# Exercise 6

Introduction to High-Performance Computing

Winter 2019

Julian Miller

Dr. Sandra Wienke

contact@hpc.rwth-aachen.de

# Exercise Tasks

1. **GPU Performance Modeling**
   1. Kernel Execution Time
   2. Data Transfer Time
   3. GPU Runtime
   4. GPU Tuning Impact
2. **GPU Occupancy**
3. **CUDA code & GPU performance tuning**
4. **Jacobi – Parallelize the code with OpenACC**

# GPU Performance Modeling

- **Given system setup as follows:**

| | CPU-based system | GPU-based system |
|---|---|---|
| Processing elements | 2-sockets, 48 cores in total | 80 streaming multiprocessors (SM), 32 (DP) cores per SM |
| Clock frequency | 2.1 GHz | 1.4 GHz |
| Fused multiply add (FMA) | 2 units per core | 1 unit per core |
| Cache sizes | L1 ≙ 32 KB, L2 ≙ 1024 KB, L3 ≙ 33 MB | L1 ≙ 128 KB/SM, L2 ≙ 6 MB/SM |
| Memory sizes | Not given | smem ≙ 128 KB/SM, gmem ≙ 16 GB |
| Memory bandwidth | 170 GB/s | 840 GB/s (gmem) |
| PCIe bandwidth | 16 GB/s | |
| PCIe latencies | H2D ≙ 2 µs, D2H ≙ 3 µs | |

# GPU Performance Modeling

- **Given code snippet as follows:**

```
#define N 100000
double a[N];
double b[N];
double c[N];
double s;

// init a, b, c, and s...

// compute a
for (int i =0; i<N; ++i) {
    a[i] = b[i] + s * c[i] ;
}
```

# 1.1 Kernel Execution Time

- **Code shall be offloaded to the GPU. Determine the potential kernel execution time $t_{kernel}$. In addition, state whether this kernel is compute bound or memory bound on the GPU.**

A $t_{compute} = \frac{arithmetic\ operations\ [Flop]}{P_{peak}}$

B $t_{memory} = \frac{data\ transfers(LOAD,STORE)\ [words]}{b_s}$

$t_{kernel} = \max(t_{compute}, t_{memory})$

- $P_{peak} = 80\ SM \cdot 32\ cores \cdot 1.4\ GHz \cdot 2\frac{Flop}{cycle} = 7{,}168\ GFlop/s$
- $b_s = \frac{840\ GB/s}{8\ B/word} = 105\ Gword/s$
- $t_{compute} = \frac{2{\cdot}N\ Flop}{7{,}168\ GFlop/s} = \frac{200{,}000\ Flop}{7{,}168\ GFlop/s} \approx 27.90 \cdot 10^{-9} s$
- $t_{memory} = \frac{3{\cdot}N\ Words}{105\ Gwords/s} = \frac{300{,}000\ Words}{105\ Gwords/s} \approx 2.857 \cdot 10^{-6} s = 2.86\ \mu s$

$t_{kernel} = 2.86\ \mu s$

- **Code is memory bound on the GPU.**

# 1.2 Data Transfer Time

- **Determine the potential data transfer time $t_{data}$ that is needed to move data between CPU and GPU (and vice verse).**

$$t_{data} = t_{H2D} + t_{D2H} = (a_{H2D} + \frac{data\ transfers(LOAD)}{b_{PCI}}) + (a_{D2H} + \frac{data\ transfers(STORES)}{b_{PCI}})$$

- $b_{PCI} = \frac{16\ GB/s}{8\ B/word} = 2\ Gwords/s$
- $t_{H2D} = 2 \cdot 10^{-6}s + \frac{2 \cdot N\ words}{2\ \frac{Gwords}{s}} = 2 \cdot 10^{-6}s + 100{,}000 \cdot 10^{-9}s = 102\ \mu s$
- $t_{D2H} = 3 \cdot 10^{-6}s + \frac{N\ words}{2\ \frac{Gwords}{s}} = 3 \cdot 10^{-6}s + 50{,}000 \cdot 10^{-9}s = 53\ \mu s$
- $t_{data} = 102\ \mu s + 53\ \mu s = 155\ \mu s$

# 1.3 GPU Runtime

- **Compute the complete GPU runtime $t_{GPU}$. Is it beneficial to run this code on the GPU (in comparison to run it on the CPU)?**

$$t_{GPU} = t_{data} + t_{kernel}$$

- $t_{GPU} = 155\ \mu s + 2.86\ \mu s = 157.86\ \mu s$

- **Here**: $t_{CPU} = t_{memory}$ (must be re-evaluated!)

→ With $b_{s\_CPU} = \frac{170\ GB/s}{8\ B/word} = 21.25\ Gwords/s$

- $t_{CPU} = \frac{3 \cdot N\ words}{21.25\ Gwords/s} = \frac{300{,}000\ words}{21.25\ Gwords/s} = 14{,}117.65 \cdot 10^{-9} s = 14.1\ \mu s$
- **Although GPU kernel time is smaller than the CPU runtime, it is not beneficial to offload this application to the GPU (due to the data transfer time)**

# 1.4 GPU Tuning Impact

- **If you consider overlapping kernel computations and data transfers, what would be the potential GPU runtime $t_{GPU\_Overlap}$ using 4 streams? What would be the speedup over the synchronous execution $t_{GPU}$?**

  Kernel-dominated: $t_{GPU\_Overlap} = t_{kernel} + \frac{t_{data}}{\#streams}$

  Transfer-dominated: $t_{GPU\_Overlap} = t_{data} + \frac{t_{kernel}}{\#streams}$

- **Since $t_{data} > t_{kernel}$, the code is transfer dominated.**
- $t_{GPU\_Overlap} = 155\ \mu s + \frac{2.86\ \mu s}{4} = 155.72\ \mu s$
- Speedup over synchronous execution: $Sp = \frac{t_{GPU}}{t_{GPU\_Overlap}} = \frac{157.86\ \mu s}{155.72\ \mu s} = 1.014$

  → This means a runtime improvement of (only) 1.4 %

# Exercise Tasks

1. GPU Performance Modeling
2. **GPU Occupancy**
3. CUDA code & GPU performance tuning
4. Jacobi – Parallelize the code with OpenACC

# GPU Occupancy

- **Given is an NVIDIA GPU of compute capability 7.0. Its technical specification can be found under https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#features-and-technical-specifications. Furthermore, a GPU kernel is given with the following characteristics:**
  - → Launch configuration: 128 threads per block
  - → Each thread block uses 8 KB of shared memory
  - → 8192 32-bit registers for each thread block (evenly distributed across threads within block)
- **What is the limiting factor here and why? To answer this question, use the technical specification given above and compute the occupancies (per SM) for the different limiters. Compare the limiters for warps (block size), shared memory and registers.**
  - → Hint: Here 1 K = 1024

# GPU Occupancy: block size

$$\text{Occupancy (per SM)} = \frac{\text{active warps}}{\text{max. supported active warps}}$$

- **Assumption**
  - → Kernel is launched with 128 threads per block ≙ 4 warps
- **Occupancy**
  - → 128 threads * 32 blocks per SM = 4096 threads = 128 warps
  - → 4096 threads /2048 threads = 2
  - → Occupancy: 100%

| Technical Specifications | CC 7.0 |
|---|---|
| Maximum number of resident warps per multiprocessor | 64 |
| Maximum number of resident threads per multiprocessor | 2048 |
| Maximum number of resident blocks per multiprocessor | 32 |
| Number of 32-bit registers per multiprocessor | 64 K |
| Maximum number of 32-bit registers per thread block | 64 K |
| Maximum number of 32-bit registers per thread | 255 |
| Maximum amount of shared memory per multiprocessor | 96 KB |
| Maximum amount of shared memory per thread block | 96 KB |

Source: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#features-and-technical-specifications

# GPU Occupancy: shared memory

$$\text{Occupancy (per SM)} = \frac{\text{active warps}}{\text{max. supported active warps}}$$

- **Assumption**
  - → Kernel uses 8 KB of shared memory per thread block
- **Occupancy**
  - → 8 KB < 96 KB per thread block: ok
  - → 96 KB per SM / 8 KB per block = 12
  - → Take floor, i.e., 12 active blocks per SM
  - → 12*128 threads /2048 threads = 0.75
  - → Occupancy: 75%

| Technical Specifications | CC 7.0 |
|---|---|
| Maximum number of resident warps per multiprocessor | 64 |
| Maximum number of resident threads per multiprocessor | 2048 |
| Maximum number of resident blocks per multiprocessor | 32 |
| Number of 32-bit registers per multiprocessor | 64 K |
| Maximum number of 32-bit registers per thread block | 64 K |
| Maximum number of 32-bit registers per thread | 255 |
| Maximum amount of shared memory per multiprocessor | 96 KB |
| Maximum amount of shared memory per thread block | 96 KB |

Source: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#features-and-technical-specifications

# GPU Occupancy: registers

$$\text{Occupancy (per SM)} = \frac{\text{active warps}}{\text{max. supported active warps}}$$

- **Assumption**
  - → Kernel uses 8192 32-bit registers per thread block
- **Occupancy**
  - → (8192/128=) 64 < 255 reg per thread: ok
  - → 8192=8K < 64K reg per block: ok
  - → 64K per SM / 8K per block = 8 blocks/SM
  - → 8*128 threads /2048 threads = 0.5
  - → Occupancy: 50%

| Technical Specifications | CC 7.0 |
|---|---|
| Maximum number of resident warps per multiprocessor | 64 |
| Maximum number of resident threads per multiprocessor | 2048 |
| Maximum number of resident blocks per multiprocessor | 32 |
| Number of 32-bit registers per multiprocessor | 64 K |
| Maximum number of 32-bit registers per thread block | 64 K |
| Maximum number of 32-bit registers per thread | 255 |
| Maximum amount of shared memory per multiprocessor | 96 KB |
| Maximum amount of shared memory per thread block | 96 KB |

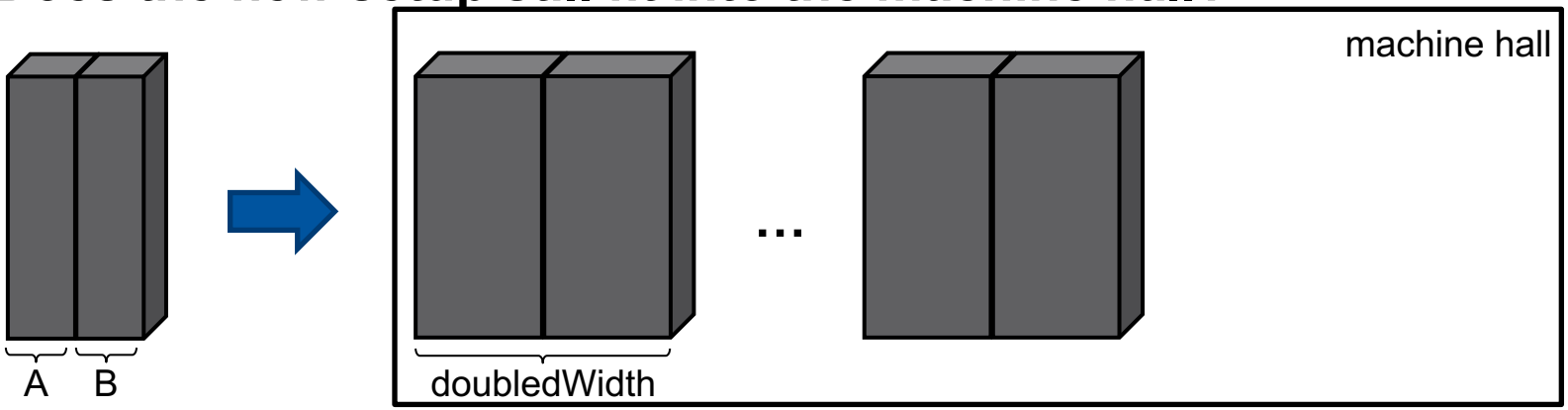
**Overall: occupancy of 50%**

Source: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#features-and-technical-specifications

# Exercise Tasks

1. GPU Performance Modeling
2. GPU Occupancy
3. **CUDA code & GPU performance tuning**
    1. Complete the CUDA code
    2. Performance tuning
4. Jacobi – Parallelize the code with OpenACC

- **Rack in machine hall: part A w/ width1 & part B w/ width2**
- **New setup: doubled rack widths**
- **Does the new setup still fit into the machine hall?**



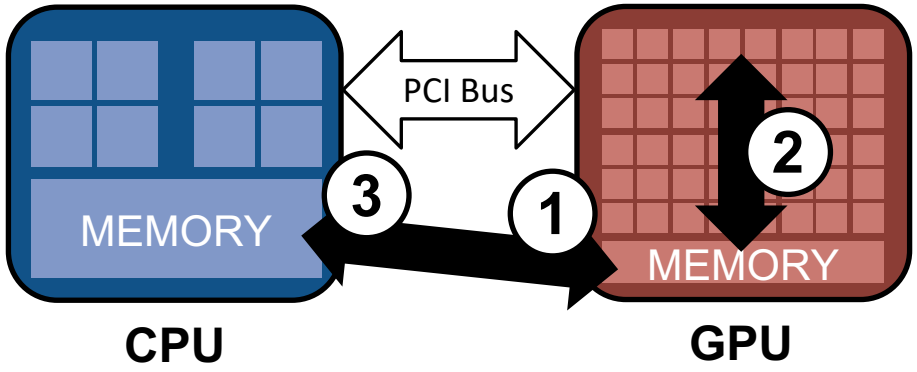

- **Solve with CUDA on GPU (following Todos)!**
  1. Allocate memory on GPU
  2. Copy initialized data from CPU to GPU
  3. Compute no. of *blocksPerGrid* so that each thread works on one rack element
  4. Call the CUDA kernel (given)
  5. Copy result data from GPU to CPU

# 3.1 Complete the CUDA code

```
// GPU kernel
__global__ void doubleTheWidth(rack_t *racks, int n)
{
  int tid = blockDim.x * blockIdx.x + threadIdx.x;
  if (tid < n) {
    // Compute the doubled width for each rack element
    racks[tid].doubledWidth = 2 * (racks[tid].widthA +
                                   racks[tid].widthB);
  }
}
```

```
struct rack_t {
  float widthA;
  float widthB;
  float doubledWidth;
};
```

- **Complete the CUDA code**




1. **cudaMalloc(…)**
   **cudaMemcpy(…)**
2. **compute b, t**
   **kernel<<<b,t>>>(args)**
3. **cudaMemcpy(…)**
   **cudaFree(…)**

# 3.1 Complete the CUDA code

```
...
// TODO 1: allocate memory on GPU
cudaMalloc((void**)&d_racks,n*sizeof(rack_t));
initRacks(h_racks,n); // init racks struct w/ values

// TODO 2: copy initialized data from CPU to GPU
cudaMemcpy(d_racks,h_racks,n*sizeof(rack_t),cudaMemcpyHostToDevice);

dim3 threads_per_block(THREADSPERBLOCK); dim3 blocks_per_grid;
// TODO 3: Compute the number of blocks_per_grid
// so that each thread works on one rack element
blocks_per_grid = dim3((n+(THREADSPERBLOCK-1))/THREADSPERBLOCK);

// TODO 4: Call the CUDA kernel
doubleTheWidth<<<blocks_per_grid,threads_per_block>>>(d_racks,n);
cudaDeviceSynchronize();

// TODO 5: Copy results data from GPU to CPU
cudaMemcpy(h_racks,d_racks,n*sizeof(rack_t),cudaMemcpyDeviceToHost);

// free memory
free(h_racks);
cudaFree(d_racks);
...
```

# 3.2 Performance Tuning (1/6)

- What is the performance problem?
- How can you solve it?

- Hint: Express the problem in terms of bus utilization (assume a GPU NVIDIA Volta architecture w/ 128B cache line).

- Re-define the *struct* correspondingly.

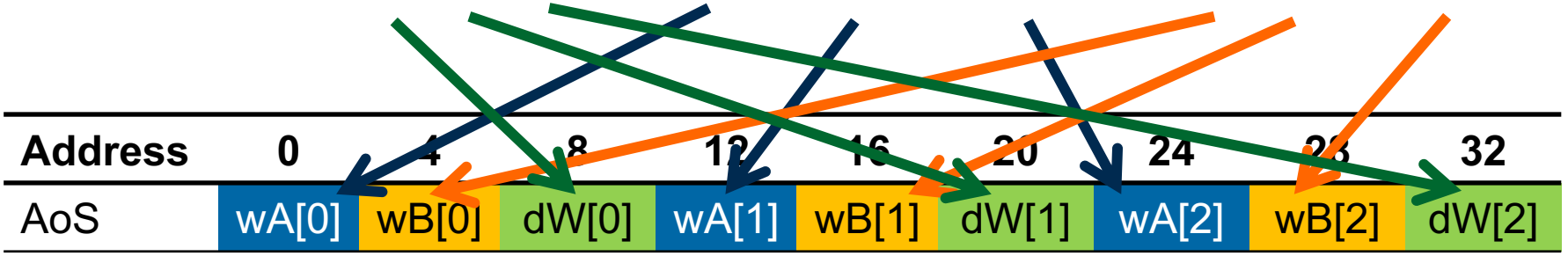
# 3.2 Performance Tuning (2/6)

- **Performance tuning (performance issue & solution)**

```
struct rack_t {
  float widthA;
  float widthB;
  float doubledWidth;
}
rack_t *racks;
```

Problem:
Array of Structures (AoS)

```
// kernel
racks[tid].doubledWidth = 2* (racks[tid].widthA + racks[tid].widthB);
```

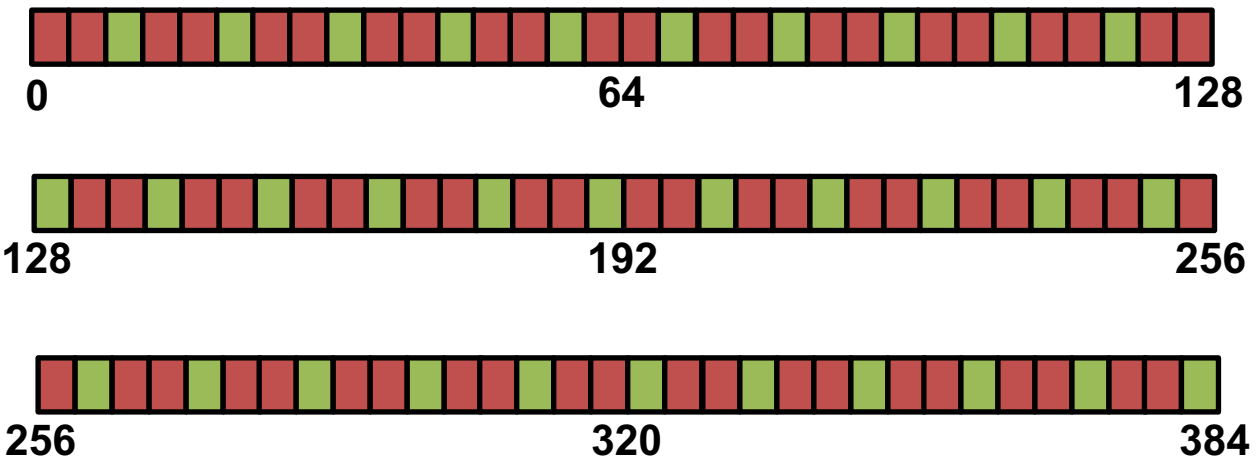




This slide is based on: NVIDIA Corporation

# 3.2 Performance Tuning (3/6)

- **racks[tid].widthA & racks[tid].widthB**



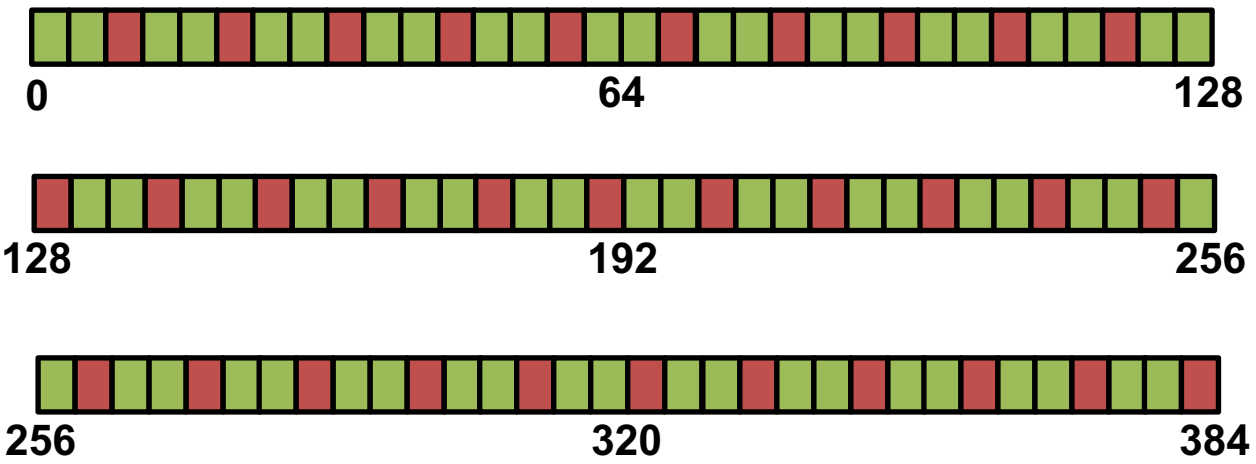

memory access of one warp (32 threads)

- **Needed: 2 * 32 * 4 Byte = 256 Byte**
- **Moved: 3 * 128 Byte = 384 Byte**

→ **Bus utilization: 256 / 384 ≈ 0.6667 = 66.67 %**

# 3.2 Performance Tuning (4/6)

- **Bus utilization (Volta): racks[tid]. doubledWidth**




- **Needed: 32 * 4 Byte = 128 Byte**
- **Moved: 3 * 128 Byte = 384 Byte**

→ **Bus utilization: 128 / 384 ≈ 0.3333 = 33.33 %**

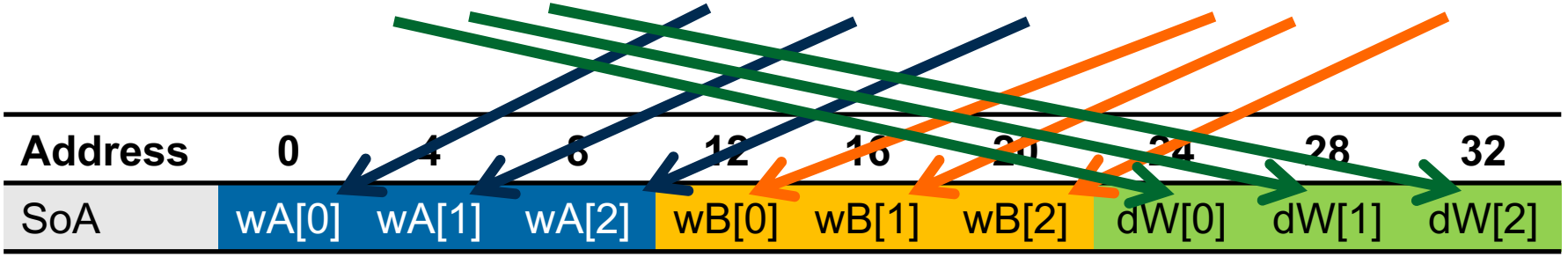
# 3.2 Performance Tuning (5/6)

## Re-definition*

```
struct rackSoA_t {
  float* widthA;
  float* widthB;
  float* doubledWidth;
}
rackSoA_t racks;
```

Structure of Arrays (SoA): coalesced memory access

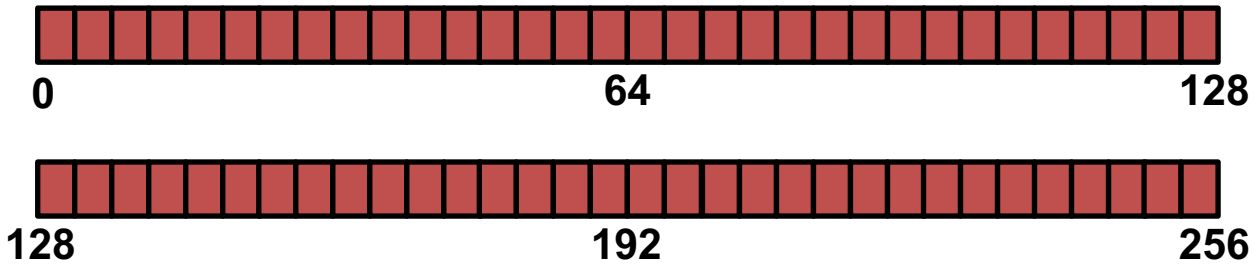
```
// kernel
racks.doubledWidth[tid] = 2* (racks.widthA[tid] + racks.widthB[tid]);
```




* rewritten source code online

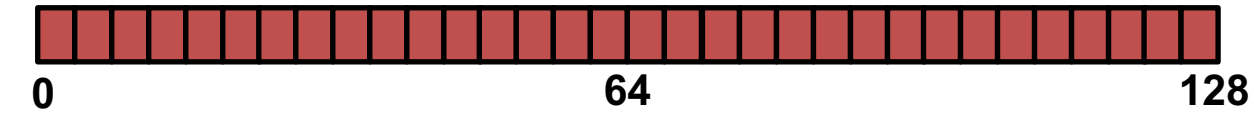
# 3.2 Performance Tuning (6/6)

- **Bus utilization (Volta): racks.widthA[tid] & racks.widthB[tid]**




memory access of one warp (32 threads)

- **Bus utilization: racks. doubledWidth[tid]**




- **Needed: 2 * 32 * 4 Byte = 256 Byte**
- **Moved: 2 * 128 Byte = 256 Byte**
- → **Bus utilization: 100 %**

additional data transfer savings
- no copy doubledWidth to dev
- no copy widthA, widthB from dev

→ If kernel memory bound, ~2x performance increase



**LIVE DEMO**

# Exercise Tasks

1. GPU Performance Modeling
2. GPU Occupancy
3. CUDA code & GPU performance tuning
4. **Jacobi – Parallelize the code with OpenACC**
   1. Prepare your GPU cluster environment
   2. First Jacobi parallelization w/ OpenACC
   3. Code profiling
   4. Data transfers
   5. Loop scheduling

# 4.1 Prepare your GPU cluster environment

- **Login to:** `login18-g-1`
- **Examine GPU hardware:** `pgaccelinfo`

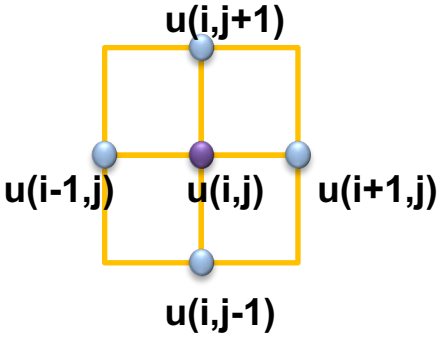
- **Solution**

| Feature | Value |
|---|---|
| Number & name of device | 2, Tesla V100-SXM2-16GB |
| Number of SMs and cores | 5120 (80 SMPs with each 64 SP cores) |
| CUDA compute capability (cc) | 7.0 |

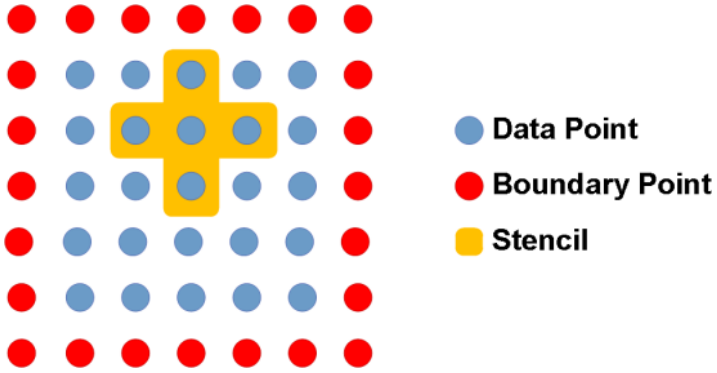
# Jacobi – Parallelize the Code w/ OpenACC

- **Solving Laplace equation (2D) by Jacobi iterative method**

$$\Delta u(x,y) = \nabla^2 u(x,y) = 0$$




  - → Start with guess of objective function *u(x,y)*
  - → Use finite difference discretization (stencil) $u_{k+1}(i,j) = \frac{u_k(i-1,j)+u_k(i+1,j)+u_k(i,j-1)+u_k(i,j+1)}{4}$
  - → Iterate over inner elements of 2D grid




- **Repeat until**
  - → Maximum of number of iterations reached
  - → Computed approximation close to solution
    - →Biggest change on any matrix element < tolerance value

# Jacobi – Parallelize the Code w/ OpenACC

- **LIVE DEMO**

| Hardware | Version | Runtime [s] |
|---|---|---|
| 2x Intel Xeon Platinum 8160 @ 2.1GHz 48 cores in total) | Serial (PGI compiler) | 4.88 |
| | OpenMP (PGI compiler) | 0.61 |
| NVIDIA Volta V100 (cc 7.0) | OpenACC-basic | |
| | OpenACC-data | |
| | OpenACC-loop | |

# 4.2 First Jacobi parallelization w/ OpenACC

- **Use `acc parallel` directive + work sharing on first loop nest**
- **Investigate compiler feedback**
  - → Accelerator kernel generated
  - → Reduction for variable `err`
- **Run code and note runtime**

# 4.2 First Jacobi parallelization w/ OpenACC

```
  while (err > tol && iter < iter_max) {
    err = 0.0;
#pragma acc parallel loop copyin(U[0:n][0:m]) copyout(Unew[0:n][0:m]) \
        reduction(max : err)
    for (i = 1; j < n - 1; i++) {
      for (j = 1; j < m - 1; j++) {
        Unew[i][j] = 0.25 * (U[i][j+1] + U[i][j-1] +
                             U[i-1][j] + U[i+1][j]);
        err = fmax(err, fabs(Unew[i][j] - U[i][j]));
    } }

    for (i = 1; i < n - 1; i++) {
      for (j = 1; j < m - 1; j++) {
        U[i][j] = Unew[i][j];
    } }
    iter++;
  } // end while
```

# 4.2 First Jacobi parallelization w/ OpenACC

- **LIVE DEMO**

| Hardware | Version | Runtime [s] |
|---|---|---|
| 2x Intel Xeon Platinum 8160 @ 2.1GHz 48 cores in total) | Serial (PGI compiler) | 4.88 |
| | OpenMP (PGI compiler) | 0.61 |
| NVIDIA Volta V100 (cc 7.0) | OpenACC-basic | 5.50 |
| | OpenACC-data | |
| | OpenACC-loop | |

# 4.3 Code Profiling

- **Basic runtime profiling by setting environment variable `PCI_ACC_TIME=1`**

  → Compare time spent for kernel execution and data transfer

- **Use NVIDIA's Visual Profiler (`nvvp`)**

  → Read kernel execution time and data transfer time

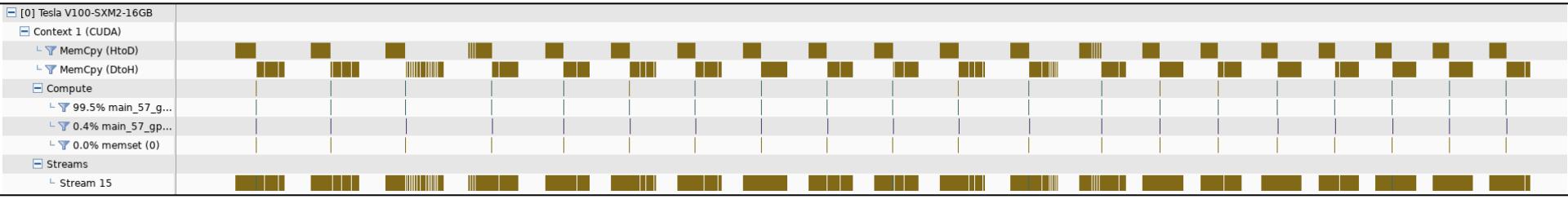
# 4.3 Code Profiling – PGI_ACC_TIME=1

```
main  NVIDIA  devicenum=0
   time(us): 1,755,246
   57: compute region reached 20 times
       57: kernel launched 20 times
           grid: [8190]  block: [128]
            device time(us): total=62,485 max=3,146 min=[illegible] avg=3,124
           elapsed time(us): total=63,302 max=3,186 min=[illegible],146 avg=3,165
       57: reduction kernel launched 20 times
           grid: [1]  block: [256]
            device time(us): total=301 max=16 min=15 avg=15
           elapsed time(us): total=692 max=38 min=33 avg=34
   57: data region reached 40 times
       57: data copyin transfers: 660
            device time(us): total=871,541 max=1,382 min=10 avg=1,320
       72: data copyout transfers: 660
            device time(us): total=820,919 max=1,299 min=8 avg=1,243
```

Kernel: 63 ms
Data transfer: 872 ms + 821 ms = 1693 ms
Ratio: 1693/63 = 27

# 4.3 Code Profiling – nvvp

## Timeline




Data transfers:
867 ms + 815 ms

Properties

**Compute**

| | |
|---|---|
| Duration | |
| Session | 6.699 s (6,698,861,903 ns) |
| Kernels | 62.565 ms (62,564,771 ns) |
| Compute Utilization | 0.9% |
| Kernel Invocations | 60 |

Kernel: 63 ms

Properties

**MemCpy (HtoD)**

| | |
|---|---|
| Duration | |
| Session | 6.699 s (6,698,861,903 ns) |
| Memcpys | 867.417 ms (867,417,204 ns) |
| Invocations | 640 |
| Total Bytes | 10.737 GB |
| Avg. Throughput | 12.379 GB/s |

Properties

**MemCpy (DtoH)**

| | |
|---|---|
| Duration | |
| Session | 6.699 s (6,698,861,903 ns) |
| Memcpys | 815.011 ms (815,011,432 ns) |
| Invocations | 660 |
| Total Bytes | 10.732 GB |
| Avg. Throughput | 13.168 GB/s |

# 4.4 Data Transfers

- Offload second loop nest to GPU
- Use `acc data` to remove unneeded data movements. Use the `present` clause, too.
- Investigate compiler feedback (changes?)
- Run code and note runtime
- Check runtime changes by profiling

# 4.4 Data Transfers

```
#pragma acc data copy(U[0:n][0:m]) create(Unew[0:n][0:m])
{
  while (err > tol && iter < iter_max) {
    err = 0.0;
#pragma acc parallel loop present(U,Unew) reduction(max : err)
    for (i = 1; j < n - 1; i++) {
      for (j = 1; j < m - 1; j++) {
        Unew[i][j] = 0.25 * (U[i][j+1] + U[i][j-1] +
                             U[i-1][j] + U[i+1][j]);
        err = fmax(err, fabs(Unew[i][j] - U[i][j]));
    } }
#pragma acc parallel loop present(U,Unew)
    for (i = 1; i < n - 1; i++) {
      for (j = 1; j < m - 1; j++) {
        U[i][j] = Unew[i][j];
    } }
    iter++;
  } // end while
}
```

# 4.4 Data Transfers

- **LIVE DEMO**

| Hardware | Version | Runtime [s] |
|---|---|---|
| 2x Intel Xeon Platinum 8160 @ 2.1GHz 48 cores in total) | Serial (PGI compiler) | 4.88 |
| | OpenMP (PGI compiler) | 0.61 |
| NVIDIA Volta V100 (cc 7.0) | OpenACC-basic | 5.50 |
| | OpenACC-data | 0.32 |
| | OpenACC-loop | |

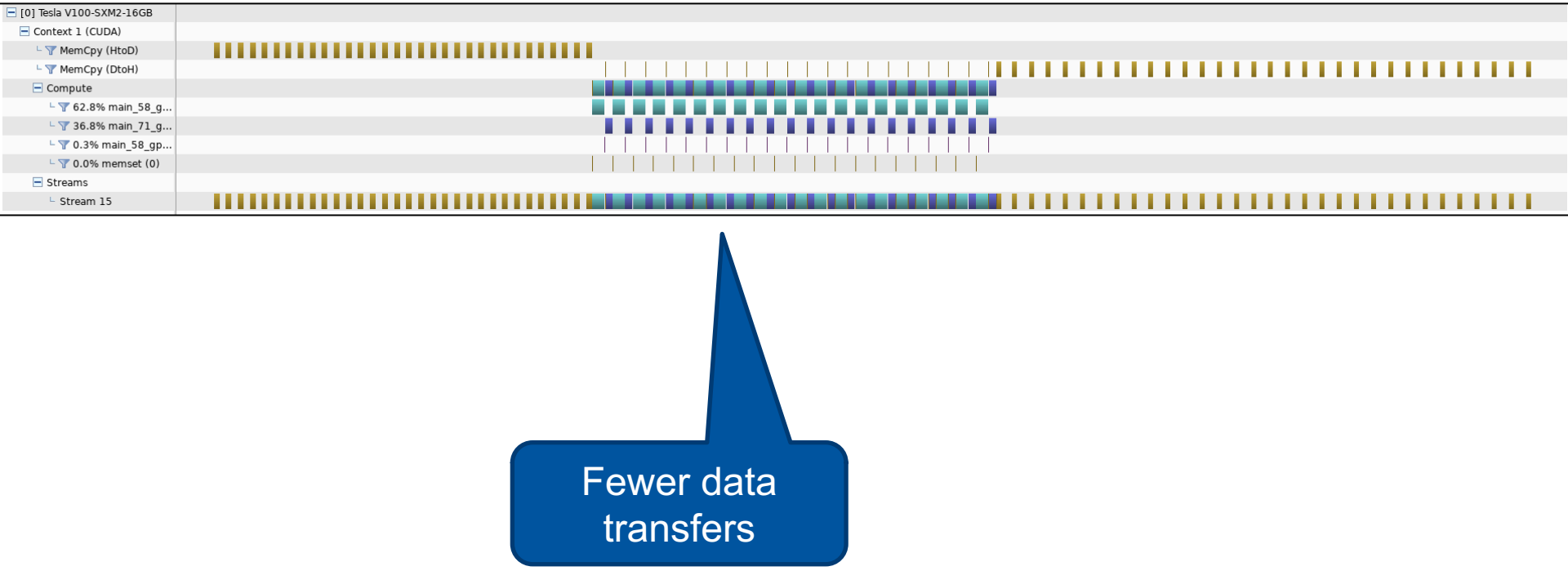
# 4.4 Data Transfers – PGI_ACC_TIME=1

```
main  NVIDIA  devicenum=0
    time(us): 183,937
    52: data region reached 2 times
        52: data copyin transfers: 32
             device time(us): total=43,626
        88: data copyout transfers: 33
             device time(us): total=41,007 max=1,28          g=1,242
    58: compute region reached 20 times
        58: kernel launched 20 times
            grid: [8190]  block: [128]
             device time(us): total=62,222 max=3,137 in=3,091 avg=3,111
            elapsed time(us): total=62,622 max=3,184 min=3,109 avg=3,131
        58: reduction kernel launched 20 times
            grid: [1]  block: [256]
             device time(us): total=288 max=16 min=14 avg=14
            elapsed time(us): total=652 max=42 min=30 avg=32
    58: data region reached 80 times
        58: data copyin transfers: 20
             device time(us): total=72 max=13 min=3 avg=3
        71: data copyout transfers: 20
             device time(us): total=159 max=17 min=7 avg=7
    71: compute region reached 20 times
        71: kernel launched 20 times
            grid: [8190]  block: [128]
             device time(us): total=36,563 max=1,845 min=1,813 avg=1,828
            elapsed time(us): total=36,959 max=1,864 min=1,832 avg=1,847
    71: data region reached 40 times
```

Kernel: 63 ms + 37 ms = 100 ms
Data transfer: 44 ms + 41 ms = 85 ms
Ratio: 85/100 = 0.85

# 4.4 Data Transfers – nvvp

- **Timeline**




Fewer data transfers

# 4.5 Loop Scheduling

- **Compare the compiler feedback of two different OpenACC Jacobi solver versions (see next slide)**
- **Which program would you expect to run faster? Why**
- **Run the codes and note runtimes**

# 4.5 Loop Scheduling

```
main:
     52, Gener
         Gener
     58, Gener
         prese
         Accel
         Gener
         58, Generating
         reduction(max:err)
         60, #pragma acc loop gang /*
         blockIdx.x */
         63, #pragma acc loop vector(128)
         /* threadIdx.x */
     63, Loop is parallelizable
     71, Generating
         present(Unew[:][:],U[:][:])
         Accelerator kernel generated
         Generating Tesla code
         73, #pragma acc loop gang /*
         blockIdx.x */
         76, #pragma acc loop vector(128)
         /* threadIdx.x */
     76, Loop is parallelizable
```

Outer loop across thread blocks
Inner loop across threads within blocks
→ Lots of parallelism: (8192 x 8192)

```
main:
     52, Gener
         Gener
     58, Gener
         prese
         Accel
         Gener
         58, Generating
         reduction(max:err)
         60, #pragma acc loop gang,
         vector(128) /* blockIdx.x
         threadIdx.x */
         63, #pragma acc loop seq
     63, Loop is parallelizable
     71, Generating
         present(Unew[:][:],U[:][:])
         Accelerator kernel generated
         Generating Tesla code
         73, #pragma acc loop gang,
         vector(128) /* blockIdx.x
         threadIdx.x */
         76, #pragma acc loop seq
     76, Loop is parallelizable
```

Outer loop across threads and thread blocks
Inner loop sequential
→ Less parallelism: (8192)

# 4.5 Loop Scheduling

- **LIVE DEMO**

| Hardware | Version | Runtime [s] |
|---|---|---|
| 2x Intel Xeon Platinum 8160 @ 2.1GHz 48 cores in total) | Serial (PGI compiler) | 4.88 |
| | OpenMP (PGI compiler) | 0.61 |
| NVIDIA Volta V100 (cc 7.0) | OpenACC-basic | 5.50 |
| | OpenACC-data | 0.32 |
| | OpenACC-loop | 0.30 |