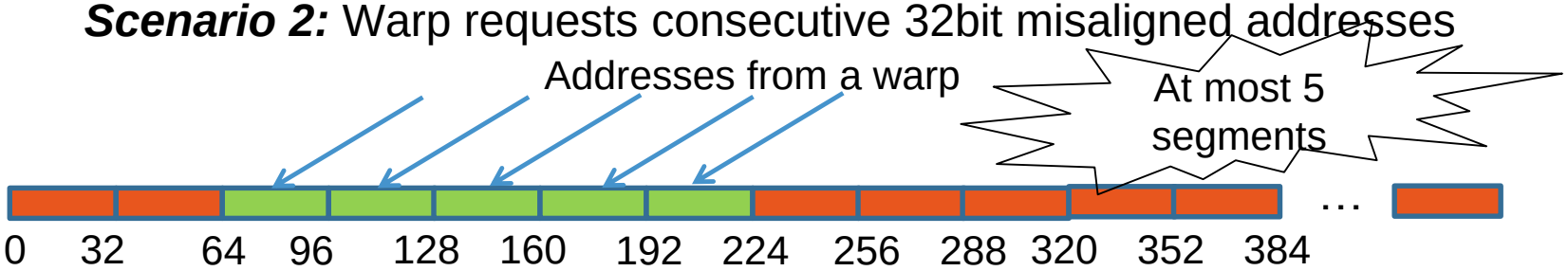
Memory Coalesced Access: each thread request a float (4 bytes) from global memory

Scenario 1: Warp requests consecutive 32bit aligned addresses



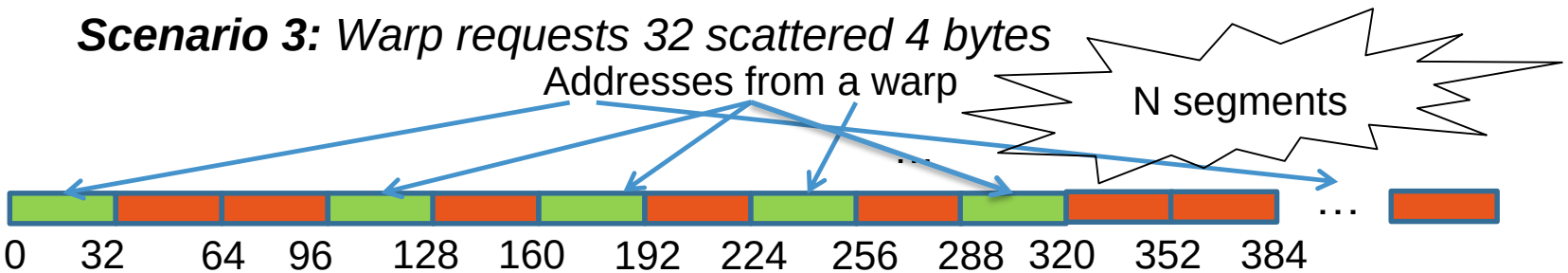
X = A[tid]

Scenario 2: Warp requests consecutive 32bit misaligned addresses



X = A[tid + 1]

Scenario 3: *Warp requests 32 scattered 4 bytes*



X = A[coeff * tid]

Access Patterns

Takeaways

- Know your access patterns
- Use the profiler (metrics, counters) to check how many sectors are moved. Is that what you expect? Is it optimal?
- Using the largest type possible (e.g. float4) will maximize the number of sectors moved per instruction

Shared Memory

Scratch-pad memory on each SM

- **User-managed cache**, HW does not evict data
- Data written to SMEM stays there till user overwrites

Useful for:

- **Storing frequently-accessed data**, to reduce DRAM accesses
- **Communication** among threads of a threadblock

Performance benefits compared to DRAM:

- **20-40x lower latency**
- **~15x higher bandwidth**
- Accessed at 4-byte granularity
- GMEM granularity is 32B.

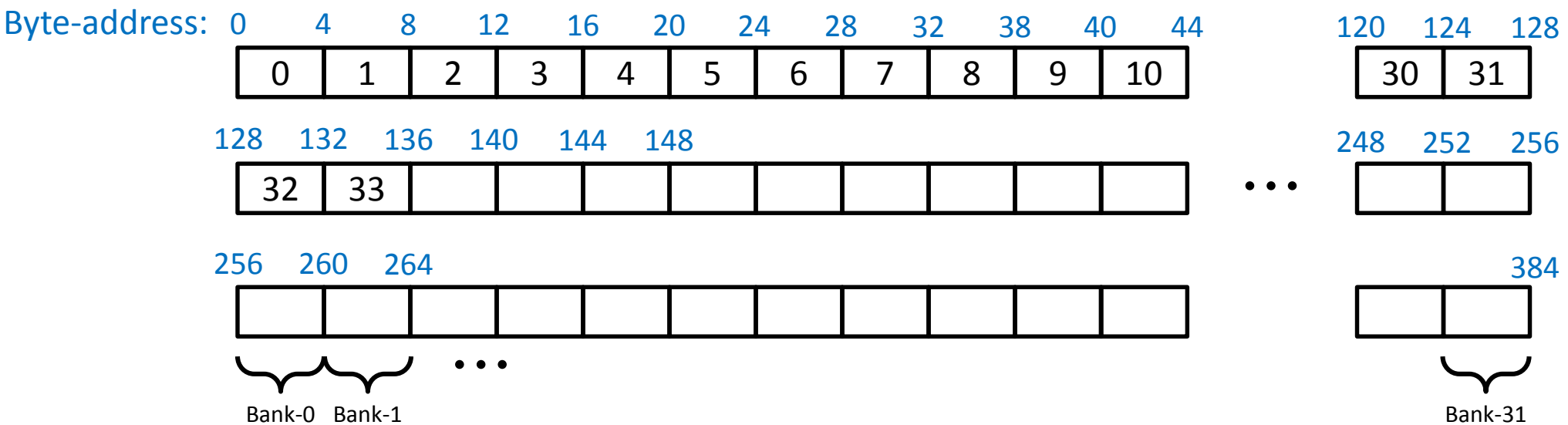
Volta Shared Memory

- **Default 48KB/threadblock, opt in to get 96KB**
- **32 banks, 4 bytes wide**
  - Bandwidth: 4 bytes per bank per clock per SM
    128 bytes per clk per SM
  - V100: ~14 TB/s aggregate across 80 SMs
- **Mapping addresses to banks:**
  - Successive 4-byte words go to successive banks
  - Bank index computation examples:

    (4B word index) % 32
    ((1B word index) / 4) % 32
    8B word spans two successive banks

Logical View Of SMEM Banks

With 4-Bytes data



Shared Memory Instruction Operation

Threads in a warp provide addresses

HW determines into which 4-byte words addresses fall

Reads (LDS):

Fetch the data, distribute the requested bytes among threads
Multi-cast capable

Writes (STS):

Multiple threads writing the same address: one “wins”

Shared Memory Bank Conflicts

A **bank conflict** occurs when, **inside a warp**:

2 or more threads access within **different** 4B words in the **same bank**

Think: 2 or more threads access different "rows" in the same bank

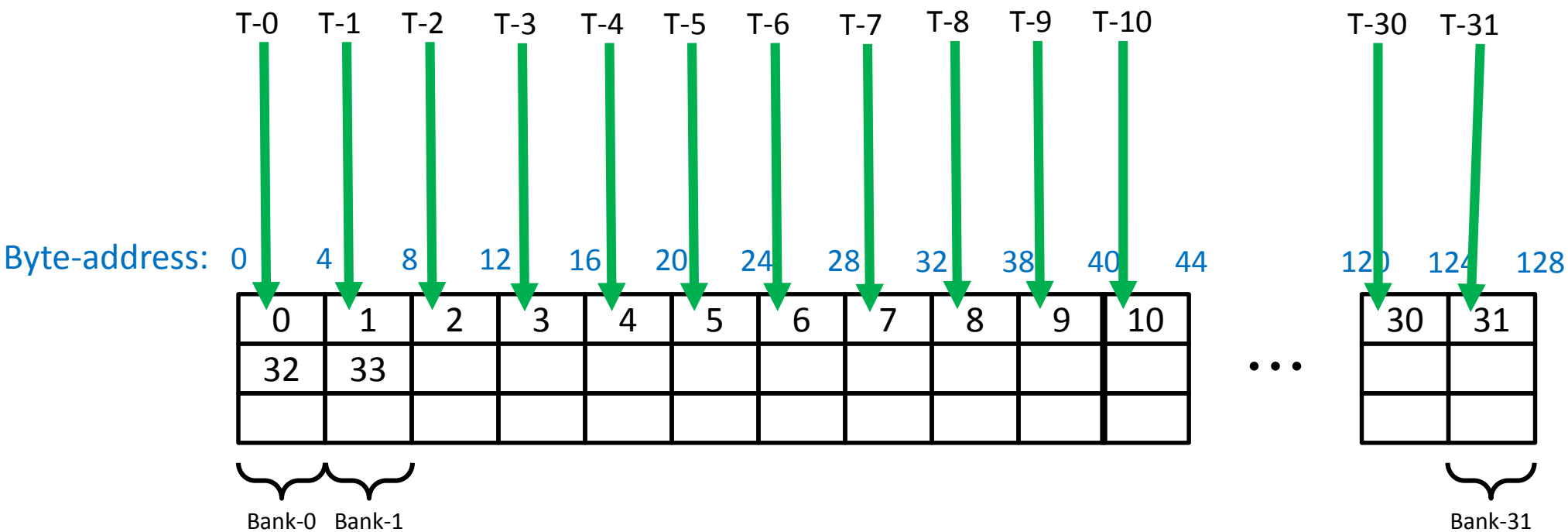
N-way bank conflict: ***N*** threads in a warp conflict

- Increases latency
- Worst case: 32-way conflict → 31 replays
- Each replay adds a few cycles of latency

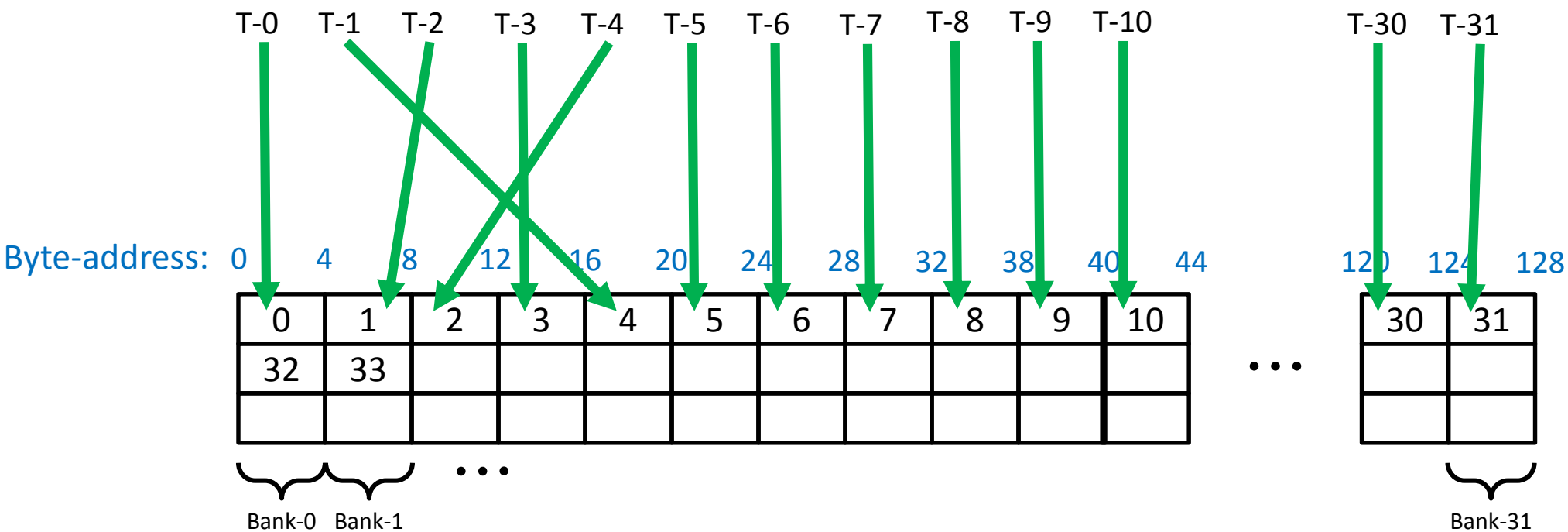
There is **no bank conflict** if:

- Several threads access the same 4-byte word
- Several threads access different bytes of the same 4-byte word

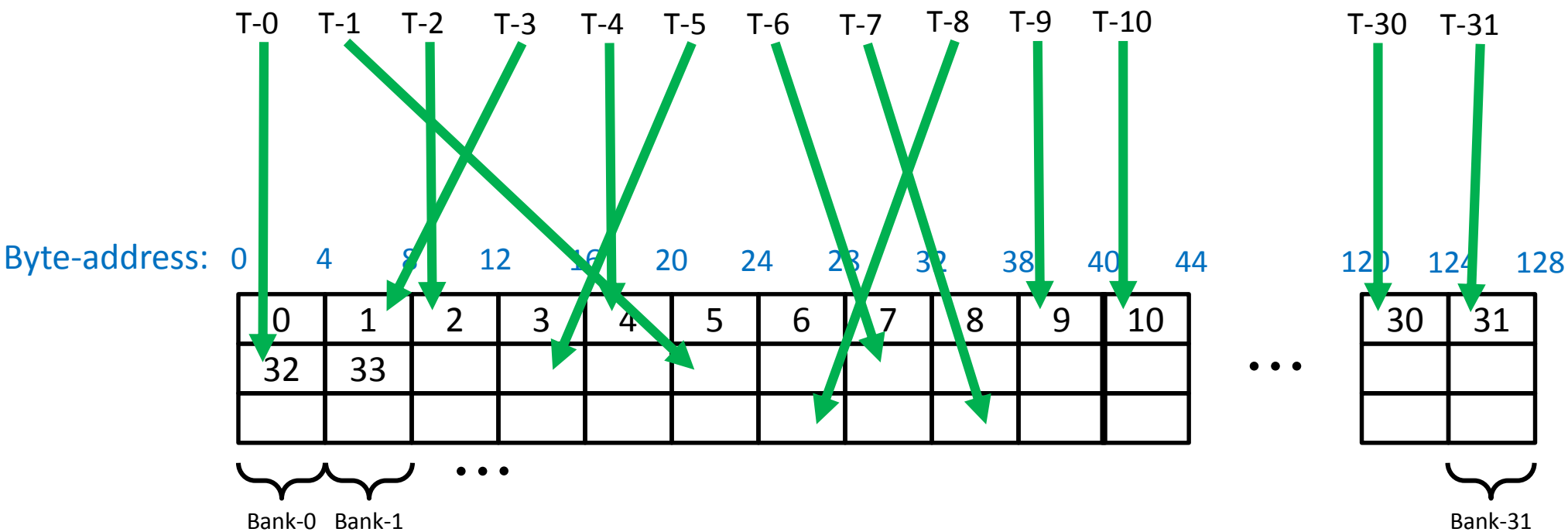
No Bank Conflicts



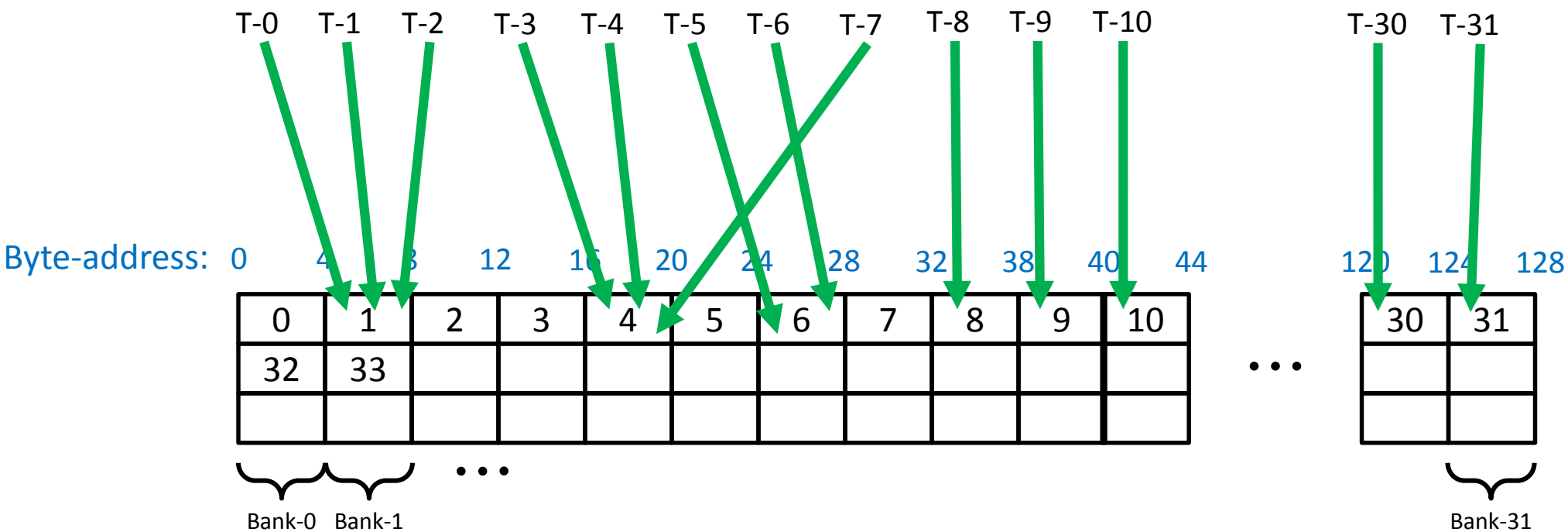
No Bank Conflicts



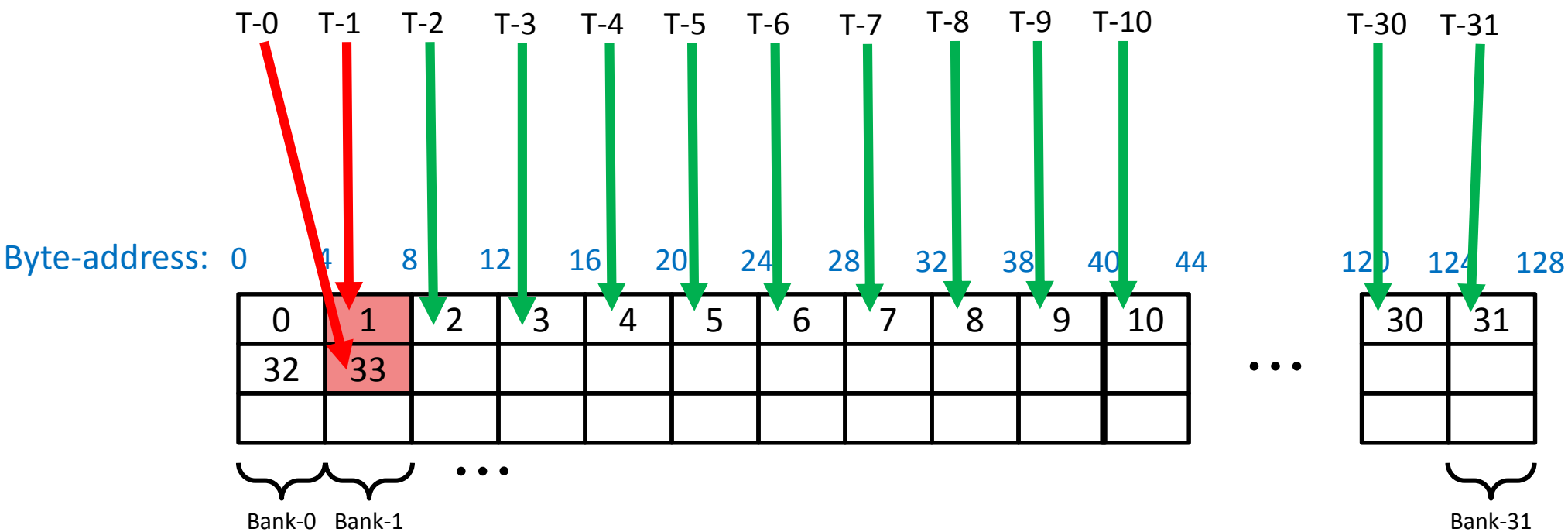
No Bank Conflicts



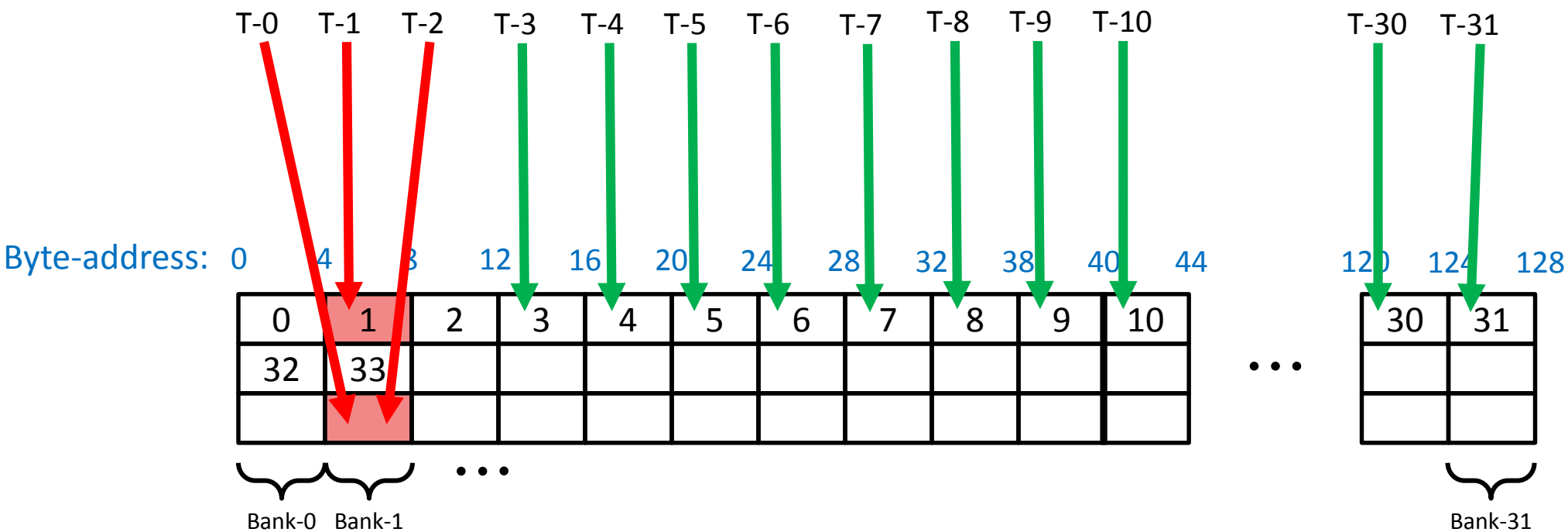
No Bank Conflicts



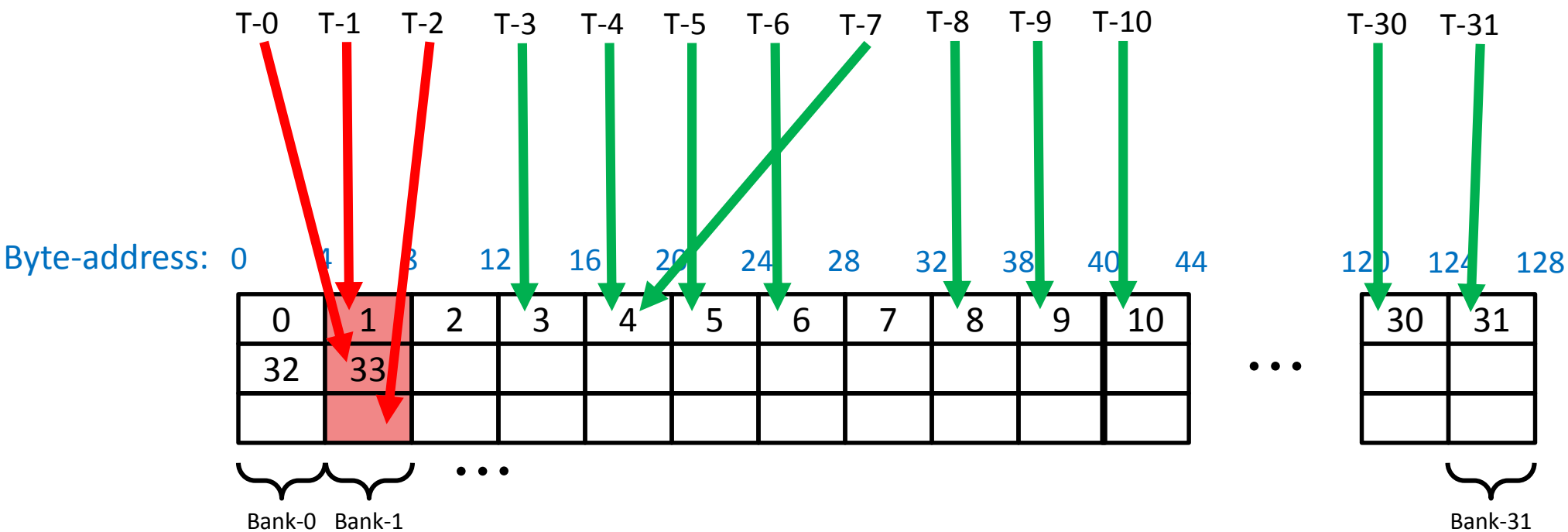
2-way Bank Conflict



2-way Bank Conflict



3-way Bank Conflict



Bank Conflict Resolution

4B or smaller words:

- Process addresses of all threads in a warp in a single phase

8B words are accessed in 2 phases:

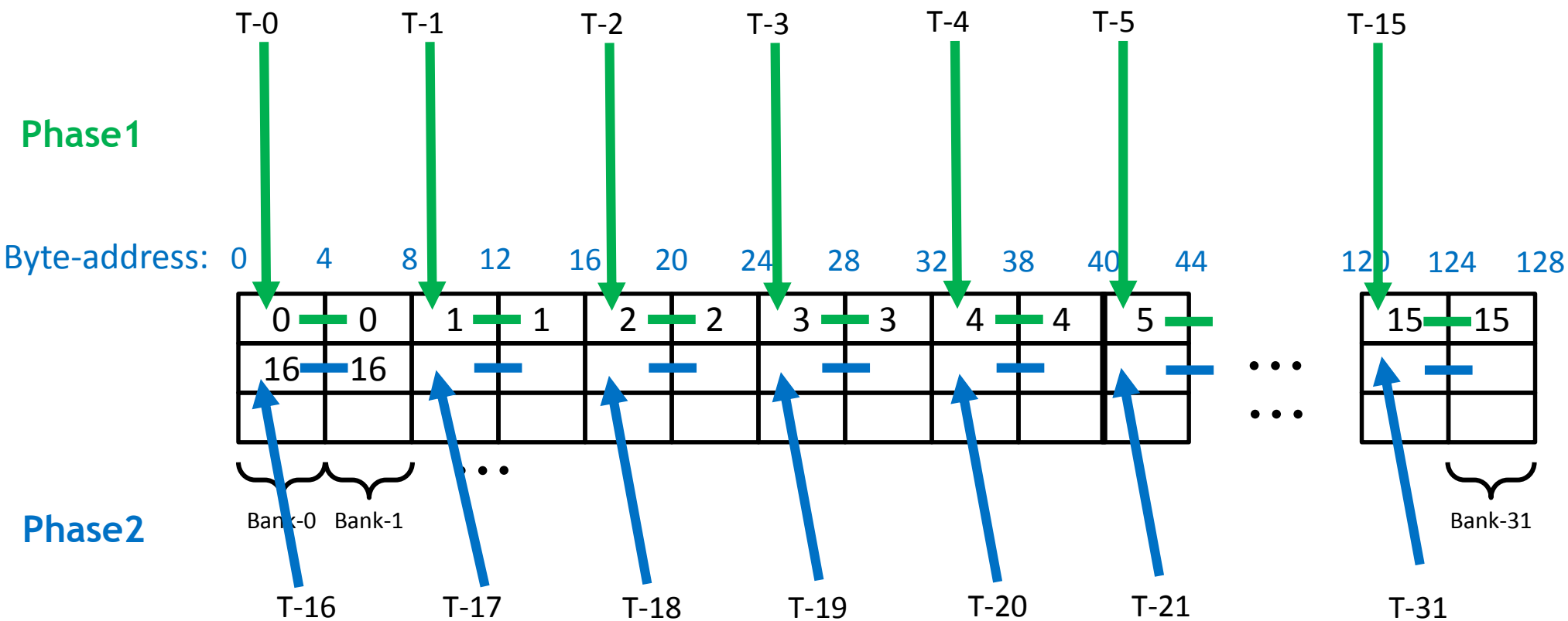
- Process addresses of the **first 16** threads in a warp
- Process addresses of the **second 16** threads in a warp

16B words are accessed in 4 phases:

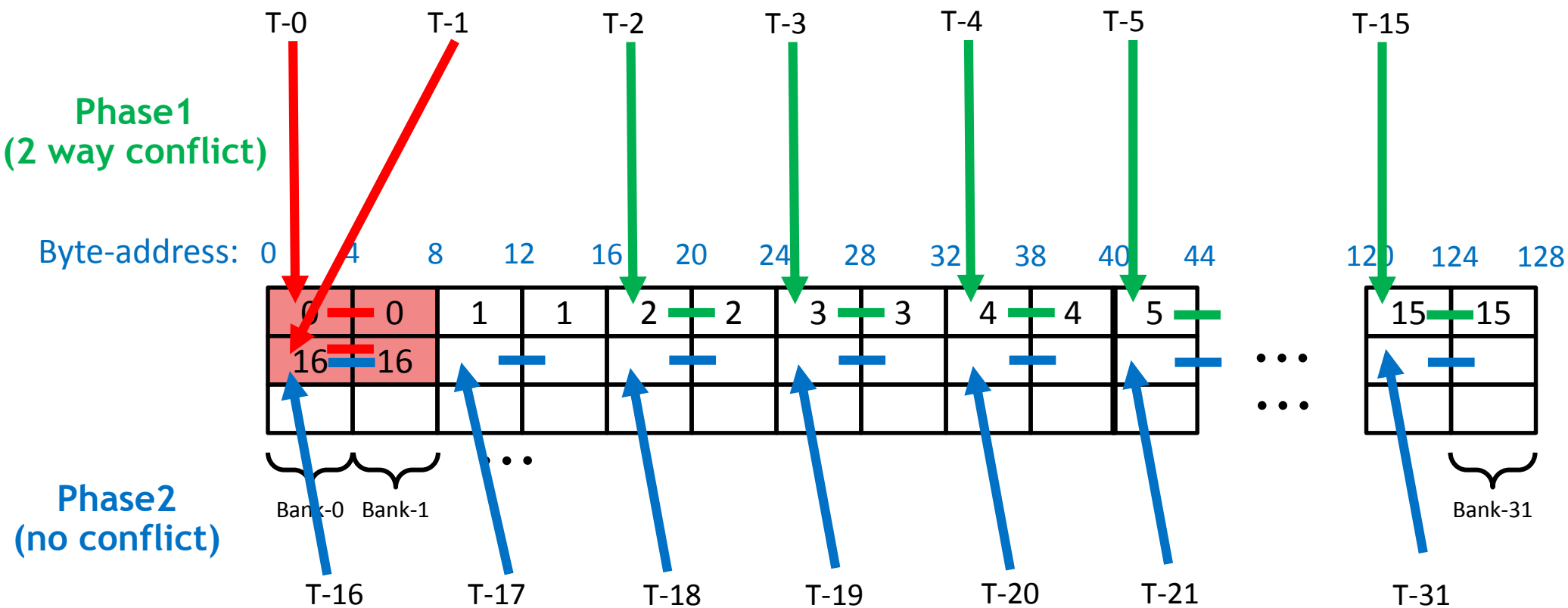
- Each phase processes a quarter of a warp

Bank conflicts occur only between threads in the same phase

8B words, No Conflicts



8B words, 2-way Conflict



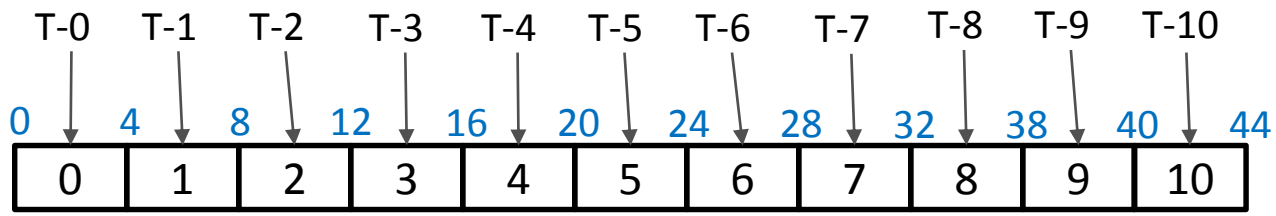
Constant Memory

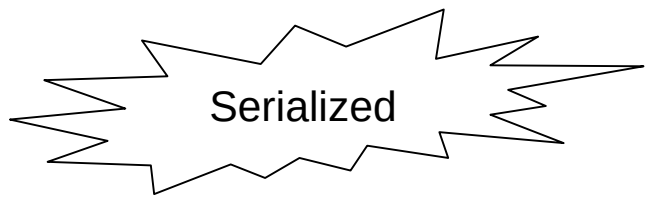
- Globally-scoped arrays qualified with **__constant__**
- Total constant data size limited to **64 KB**
- Throughput = 4B per clock per SM (ideal if entire warp reads the same address)
- Can be used directly in arithmetic instructions (saving registers)
- Example use : Stencil coefficients

Constant Memory

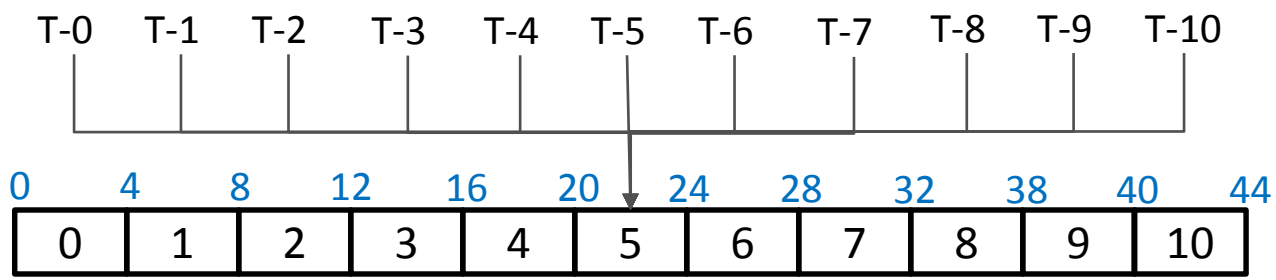
```
__constant__ int array[1024];
```

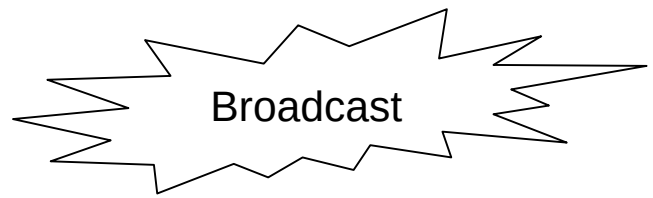
```
int i = array[threadIdx.x];
```






```
int i = array[5];
```





GPU Registers

- Scalar Variables are usually placed in Registers
- Total Register Size 256KB per SM
- **Zero** latency
- Example use : Manually unrolling loop/register rotation
- PTX info shows how many registers are used for each kernel
  - nvcc –Xptxas=-v sample.cu

```
ptxas info    : 26 bytes gmem
ptxas info    : Compiling entry function '_Z9vectoraddPfS_S_' for 'sm_60'
ptxas info    : Function properties for _Z9vectoraddPfS_S_
    32 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 20 registers, 344 bytes cmem[0]
```

AGENDA

Volta V100 Overview

GPU Programming Models: CUDA and OpenACC

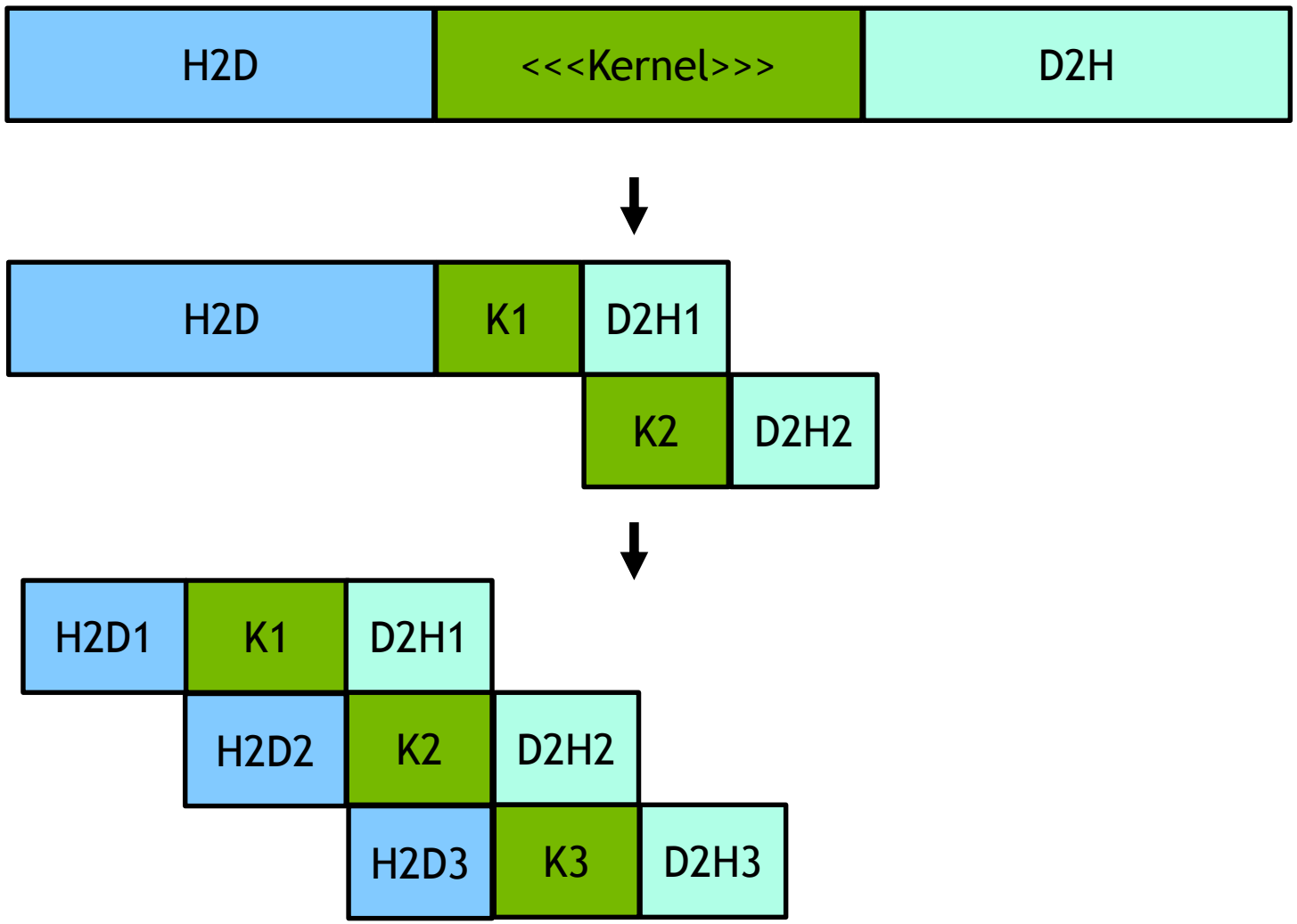
CUDA Optimization Tips

Asynchronous Execution: Stream and Concurrency

Multiple GPUs Programming

New Features in Volta GPU

Concurrency: Overlapping Computing and Data Transfer



CUDA Stream

A stream is queue created and destroyed by:

```
cudaStream_t stream1;
cudaError_t result;

result = cudaStreamCreate(&stream1)

result = cudaStreamDestroy(stream1)
```

Stream is the 4th launch parameter

— **kernel<<< blocks , threads, smem, stream1>>>();**

Stream is passed into some API calls

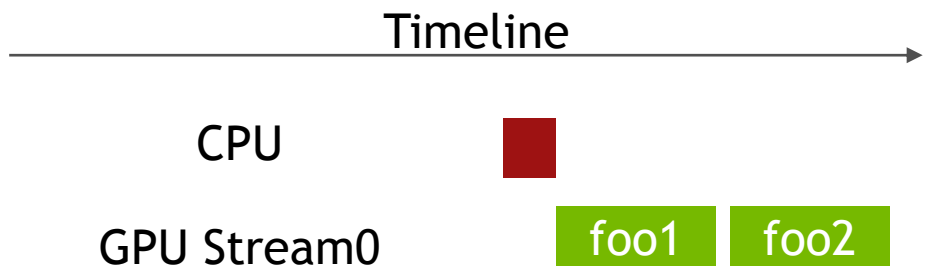
— **cudaMemcpyAsync(dst, src, size, dir, stream1);**

Kernels Concurrency

Assume both foo1 and foo2 utilize 50% of GPU resource

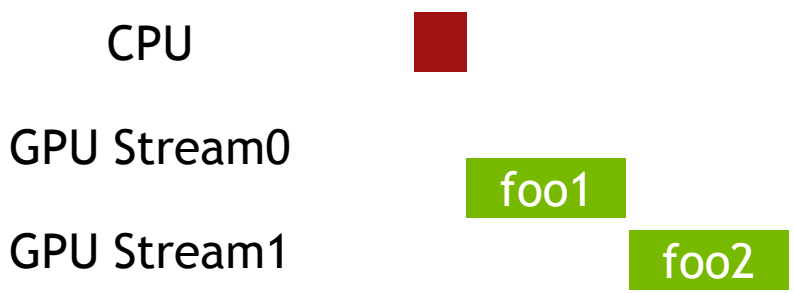
Default Stream

```
... //CPU code

foo1<<<blocks,threads>>>();

foo2<<<blocks,threads>>>();
```



Default & User Streams

```
cudaStream_t stream1;
cudaStreamCreate(&stream1);
foo1<<<blocks,threads>>>();
foo2<<<blocks,threads,0,stream1>>>();
cudaStreamDestroy(stream1);
```

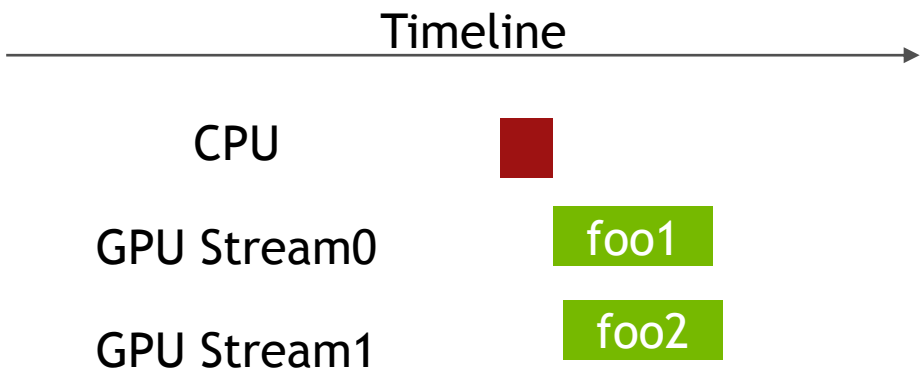


Kernels Concurrency

Assume both foo1 and foo2 utilize 50% of GPU resource

Default Stream & User Stream

```
cudaStream_t stream1;
cudaStreamCreateWithFlags(&stream1,
                          cudaStreamNonBlocking);
foo<<<blocks,threads>>>();
foo<<<blocks,threads,0,stream1>>>();
cudaStreamDestroy(stream1);
```



AGENDA

Volta V100 Overview

GPU Programming Models: CUDA and OpenACC

CUDA Optimization Tips

Asynchronous Execution: Stream and Concurrency

Multiple GPUs Programming

New Features in Volta GPU

Motivation

Why use multiple GPUs?

Need to compute larger, e.g. bigger networks, car models, …

Need to compute faster, e.g. weather prediction

Better energy efficiency with dense nodes with multiple GPUs

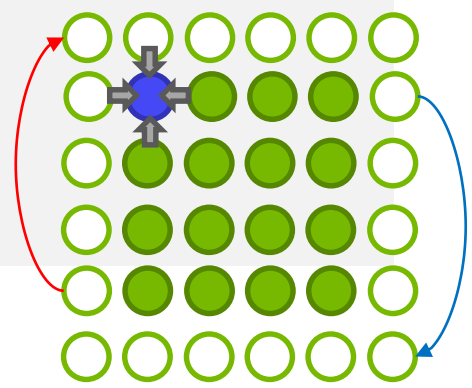
Example: Jacobi solver

Single GPU

While not converged

Do Jacobi step:

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



Apply periodic boundary conditions

Swap `a_new` and `a`

Next iteration

Example: Jacobi solver

Multi GPU

While not converged

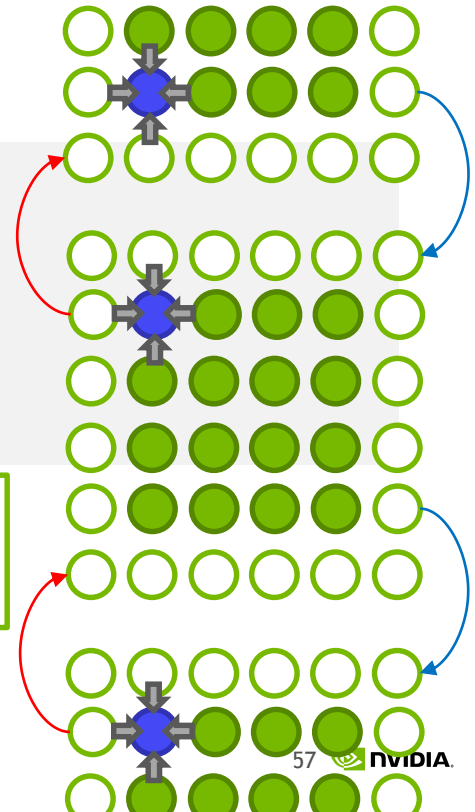
Do Jacobi step:

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

Apply periodic boundary conditions

Exchange halo with 2 neighbors

One-step with ring exchange



Swap `a_new` and `a`

Next iteration

single Threaded Multi GPU Programming

```
while ( l2_norm > tol && iter < iter_max ) {
    for ( int dev_id = 0; dev_id < num_devices; ++dev_id ) {
        const int top = dev_id > 0 ? dev_id - 1 : (num_devices-1); const int bottom = (dev_id+1)%num_devices;
        cudaSetDevice( dev_id );
        cudaMemsetAsync(l2_norm_d[dev_id], 0 , sizeof(real) );
        jacobi_kernel<<<dim_grid,dim_block>>>( a_new[dev_id], a[dev_id], l2_norm_d[dev_id],
                                               iy_start[dev_id], iy_end[dev_id], nx );
        cudaMemcpyAsync( l2_norm_h[dev_id], l2_norm_d[dev_id], sizeof(real), cudaMemcpyDeviceToHost );
        cudaMemcpyAsync( a_new[top]+(iy_end[top]*nx), a_new[dev_id]+iy_start[dev_id]*nx, nx*sizeof(real), ...);
        cudaMemcpyAsync( a_new[bottom], a_new[dev_id]+(iy_end[dev_id]-1)*nx, nx*sizeof(real), ...);
    }
    l2_norm = 0.0;
    for ( int dev_id = 0; dev_id < num_devices; ++dev_id ) {
        cudaSetDevice( dev_id ); cudaDeviceSynchronize();
        l2_norm += *(l2_norm_h[dev_id]);
    }
    l2_norm = std::sqrt( l2_norm );
    for ( int dev_id = 0; dev_id < num_devices; ++dev_id ) std::swap(a_new[dev_id],a[dev_id]);
    iter++;
}
```

Scalabilty metrics for success

Serial Time: T_s: How long it takes to run the problem with a single process

Parallel Time: T_p: How long it takes to run the problem with multiple processes

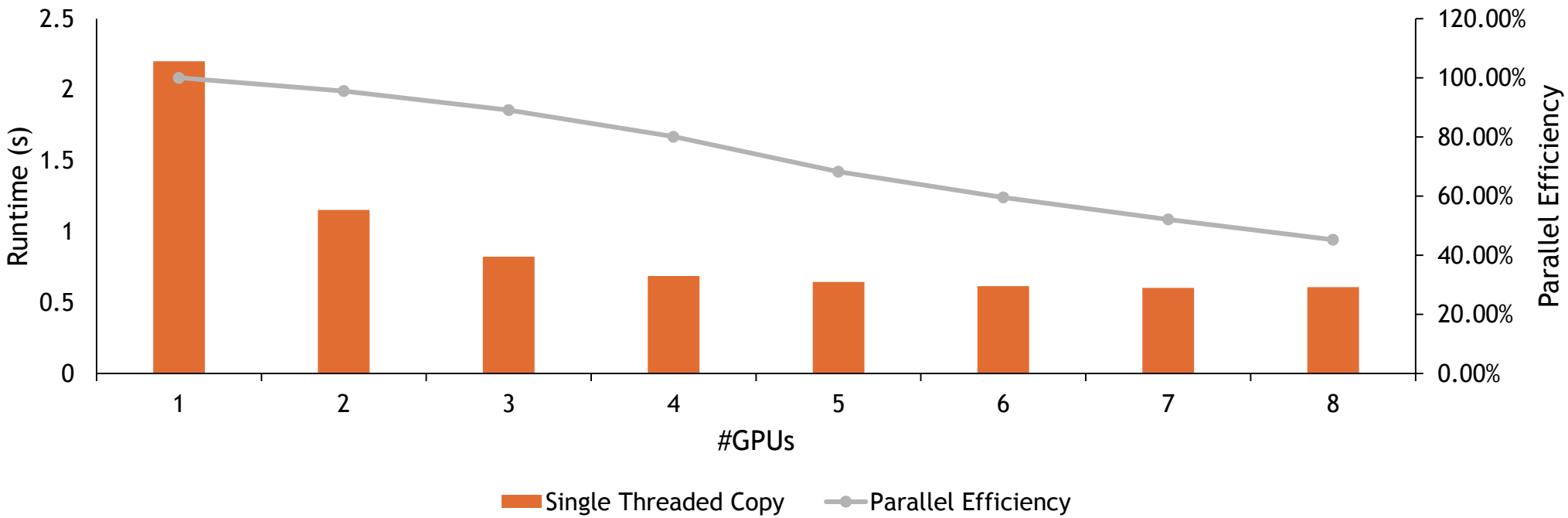
Number of Processes: P: The number of Processes operating on the task at hand

Speedup: $S = \frac{T_s}{T_p}$: How much faster is the parallel version vs. serial. (optimal is P)

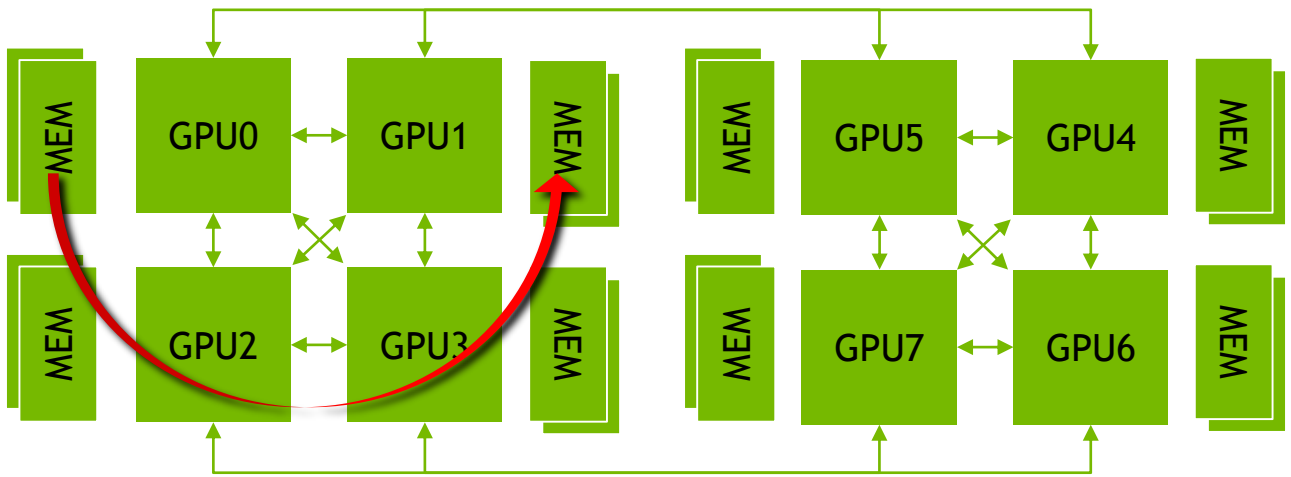
Efficiency: $E = \frac{S}{P}$: How efficient are the processors used (optimal is 1)

Multi GPU Jacobi Runtime

DGX1 - 1024 x 1024, 1000 iterations



GPUDirect P2P



Maximizes intra node inter GPU Bandwidth

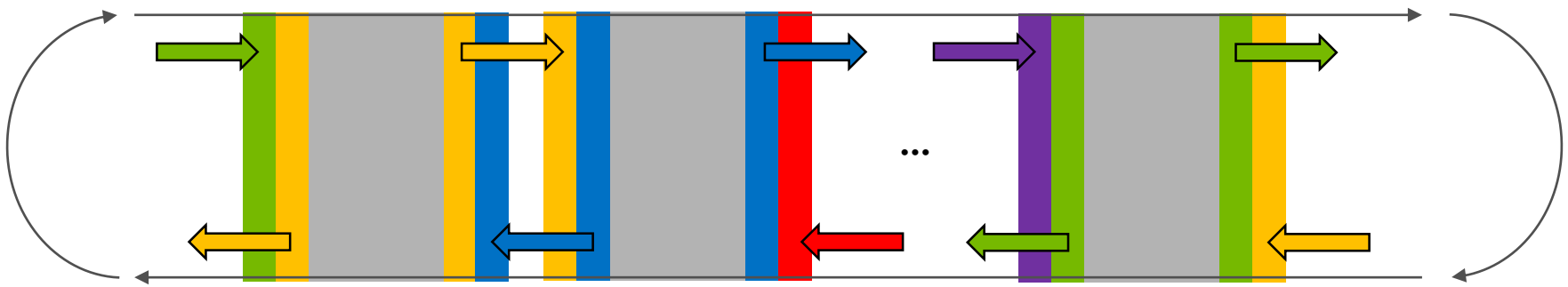
Avoids Host memory and system topology bottlenecks

GPUDirect P2P

Enable P2P

```
for ( int dev_id = 0; dev_id < num_devices; ++dev_id ) {
    cudaSetDevice( dev_id );
    const int top = dev_id > 0 ? dev_id - 1 : (num_devices-1);
    int canAccessPeer = 0;
    cudaDeviceCanAccessPeer ( &canAccessPeer, dev_id, top );
    if ( canAccessPeer )
        cudaDeviceEnablePeerAccess ( top, 0 );
    const int bottom = (dev_id+1)%num_devices;
    if ( top != bottom ) {
        cudaDeviceCanAccessPeer ( &canAccessPeer, dev_id, bottom );
        if ( canAccessPeer )
           cudaDeviceEnablePeerAccess ( bottom, 0 );
    }
}
```

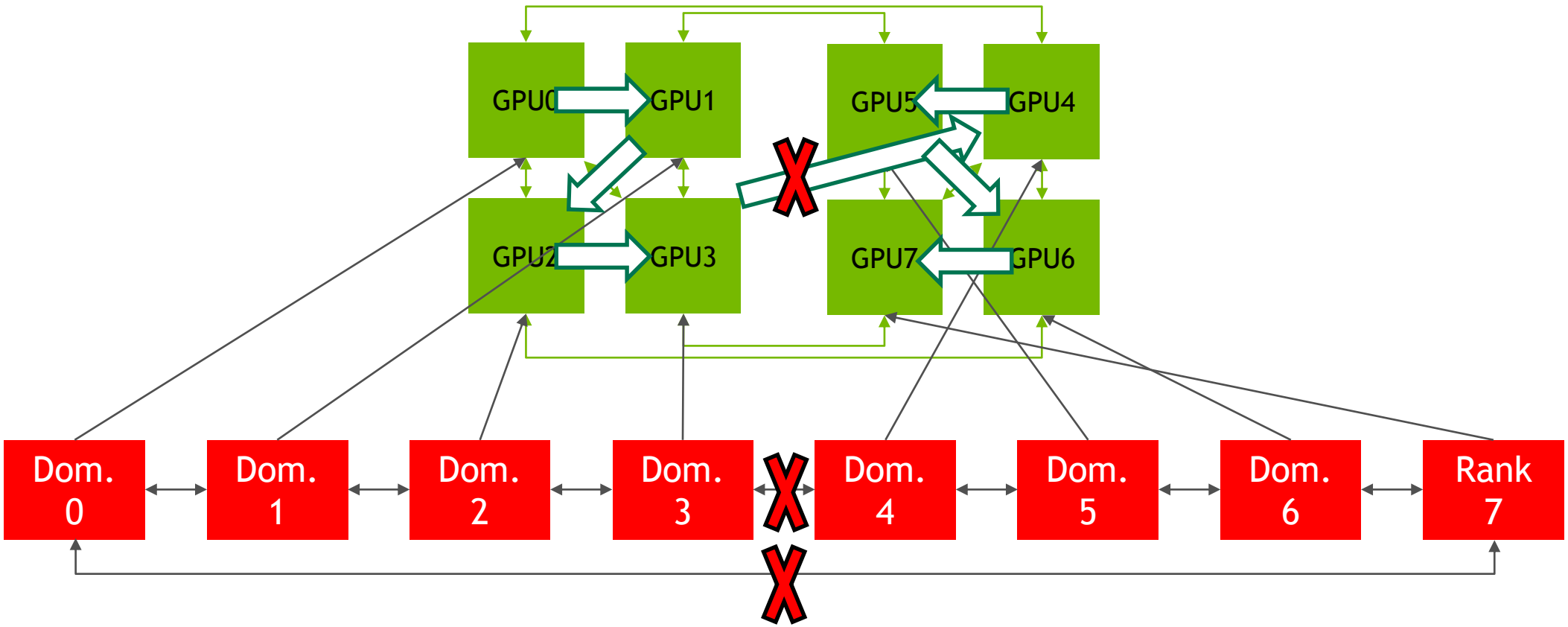
1D Ring exchange



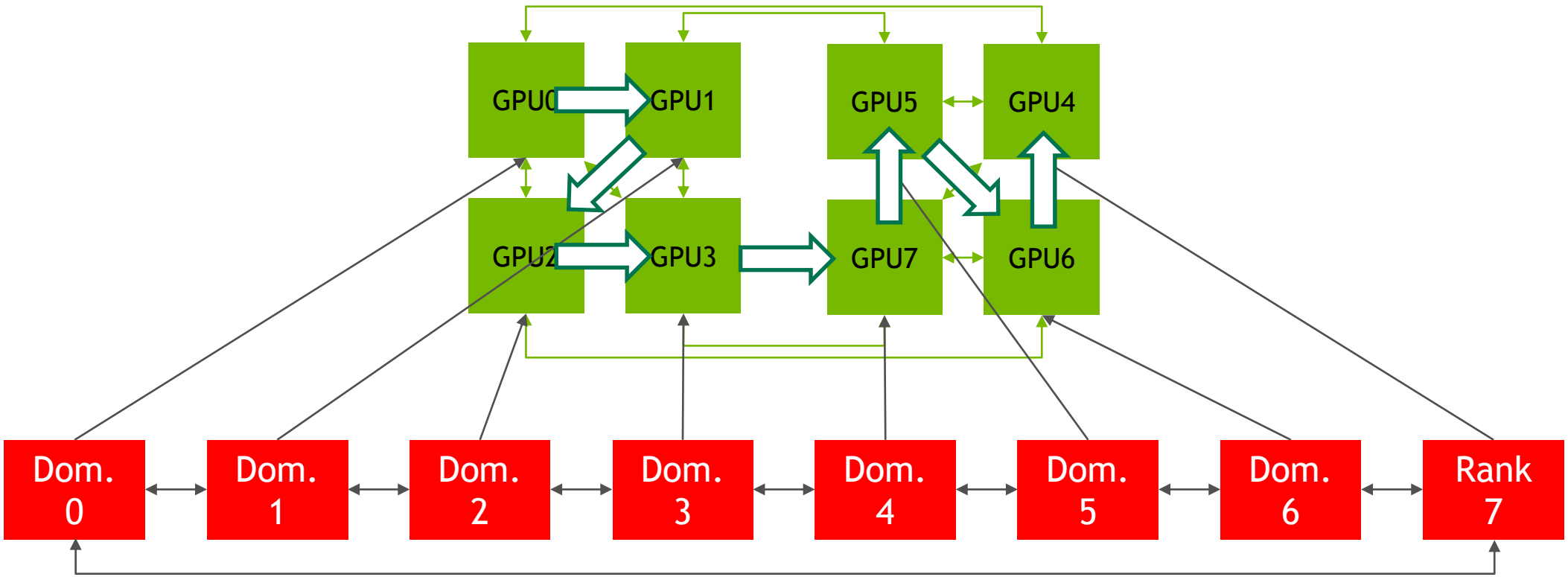
Halo updates for 1D domain decomposition with periodic boundary conditions

Unidirectional rings are important building block for collective algorithms

Mapping 1D Ring Exchange to DGX-1



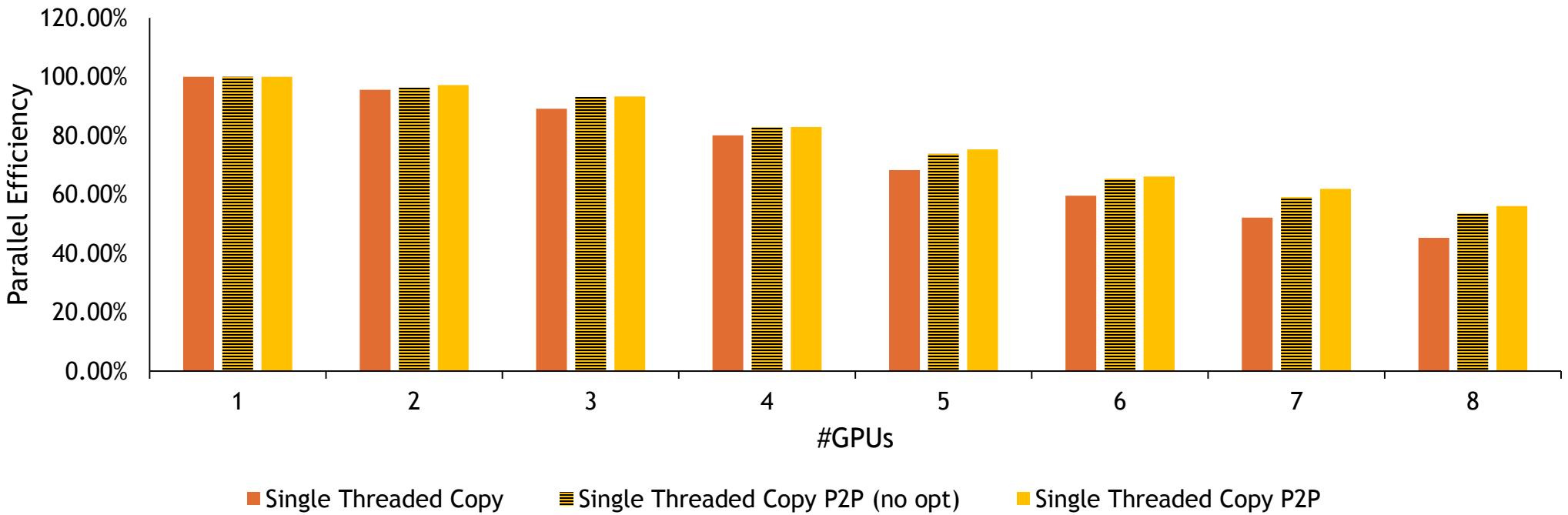
Mapping 1D Ring Exchange to DGX-1




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

Multi GPU Jacobi Runtime

DGX1 - 1024 x 1024, 1000 iterations



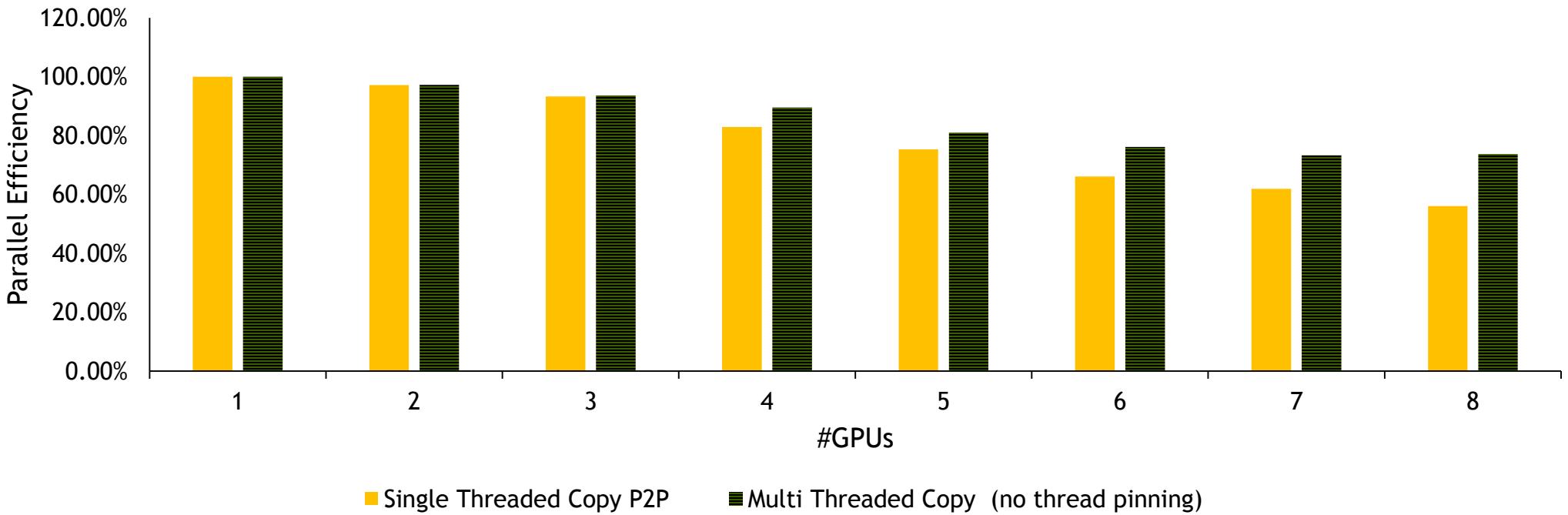
Multi Threaded Multi GPU Programming

Using OpenMP

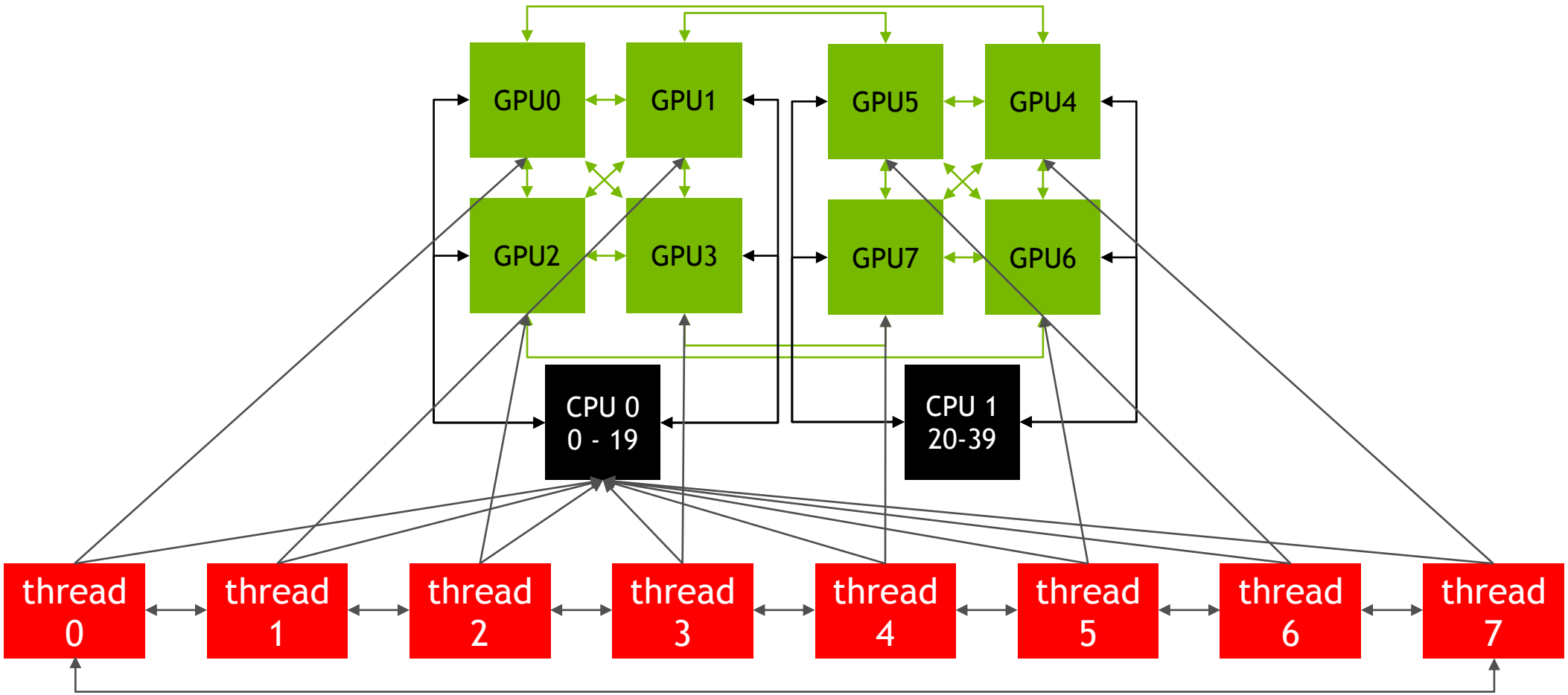
```
int num_devices = 0;

cudaGetDeviceCount( &num_devices );

#pragma omp parallel num_threads( num_devices )

{

    int dev_id = omp_get_thread_num();

    cudaSetDevice( dev_id );

}
```

Multi GPU Jacobi Runtime

DGX1 - 1024 x 1024, 1000 iterations

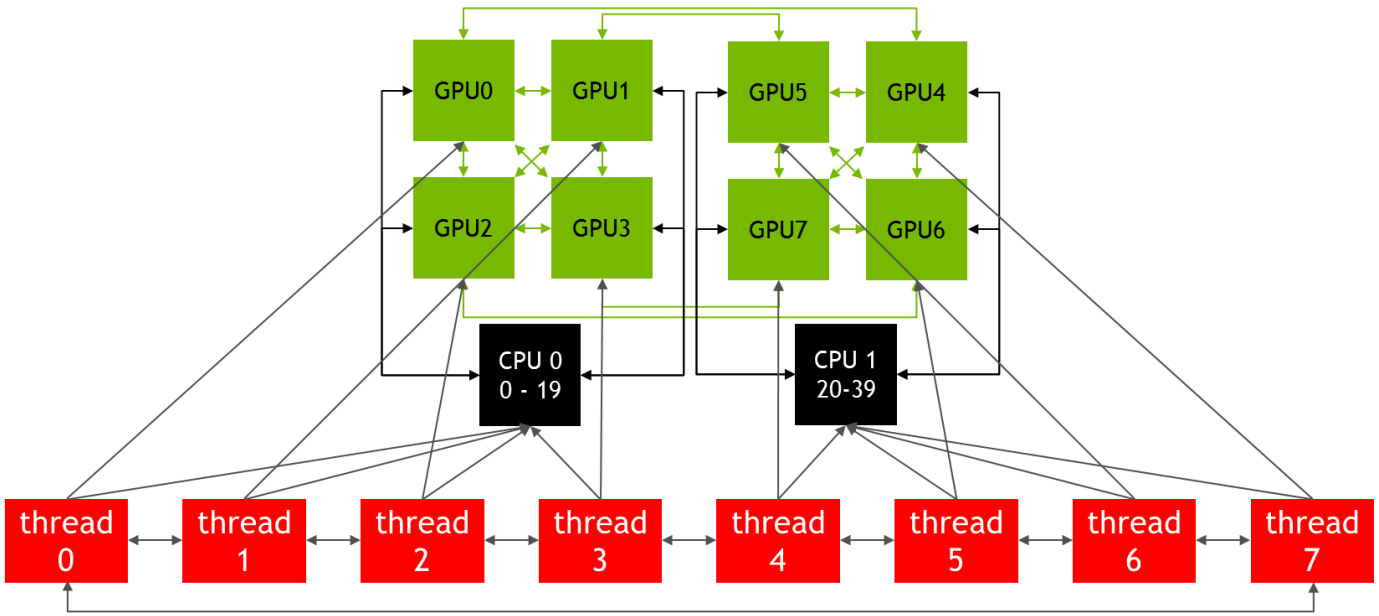


GPU/CPU Affinity



GPU/CPU Affinity

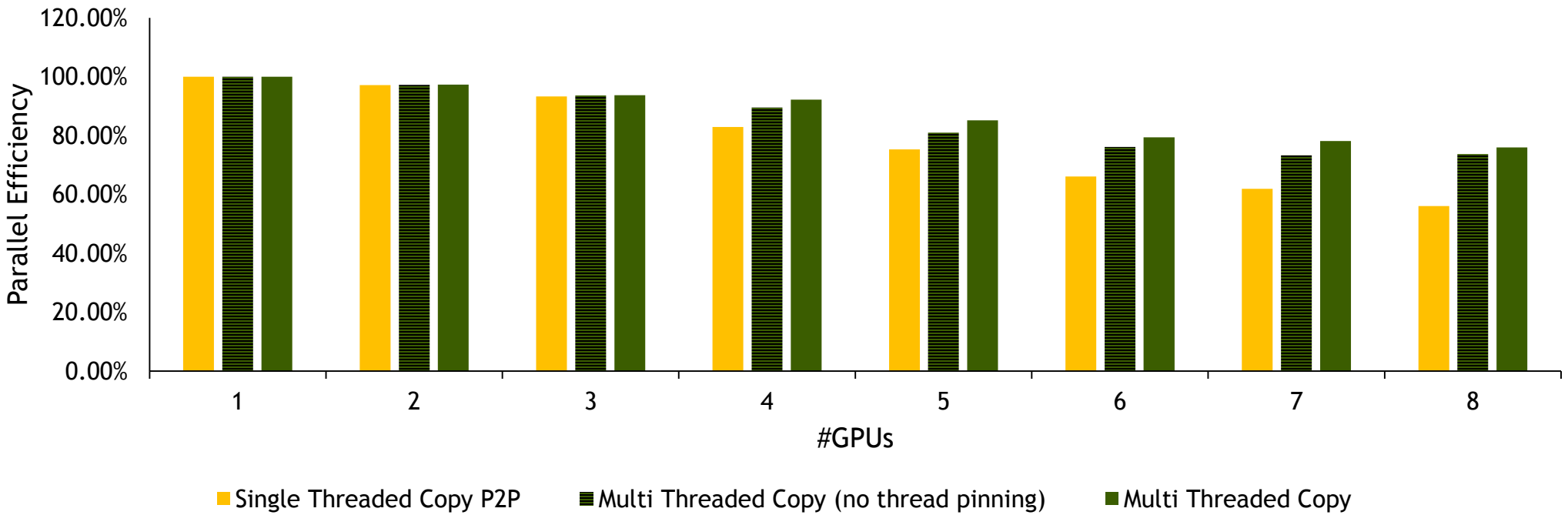
Using CUDA_VISIBLE_DEVICES and OpenMP env. vars.



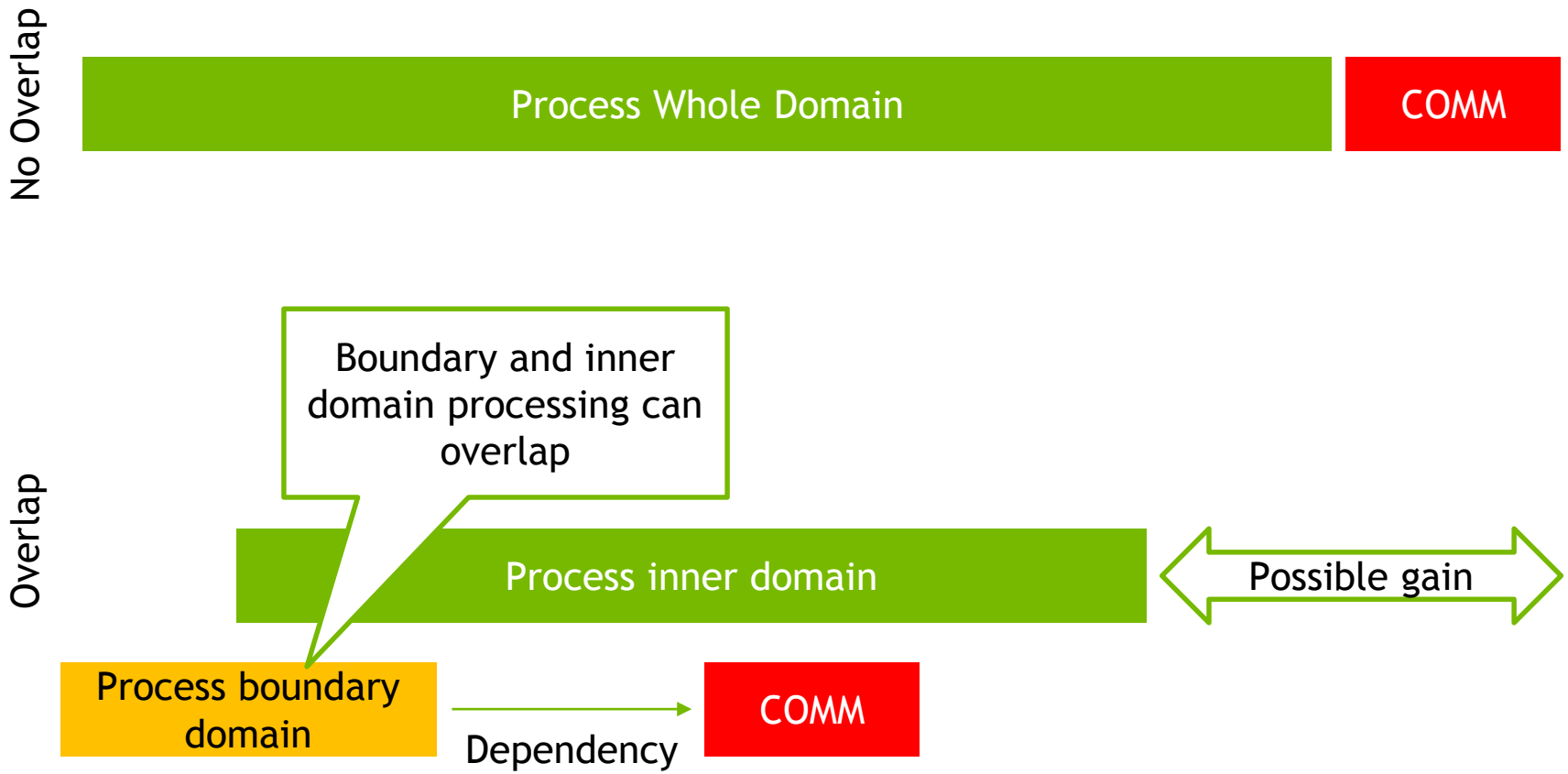

```
export OMP_PROC_BIND=TRUE
export CUDA_VISIBLE_DEVICES="0,1,2,3,7,6,5,4"
export OMP_PLACES="{0},{1},{2},{3},{20},{21},{22},{23}"
```

Multi GPU Jacobi Runtime

DGX1 - 1024 x 1024, 1000 iterations



communication + computation overlap

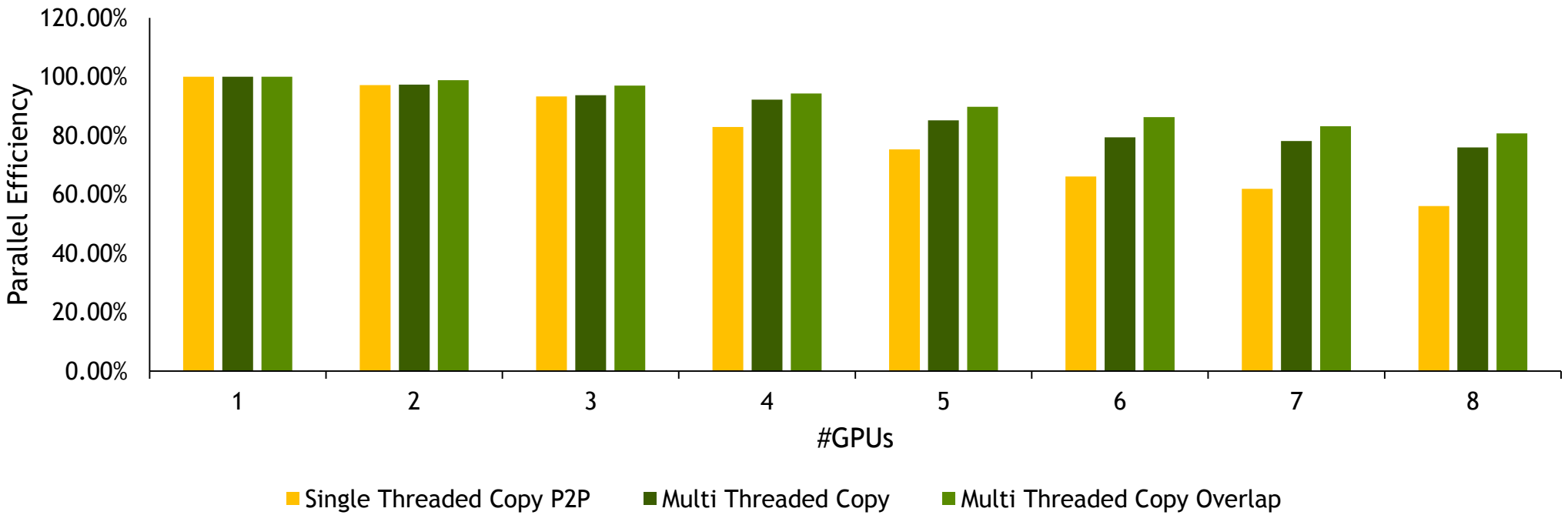


communication + computation overlap

```
//Compute bulk
cudaStreamWaitEvent(compute_stream,push_top_done[(iter%2)][dev_id],0);
cudaStreamWaitEvent(compute_stream,push_bottom_done[(iter%2)][dev_id],0);
jacobi_kernel<<<dim_grid,dim_block,0,compute_stream>>>(a_new[dev_id],a,l2_norm_d,(iy_start+1),(iy_end[dev_id]-1),nx);

//Compute boundaries
cudaStreamWaitEvent( push_top_stream, reset_l2norm_done, 0 );
cudaStreamWaitEvent( push_top_stream, push_bottom_done[(iter%2)][top], 0 );
jacobi_kernel<<<nx/128+1,128,0,push_top_stream>>>( a_new[dev_id],a,l2_norm_d,iy_start,(iy_start+1),nx);
cudaStreamWaitEvent(push_bottom_stream,reset_l2norm_done,0);
cudaStreamWaitEvent(push_bottom_stream,push_top_done[(iter%2)][bottom], 0 ) ;
jacobi_kernel<<<nx/128+1,128,0,push_bottom_stream>>>( a_new[dev_id],a,l2_norm_d,(iy_end[dev_id]-1),iy_end[dev_id],nx);

//Apply periodic boundary conditions and exchange halo
cudaMemcpyAsync(a_new[top]+(iy_end[top]*nx),a_new[dev_id]+iy_start*nx,nx*sizeof(real),cudaMemcpyDeviceToDevice,push_top_stream);
cudaEventRecord(push_top_done[((iter+1)%2)][dev_id],push_top_stream);
cudaMemcpyAsync(a_new[bottom],a_new[dev_id]+(iy_end[dev_id]-1)*nx,nx*sizeof(real),cudaMemcpyDeviceToDevice,push_bottom_stream);
cudaEventRecord(push_bottom_done[((iter+1)%2)][dev_id],push_bottom_stream);
```

Multi GPU Jacobi Runtime

DGX1 - 1024 x 1024, 1000 iterations



Message Passing Interface - MPI

- Standard to exchange data between processes via messages
  - Defines API to exchanges messages
    - Point to Point: e.g. `MPI_Send`, `MPI_Recv`
    - Collectives: e.g. `MPI_Reduce`
- Multiple implementations (open source and commercial)
  - Bindings for C/C++, Fortran, Python, …
  - E.g. MPICH, OpenMPI, MVAPICH, IBM Platform MPI, Cray MPT, …

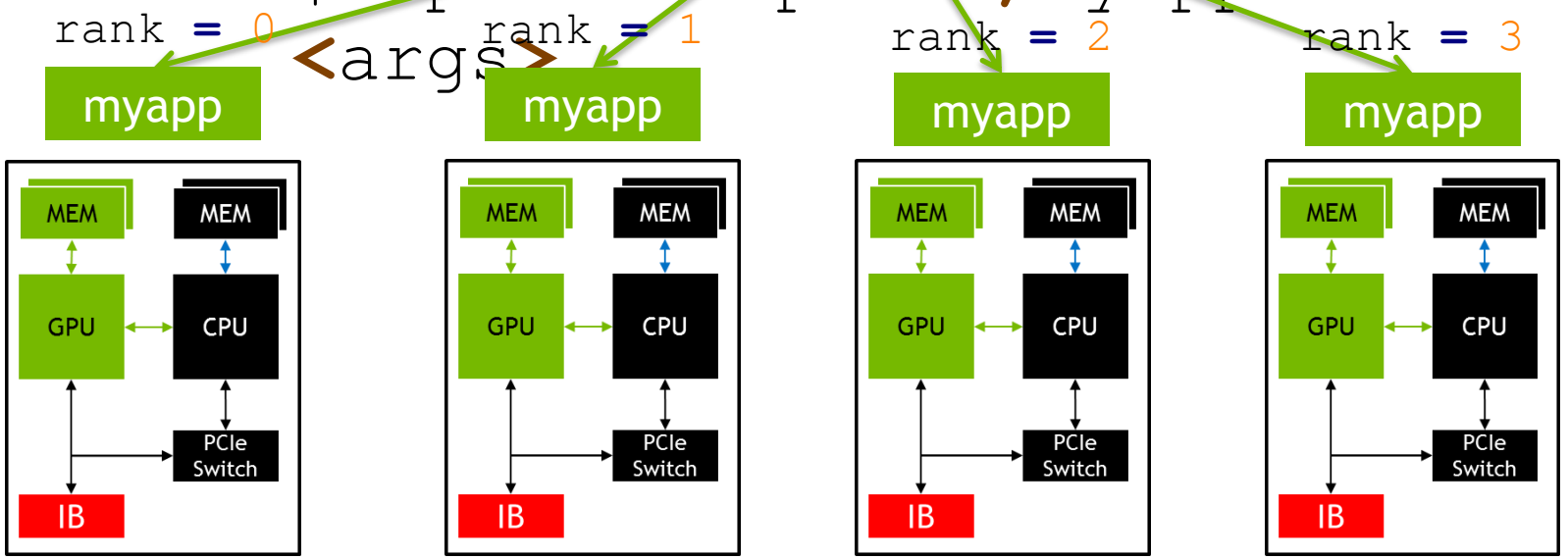
MPI - Skeleton

```c
#include <mpi.h>
int main(int argc, char *argv[]) {
    int rank,size;
    /* Initialize the MPI library */
    MPI_Init(&argc,&argv);
    /* Determine the calling process rank and total number of ranks */
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    /* Call MPI routines like MPI_Send, MPI_Recv, ... */
    ...
    /* Shutdown MPI library */
    MPI_Finalize();
    return 0;
}
```

MPI

Compiling and Launching

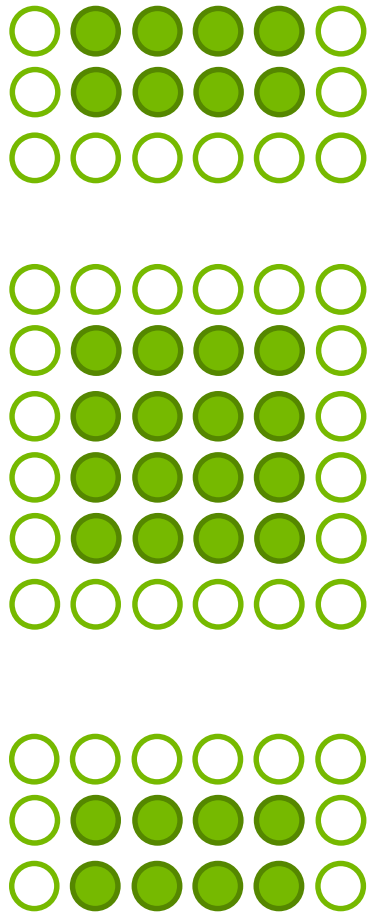
```
$ mpicc -o myapp myapp.c
$ mpirun -np 4 ./myapp <args>
```



Example Jacobi

Top/Bottom Halo

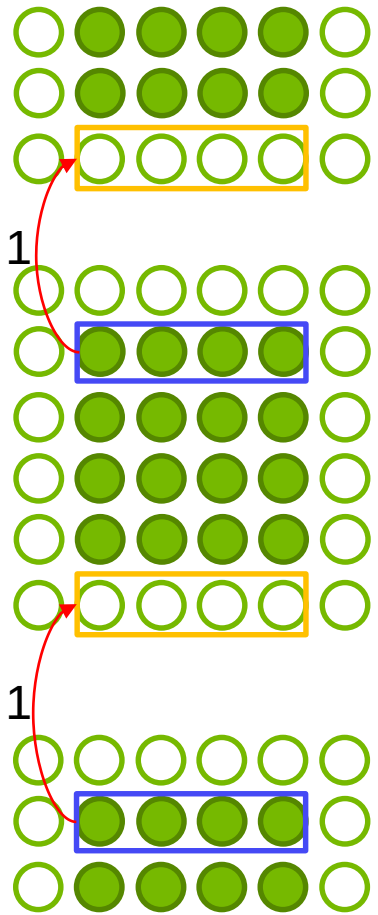
```
MPI_Sendrecv(a_new+iy_start*nx, nx, MPI_FLOAT, top , 0,
             a_new+(iy_end*nx), nx, MPI_FLOAT, bottom, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```



Example Jacobi

Top/Bottom Halo

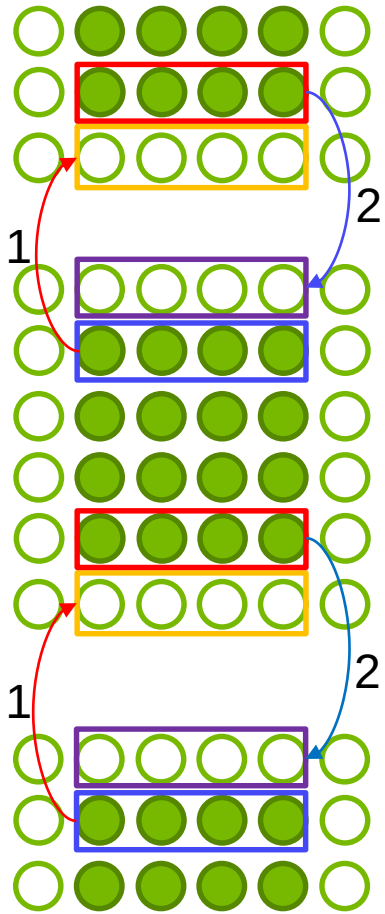
```
MPI_Sendrecv(a_new+iy_start*nx, nx, MPI_FLOAT, top , 0,
             a_new+(iy_end*nx), nx, MPI_FLOAT, bottom, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```



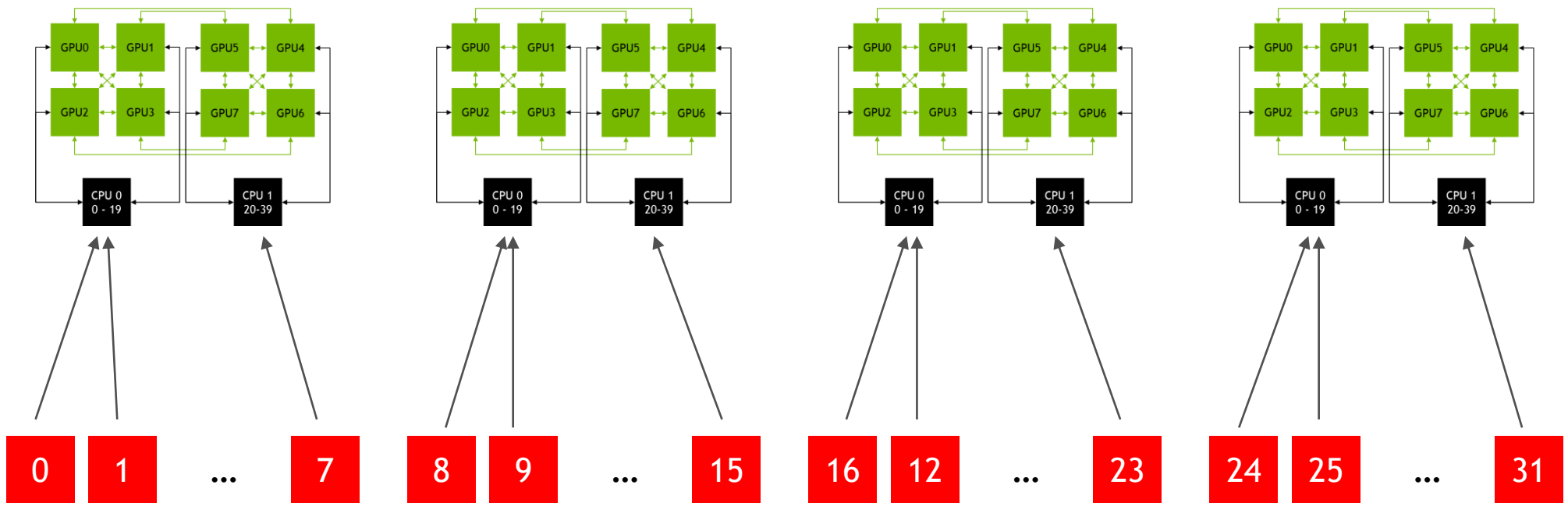
Example Jacobi

Top/Bottom Halo

```
MPI_Sendrecv(a_new+iy_start*nx, nx, MPI_FLOAT, top , 0,
             a_new+(iy_end*nx), nx, MPI_FLOAT, bottom, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);

MPI_Sendrecv(a_new+(iy_end-1)*nx, nx, MPI_FLOAT, bottom, 0,
             a_new, nx, MPI_FLOAT, top, 0, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
```



Handling Multiple Multi GPU nodes

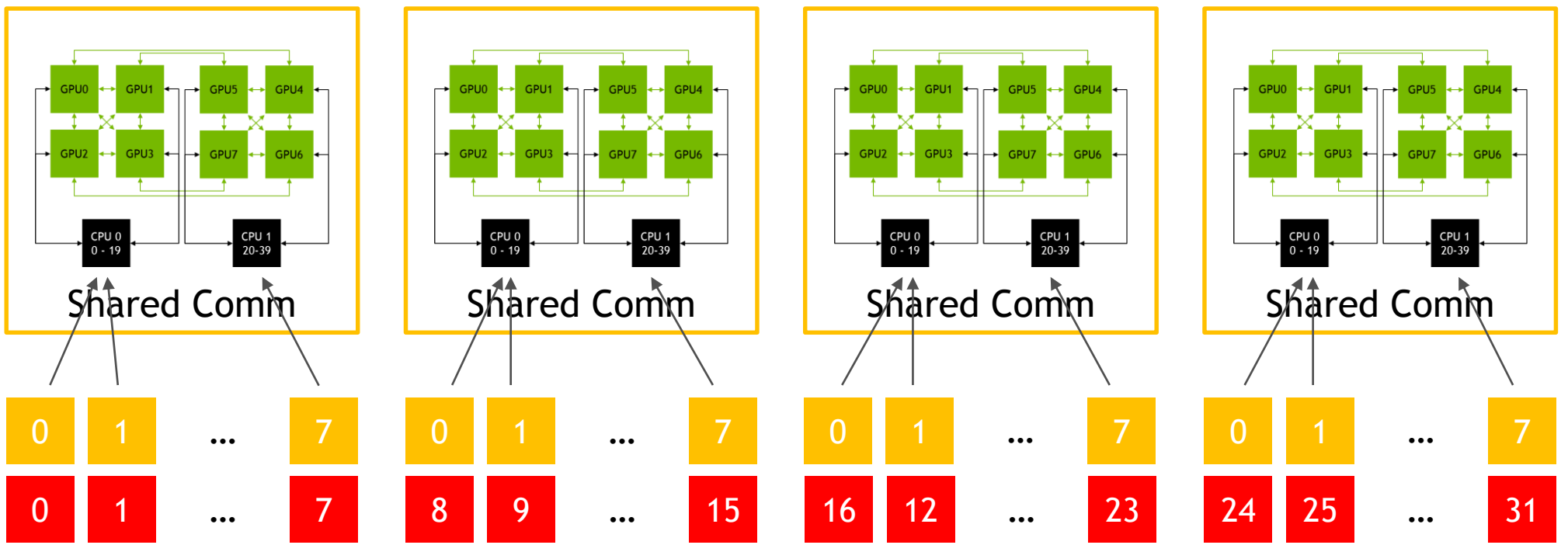


Handling Multiple Multi GPU nodes

How to determine the local rank? – MPI-3

```
MPI_Comm local_comm;

MPI_Info info;

MPI_Info_create(&info);

MPI_Comm_split_type(MPI_COMM_WORLD, MPI_COMM_TYPE_SHARED, rank, info, &local_comm);

int local_rank = -1;

MPI_Comm_rank(local_comm,&local_rank);

MPI_Comm_free(&local_comm);

MPI_Info_free(&info);
```

Handling Multiple Multi GPU nodes



Handling Multiple Multi GPU nodes

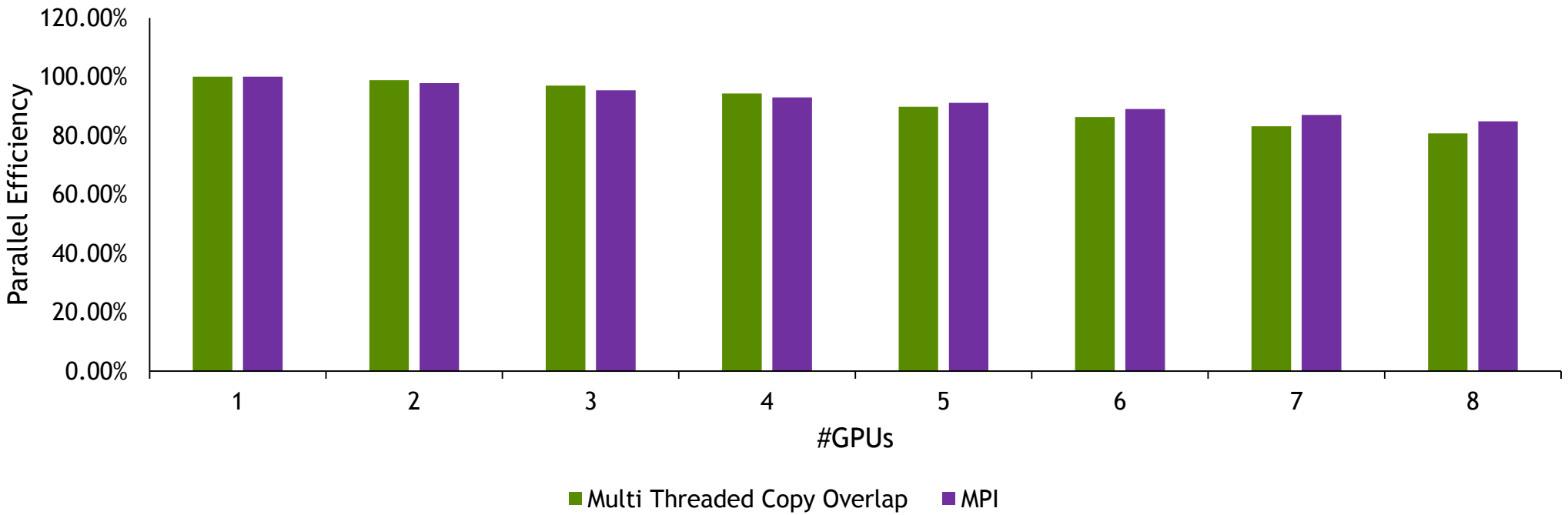
GPU-affinity

Use local rank:

```
int local_rank = -1;

MPI_Comm_rank(local_comm,&local_rank);

int num_devices = 0;

cudaGetDeviceCount(&num_devices);

cudaSetDevice(local_rank % num_devices);
```

Multi GPU Jacobi Runtime

DGX1 - 1024 x 1024, 1000 iterations

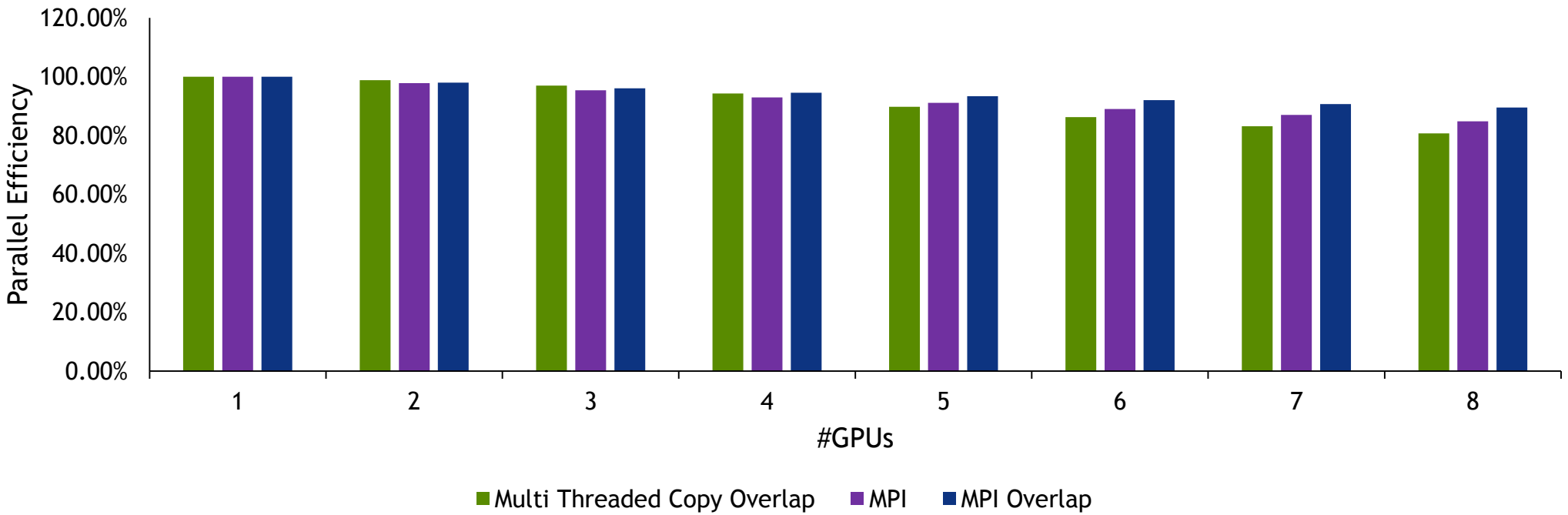


communication + computation overlap

```
launch_jacobi_kernel( a_new, a, l2_norm_d, iy_start, (iy_start+1), nx, push_top_stream );
launch_jacobi_kernel( a_new, a, l2_norm_d, (iy_end-1), iy_end, nx, push_bottom_stream );
launch_jacobi_kernel( a_new, a, l2_norm_d, (iy_start+1), (iy_end-1), nx, compute_stream );
const int top = rank > 0 ? rank - 1 : (size-1);
const int bottom = (rank+1)%size;
cudaStreamSynchronize( push_top_stream );
MPI_Sendrecv( a_new+iy_start*nx, nx, MPI_REAL_TYPE, top , 0,
              a_new+(iy_end*nx), nx, MPI_REAL_TYPE, bottom, 0,
              MPI_COMM_WORLD, MPI_STATUS_IGNORE );
cudaStreamSynchronize( push_bottom_stream );
MPI_Sendrecv( a_new+(iy_end-1)*nx, nx, MPI_REAL_TYPE, bottom, 0,
              a_new, nx, MPI_REAL_TYPE, top, 0, MPI_COMM_WORLD,
              MPI_STATUS_IGNORE );
```

Multi GPU Jacobi Runtime

DGX1 - 1024 x 1024, 1000 iterations



Summary

	Programming Models	GPUDirect P2P	Multi Node
Single Threaded	CUDA	Improves Perf.	No
Multi Threaded	CUDA + OpenMP/TBB/…	Improves Perf.	No
Multi Threaded P2P	CUDA + OpenMP/TBB/…	Required	No
MPI	CUDA + MPI	Improves Perf.	Yes

AGENDA

Volta V100 Overview

GPU Programming Models: CUDA and OpenACC

CUDA Optimization Tips

Asynchronous Execution: Stream and Concurrency

Multiple GPUs Programming

New Features in Volta GPU

Volta Cooperative Group

New in Volta Architecture

- **Program counter:**
  Before Volta: Per warp
  Volta: Per thread
- Volta **guarantees Forward Progress** for diverged threads in a warp
- Allows to exchange data between diverged threads in a warp. E.g. mutexes among warp threads.

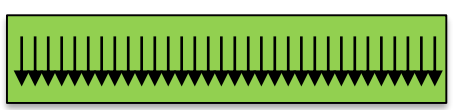
Program defined decomposition

All threads launched

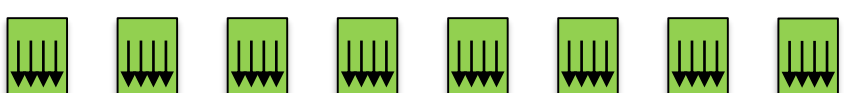
```
thread_block g = this_thread_block();
```

All threads in thread block

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



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



Restricted to powers of two, and <= 32 in initial release

Generic Parallel algorithms

Per-Block

```
g = this_thread_block();
reduce(g, ptr, myVal);
```

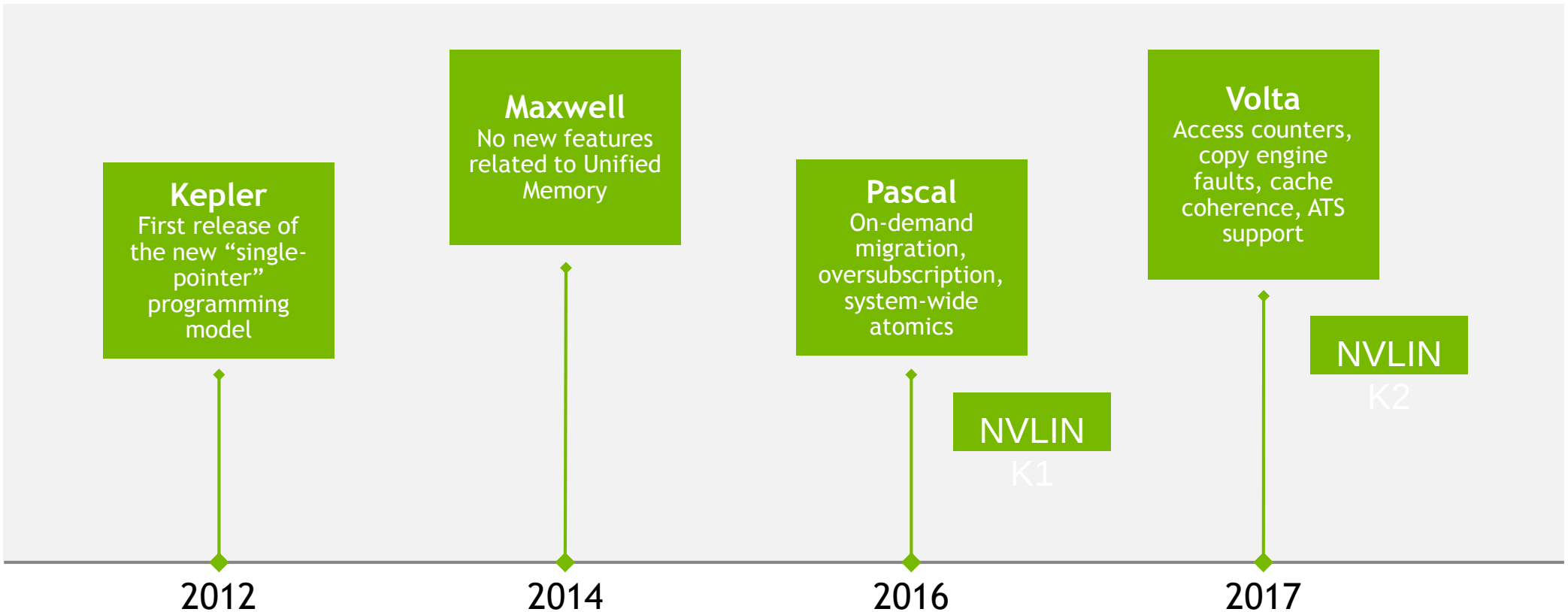
Per-Warp

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

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

UNIFIED MEMORY

Evolution of GPU Architectures



UNIFIED MEMORY FUNDAMENTALS

Deep Copy Nightmare

```
char **data;
data = (char**)malloc(N*sizeof(char*));
for (int i = 0; i < N; i++)
  data[i] = (char*)malloc(N);

char **d_data;
char **h_data = (char**)malloc(N*sizeof(char*));
for (int i = 0; i < N; i++) {
  cudaMalloc(&h_data2[i], N);
  cudaMemcpy(h_data2[i], h_data[i], N, ...);
}
cudaMalloc(&d_data, N*sizeof(char*));
cudaMemcpy(d_data, h_data2, N*sizeof(char*), ...);

gpu_func<<<...>>>(data, N);
```

```
char **data;
data =
(char**)malloc(N*sizeof(char*));
for (int i = 0; i < N; i++)
  data[i] = (char*)malloc(N);

gpu_func<<<...>>>(data, N);
```