



GPU Programming & Performance

S. Kuckuk, D. Ernst, H. Köstler, G. Wellein, G. Hager

Erlangen National High Performance Computing Center (NHR@FAU)

GPU Programming & Performance

Motivation & GPU Architecture

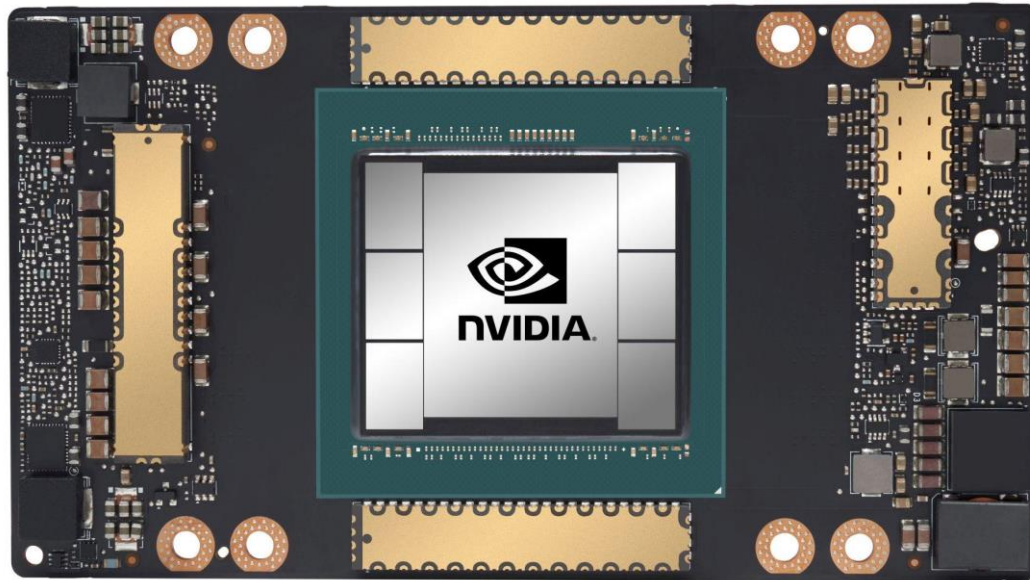


GPUs in HPC – Motivation

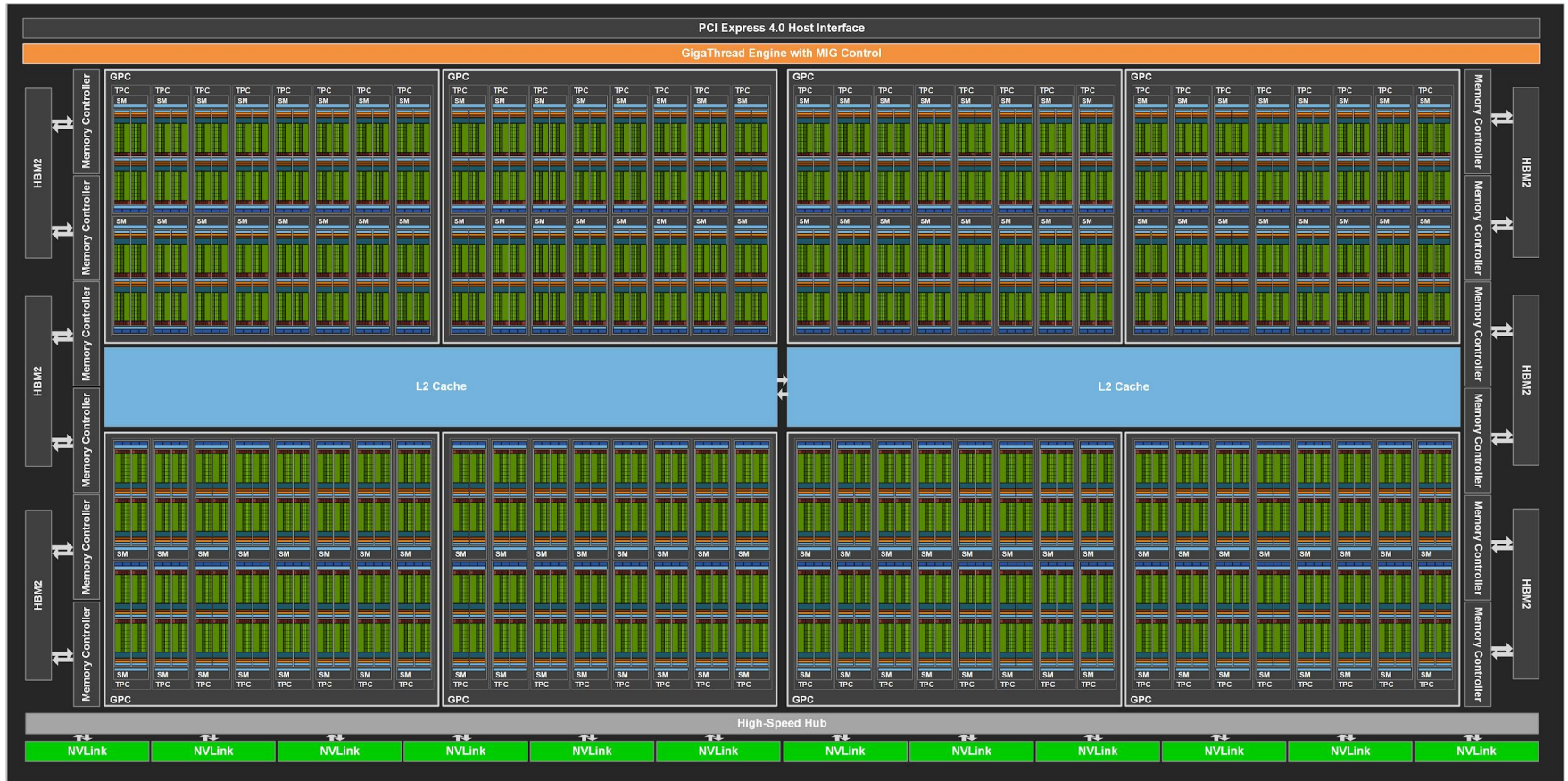
- Promises
 - Massive parallelism and performance
 - Good performance in relation to energy (FLOPs per Watt)
- Already widespread in the HPC landscape
 - c.f. Top500 list (<https://www.top500.org/lists/top500/2024/06/>)
 - 9 out of the top 10 supercomputers are equipped with GPUs
 - 6 NVIDIA (3 x H100, GH200, A100, V100)
 - 2 AMD (2 x MI250X)
 - 1 Intel (GPU Max Series)
- Where does the performance come from?

GPU Architecture – Example: A100 as in Alex

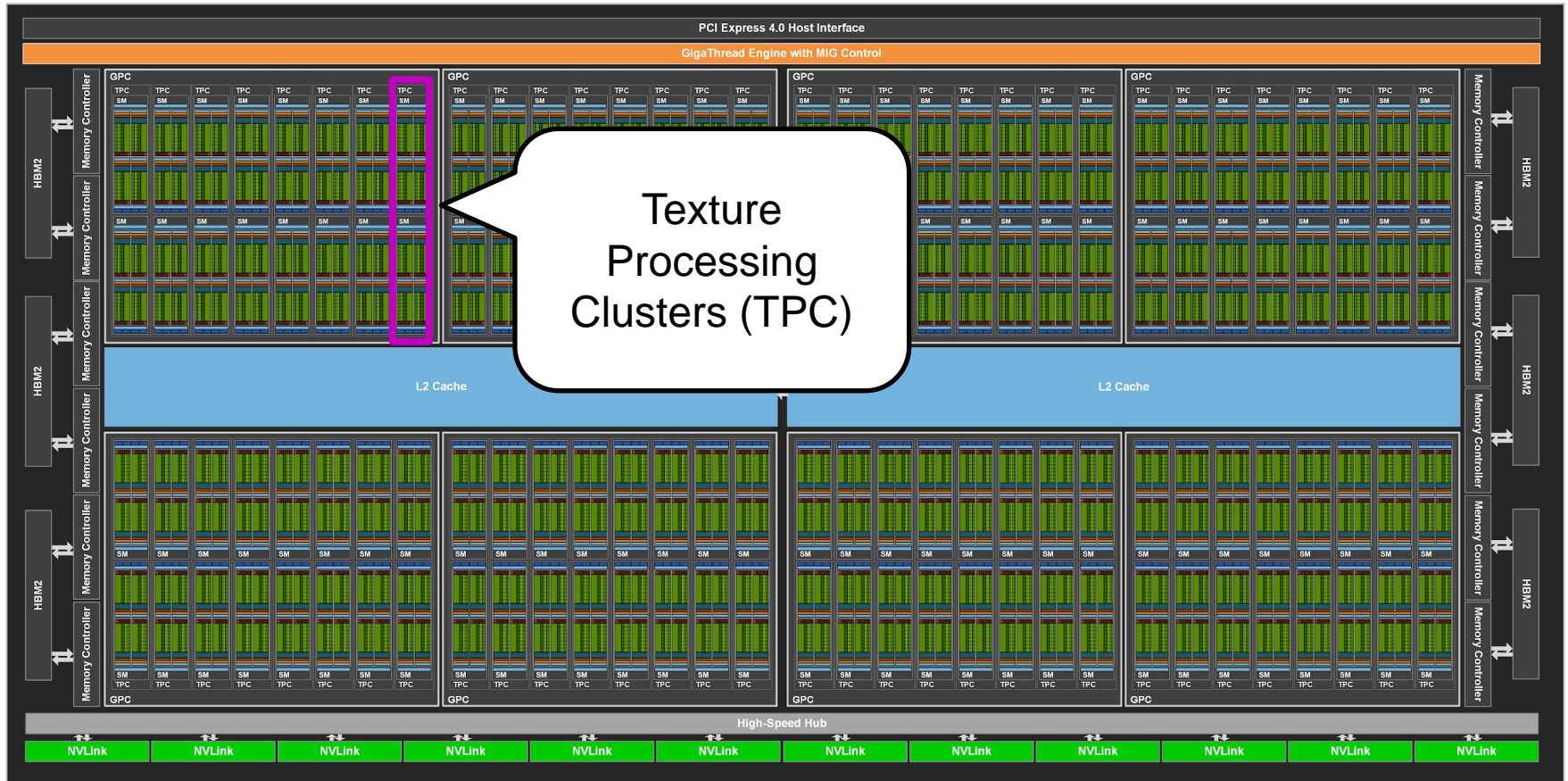
- Detailed documentation as whitepaper
 - <https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf>



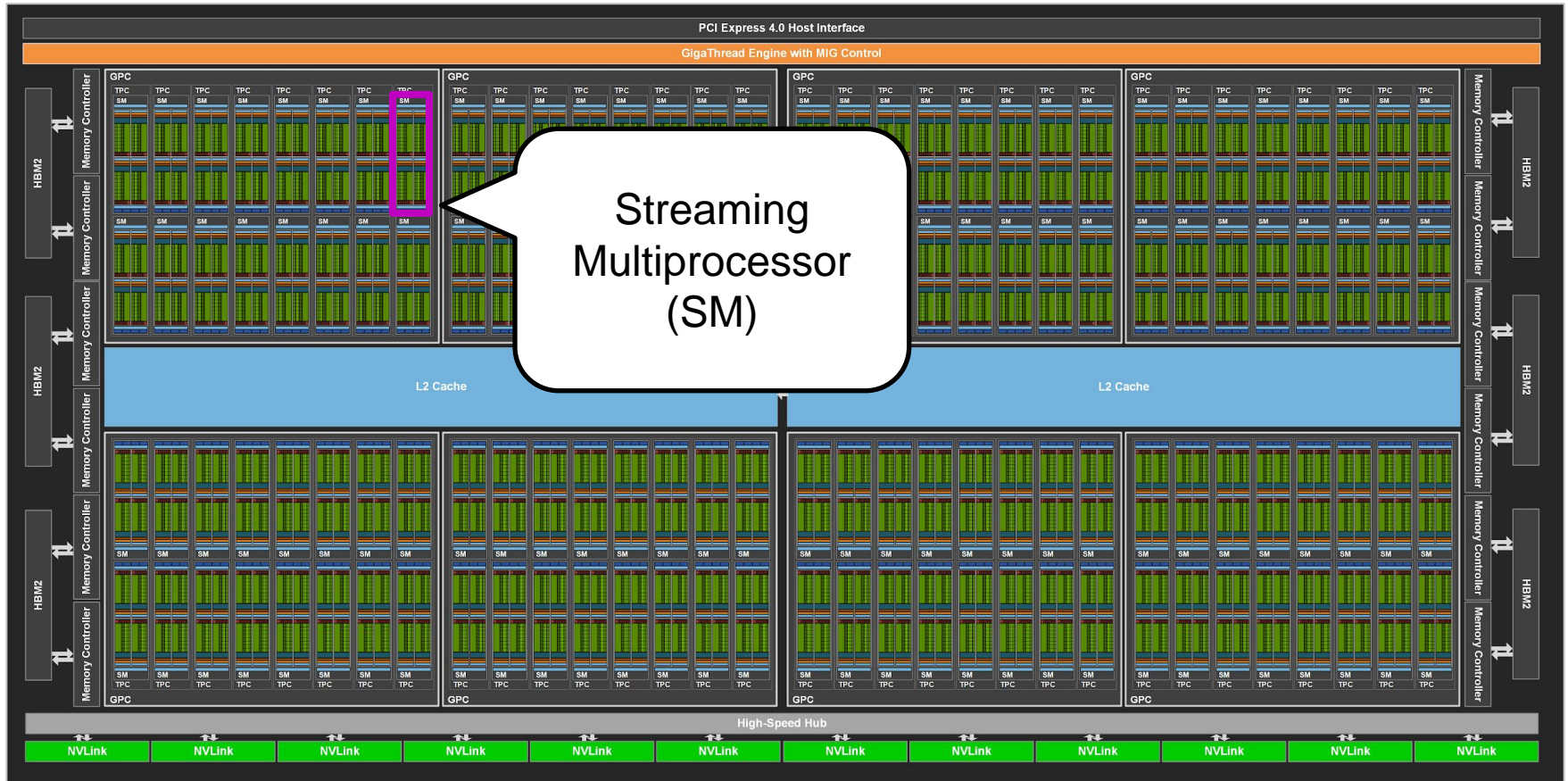
GPU Architecture – Example: A100 as in Alex



GPU Architecture – Example: A100 as in Alex



GPU Architecture – Example: A100 as in Alex



GPU Architecture – Example: A100 as in Alex

- Previous slides show ‘full configuration’
 - One A100 GPU features
 - Seven Graphics Processing Clusters (GPCs), each with
 - Seven/ eight Texture Processing Clusters (TPCs), each with
 - Two Streaming Multiprocessors (SMs)
- Total of 108 SMs

GPU Architecture – Example: A100 as in Alex

- Total of 108 SMs, each with
 - Four sub partitions, each with
 - 16 INT32 units
 - Total: $108 * 4 * 16 = 6912$
 - 16 FP32 units
 - Total: $108 * 4 * 16 = 6912$
 - 8 FP64 units
 - Total: $108 * 4 * 8 = 3456$
 - One tensor core
 - Total: $108 * 4 = 432$
 - Each: 256 FP16/FP32-mixed-prec. FMAs



GPU Architecture – Example: A100 as in Alex

Performance: $P_{chip} = n_{core} \cdot n_{super}^{FP} \cdot n_{FMA} \cdot n_{SIMD} \cdot f$

The diagram shows the performance equation $P_{chip} = n_{core} \cdot n_{super}^{FP} \cdot n_{FMA} \cdot n_{SIMD} \cdot f$. Below the equation, five labels are positioned: '#Cores' under n_{core} , 'Super-scalarity' under n_{super}^{FP} , 'FMA factor' under n_{FMA} , 'SIMD factor' under n_{SIMD} , and 'Clock Speed' under f . Blue arrows point from each label to its corresponding variable in the equation.

- For FP32

- $P_{chip} = 6912 \cdot 1 \cdot 2 \cdot 1 \cdot 1.41 \text{ GF/s} = 19.5 \text{ TF/s}$
 $= 432 \cdot 1 \cdot 2 \cdot 16 \cdot 1.41 \text{ GF/s} = 19.5 \text{ TF/s}$

- For FP64: 9.7 TF/s

- For INT32: 19.5 TO/s

GPU Architecture – Example: A100 as in Alex

Memory

- Per GPU
 - 40/ 80 GB HBM2 main memory with 1555/ 2039 GB/s
 - 20+20 = 40MB L2 cache
- Per SM
 - 192 KB of combined shared memory and L1 cache
 - 4x64 KB register file

GPU Architecture – Example: A100 as in Alex

Interconnect

- Reference: main memory with 1555/ 2039 GB/s
- Connection to CPU with PCIe 4.0 x16 with 31.5 GB/s per direction
- Connection to other GPUs in the same node via NVLink with 300 GB/s per direction

CPU-GPU Comparison

CPU

- Cores
 - Few but powerful
 - Branch prediction & speculative execution
 - Prefetchers
 - Low latency instructions
 - Out-of-Order Execution

GPU

- 'Cores'
 - Many but less powerful
 - Extreme SMT

CPU-GPU Comparison

CPU

- Cores
 - Few but powerful
- Memory
 - Large capacity
 - Latency optimized

GPU

- 'Cores'
 - Many but less powerful
- Memory
 - Small capacity
 - Bandwidth optimized

CPU-GPU Comparison

CPU

- Cores
 - Few but powerful
- Memory
 - Large capacity
 - Latency optimized
- Ideal for irregular workload/ task parallelism

GPU

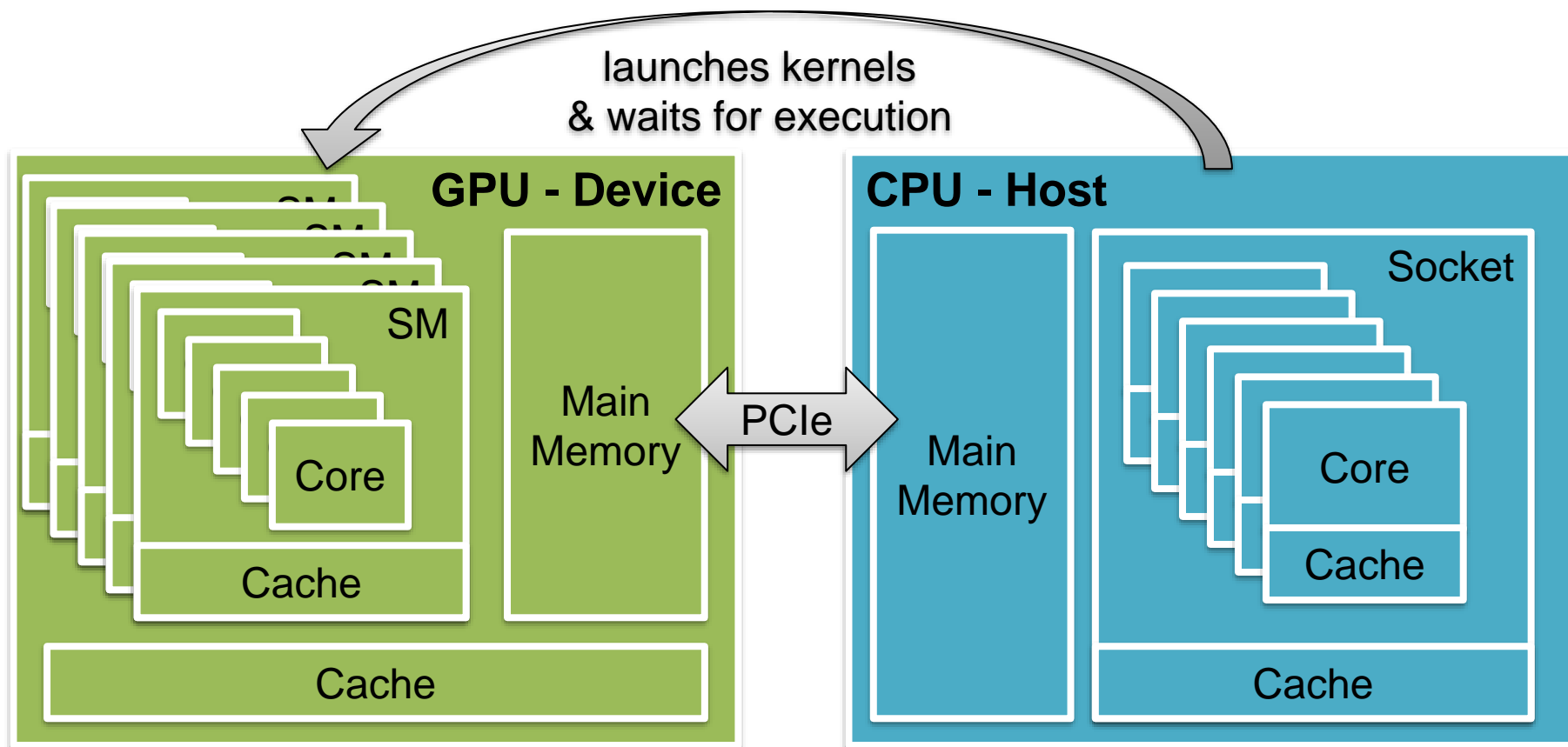
- 'Cores'
 - Many but less powerful
- Memory
 - Small capacity
 - Bandwidth optimized
- Ideal for massively parallel structured computations

GPU Programming & Performance

GPU Programming



Simplified Architecture



GPU Programming Approaches

- Dedicated programming language (extensions)
 - E.g. NVIDIA CUDA, AMD HIP, SYCL/ Intel OneAPI
 - Full control – more evolved code for maximized performance potential
- Pragma-based approaches
 - E.g. OpenMP target offloading, OpenACC
 - Easy to integrate – if everything works as intended
- Software layers
 - E.g. Kokkos
 - Performance portability – ideally


Workflow (CPU)

1. Allocate data
2. Initialize data
3. Do work
4. Post-process data
5. De-Allocate data

Workflow (GPU)

1. Allocate data for CPU/GPU
2. Initialize data on CPU
3. Copy data from CPU to GPU
4. Launch GPU kernels
5. Do independent work on CPU (optional)
6. Synchronize GPU
7. Copy data from GPU to CPU
8. Post-process data on CPU
9. De-Allocate data

Workflow (GPU)

1. Allocate data for CPU/GPU
2. Initialize data on CPU
3. Copy data from CPU to GPU
4. Launch GPU kernels 
5. Do independent work on CPU (optional)
6. Synchronize GPU
7. Copy data from GPU to CPU
8. Post-process data on CPU
9. De-Allocate data

GPU Kernel example

CPU code

```
void workOnCPU(  
    int *data, size_t nx) {  
  
    for (size_t i = 0; i < nx; ++i)  
        data[i] = 0;  
}  
  
// in main  
workOnCPU(data, nx);
```

GPU Kernel example

CPU code

```
void workOnCPU(  
    int *data, size_t nx) {  
  
    for (size_t i = 0; i < nx; ++i)  
        data[i] = 0;  
}  
  
// in main  
workOnCPU(data, nx);
```

Serial kernel (only for porting)

```
__global__ void workSerial(  
    int *data, size_t nx) {  
  
    for (size_t i = 0; i < nx; ++i)  
        data[i] = 0;  
}  
  
// in main  
workSerial<<<1, 1>>>(d_data, nx);
```


GPU Kernel example

CPU code

```
void workOnCPU(  
    int *data, size_t nx) {  
  
    for (size_t i = 0; i < nx; ++i)  
        data[i] = 0;  
}  
  
// in main  
workOnCPU(data, nx);
```

Serial kernel (only for porting)

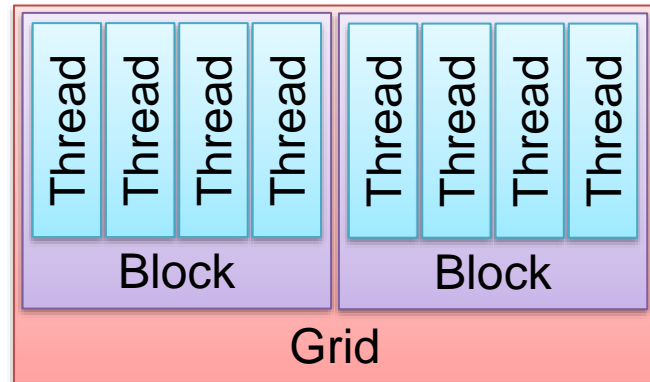
```
__global__ void workSerial(  
    int *data, size_t nx) {  
  
    for (size_t i = 0; i < nx; ++i)  
        data[i] = 0;  
}  
  
// in main  
workSerial<<<1, 1>>>(d_data, nx);
```

GPU data – see later slides

Execution configuration: one
block with one thread each

GPU Kernel Concepts

- CPU code *launches kernels* that then execute *asynchronously* on the GPU
- *Kernels* are executed by many *threads*
- The number of threads is specified by the *execution configuration*
- Threads are organized hierarchically
 - Grids > blocks > threads



GPU Kernel Concepts

- CPU code *launches kernels* that then execute *asynchronously* on the GPU
- *Kernels* are executed by many *threads*
- The number of threads is specified by the *execution configuration*
- Threads are organized hierarchically (Grids > blocks > threads)
- Each *thread* executes the whole *kernel body*
- *Built-in variables* allow identifying *thread ids* and mapping to work items

GPU Kernel example

Serial kernel (only for porting)

```
__global__ void workSerial(  
    int *data, size_t nx) {  
  
    for (size_t i = 0; i < nx; ++i)  
  
        data[i] = 0;  
}  
  
// in main  
workSerial<<<1, 1>>>(d_data, nx);
```

GPU Kernel example

Serial kernel (only for porting)

```
__global__ void workSerial(  
    int *data, size_t nx) {  
  
    for (size_t i = 0; i < nx; ++i)  
  
        data[i] = 0;  
}  
  
// in main  
workSerial<<<1, 1>>>(d_data, nx);
```

Parallel kernel (without checks)

```
__global__ void work(  
    int *data, size_t nx) {  
  
    size_t i = blockIdx.x * blockDim.x  
        + threadIdx.x;  
  
    data[i] = 0;  
}  
  
// in main  
work<<<nx / 256, 256>>>(d_data, nx);
```

GPU Kernel example

Serial kernel (only for porting)

```
__global__ void workSerial(  
    int *data, size_t nx) {
```

```
    for (size_t i = 0; i < nx; ++i)
```

```
        data[i] = 0;
```

```
    }
```

```
// in main
```

```
workSerial<<<1, 1>>>(d_data, nx);
```

Unique *data index* (or *global thread index*)

#blocks, #threads per block
(assumes evenly divisible nx)

Parallel kernel (without checks)

```
__global__ void work(  
    int *data, size_t nx) {
```

```
    size_t i = blockIdx.x * blockDim.x  
            + threadIdx.x;
```

```
    data[i] = 0;
```

```
    }
```

```
// in main
```

```
work<<<nx / 256, 256>>>(d_data, nx);
```

Each thread updates a single item

GPU Kernel example

Parallel kernel (without checks)

```
__global__ void work(  
    int *data, size_t nx) {  
  
    size_t i = blockIdx.x * blockDim.x  
        + threadIdx.x;  
  
    data[i] = 0;  
}  
  
// in main  
work<<<nx / 256, 256>>>(d_data, nx);
```

GPU Kernel example

Parallel kernel (without checks)

```
__global__ void work(  
    int *data, size_t nx) {  
  
    size_t i = blockIdx.x * blockDim.x  
            + threadIdx.x;  
  
    data[i] = 0;  
}  
  
// in main  
work<<<nx / 256, 256>>>(d_data, nx);
```

Parallel kernel


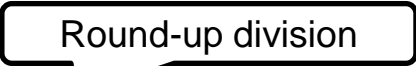
```
__global__ void work(  
    int *data, size_t nx) {  
  
    size_t i = blockIdx.x * blockDim.x  
            + threadIdx.x;  
  
    if (i < nx)  
        data[i] = 0;  
}  
  
// in main  
work<<<(nx+255)/256, 256>>>(d_data, nx);
```


GPU Kernel example

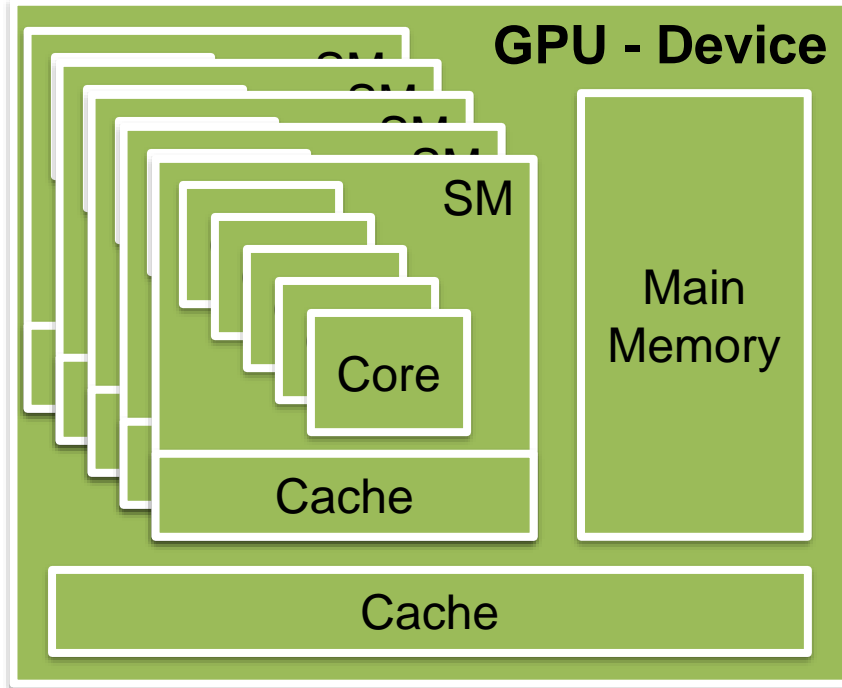
Parallel kernel (without checks)

```
__global__ void work(  
    int *data, size_t nx) {  
  
    size_t i = blockIdx.x * blockDim.x  
            + threadIdx.x;  
  
    data[i] = 0;  
}  
  
// in main  
work<<<nx / 256, 256>>>(d_data, nx);
```

Parallel kernel

```
__global__ void work(  
    int *data, size_t nx) {  
  
    size_t i = blockIdx.x * blockDim.x  
            + threadIdx.x;  
  
    if (i < nx)    
        data[i] = 0;  
}  
  
// in main    
work<<<(nx+255)/256, 256>>>(d_data, nx);
```

CUDA Mapping



- **Grids** are mapped to **devices**
- **Blocks** are mapped to **SMs**
- **Threads** are mapped to **cores**
- **Threads of a block** are executed in **warps** (groups of 32 threads)

GPU Kernel example

Parallel kernel

```
__global__ void work(  
    int *data, size_t nx) {  
  
    size_t i = blockIdx.x * blockDim.x  
        + threadIdx.x;  
  
    if (i < nx)  
        data[i] = 0;  
}  
  
// in main  
work<<<(nx+255)/256, 256>>>(d_data, nx);
```

GPU Kernel example

Parallel kernel

```
__global__ void work(  
    int *data, size_t nx) {  
  
    size_t i = blockIdx.x * blockDim.x  
            + threadIdx.x;  
  
    if (i < nx)  
        data[i] = 0;  
}  
  
// in main  
work<<<(nx+255)/256, 256>>>(d_data, nx);
```

Parallel kernel w/ grid-stride loop

```
__global__ void work(  
    int *data, size_t nx) {  
  
    size_t i = blockIdx.x * blockDim.x  
            + threadIdx.x;  
    size_t stride = blockDim.x * blockDim.x;  
  
    for ( ; i < nx; i += stride)  
        data[i] = 0;  
}  
  
// in main  
work<<<32 * 108, 256>>>(d_data, nx);
```

GPU Kernel example

Parallel kernel

```
__global__ void work(  
    int *data, size_t nx) {  
  
    size_t i = blockIdx.x * blockDim.x  
            + threadIdx.x;  
  
    if (i < nx)  
        data[i] = 0;  
}  
  
// in main  
work<<<(nx+255)/256, 256>>>(d_data, nx);
```

Parallel kernel w/ grid-stride loop

```
__global__ void work(  
    int *data, size_t nx) {  
  
    size_t i = blockIdx.x * blockDim.x  
            + threadIdx.x;  
    size_t stride = blockDim.x * blockDim.x;  
  
    for ( ; i < nx; i += stride)  
        data[i] = 0;  
}  
  
// in main  
work<<<32 * 108, 256>>>(d_data, nx);
```

Each thread updates zero, one, or more items

Arbitrary number of blocks chosen to be a multiple of #SM

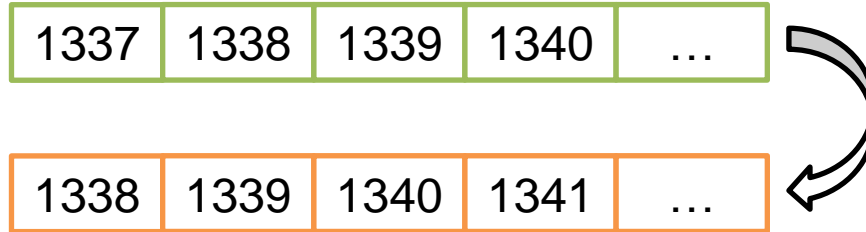
GPU Programming & Performance

GPU Programming Example – Stream Benchmark



Example Application

- Copy array and increase each element by 1 (on the GPU)



1. Allocate Data

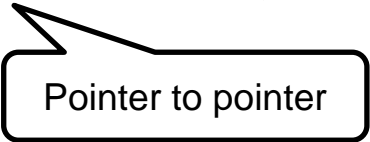
```
int main(int argc, char *argv[]) {  
    size_t nx = atoi(argv[1]);  
    size_t size = sizeof(double) * nx;  
}
```


1. Allocate Data

```
int main(int argc, char *argv[]) {
    size_t nx    = atoi(argv[1]);
    size_t size  = sizeof(double) * nx;

    // allocate _host_ arrays
    double *src, *dest;
    cudaMallocHost(&src, size);
    cudaMallocHost(&dest, size);

    // allocate _device_ arrays
    double *d_src, *d_dest;
    cudaMalloc(&d_src, size);
    cudaMalloc(&d_dest, size);
```



Pointer to pointer

2. Initialize data on CPU

```
int main(int argc, char *argv[]) {  
    // allocate
```

```
    initOnCPU(src, nx);
```

```
void initOnCPU(double *src, size_t nx) {  
    for (size_t i = 0; i < nx; ++i)  
        src[i] = 1337. + i;  
}
```

3. Copy data from CPU to GPU

```
int main(int argc, char *argv[]) {  
    // allocate  
  
    initOnCPU(src, nx);  
  
    //          to      from size direction  
    cudaMemcpy(d_src, src, size, cudaMemcpyHostToDevice);  
}
```

4. Launch GPU kernels

```
__global__ void copyOnGPU(double *src, double *dest, size_t nx) {  
    size_t i = blockIdx.x * blockDim.x + threadIdx.x;  
  
    if (i < nx)  
        dest[i] = src[i] + 1;  
}
```

4. Launch GPU kernels

Denotes a kernel to be launched from the host and executed on the device

src and dest are device arrays

```
__global__ void copyOnGPU(double *src, double *dest, size_t nx) {  
    size_t i = blockIdx.x * blockDim.x + threadIdx.x;  
  
    if (i < nx)  
        dest[i] = src[i] + 1;  
}
```

Guard against out-of-bounds

Build-in thread variables

Each thread performs one update

4. Launch GPU kernels

```
__global__ void copyOnGPU(double *src, double *dest, size_t nx) {  
    size_t i = blockIdx.x * blockDim.x + threadIdx.x;  
  
    if (i < nx)  
        dest[i] = src[i] + 1;  
}  
  
int main(int argc, char *argv[]) {  
    // allocate, init and copy  
  
    auto numThreadsPerBlock = 256;  
    auto numBlocks = (nx + numThreadsPerBlock - 1) / numThreadsPerBlock;  
  
    copyOnGPU<<<numBlocks, numThreadsPerBlock>>>(d_src, d_dest, nx);  
}
```

Ceiling/ round-up division

4. Launch GPU kernels – Alternative grid-stride loop

```
__global__ void copyOnGPU(double *src, double *dest, size_t nx) {  
    size_t start = blockIdx.x * blockDim.x + threadIdx.x;  
    size_t stride = blockDim.x * blockDim.x;  
  
    for (size_t i = start; i < nx; i += stride)  
        dest[i] = src[i] + 1;  
}
```

```
int main(int argc, char *argv[]) {  
    // allocate, init and copy
```

```
    auto numThreadsPerBlock = 256;  
    auto numBlocks = 108 * 32;
```

Number of blocks is now decoupled from nx; can be tuned for the #SM

```
    copyOnGPU<<<numBlocks, numThreadsPerBlock>>>(d_src, d_dest, nx);
```

5. Do independent work on CPU

```
int main(int argc, char *argv[]) {  
    // allocate, init and copy  
  
    copyOnGPU<<<numBlocks, numThreadsPerBlock>>>(d_src, d_dest, nx);  
  
    // copyOnGPU runs asynchronously => additional CPU work could be done here
```


6. Synchronize GPU

```
int main(int argc, char *argv[]) {  
    // allocate, init and copy  
  
    copyOnGPU<<<numBlocks, numThreadsPerBlock>>>(d_src, d_dest, nx);  
  
    // copyOnGPU runs asynchronously => additional CPU work could be done here  
  
    cudaDeviceSynchronize();  
}
```

7. Copy data from GPU to CPU

```
int main(int argc, char *argv[]) {
    // allocate, init and copy

    copyOnGPU<<<numBlocks, numThreadsPerBlock>>>(d_src, d_dest, nx);

    // copyOnGPU runs asynchronously => additional CPU work could be done here

    cudaDeviceSynchronize();

    //          to   from   size  direction
    cudaMemcpy(dest, d_dest, size, cudaMemcpyDeviceToHost);
}
```

7. Copy data from GPU to CPU

```
int main(int argc, char *argv[]) {  
    // allocate, init and copy  
  
    copyOnGPU<<<numBlocks, numThreadsPerBlock>>>(d_src, d_dest, nx);  
  
    // copyOnGPU runs asynchronously => additional CPU work could be done here  
  
    cudaDeviceSynchronize();  
  
    //           to   from   size  direction  
    cudaMemcpy(dest, d_dest, size, cudaMemcpyDeviceToHost);  
}
```

Technically not necessary
since cudaMemcpy is blocking

8. Post-process data on CPU

```
int main(int argc, char *argv[]) {  
    // allocate, init and copy  
  
    copyOnGPU<<<numBlocks, numThreadsPerBlock>>>(d_src, d_dest, nx);  
  
    // copyOnGPU runs asynchronously => additional CPU work could be done here  
  
    cudaDeviceSynchronize();  
  
    //           to   from   size  direction  
    cudaMemcpy(dest, d_dest, size, cudaMemcpyDeviceToHost);  
  
    checkOnCPU(dest, nx);  
}
```

8. Post-process data on CPU

```
void checkOnCPU(double *dest, size_t nx) {  
    for (size_t i = 0; i < nx; ++i)  
        assert(1338. + i == dest[i]);  
}
```

```
int main(int argc, char *argv[]) {  
    // steps 1 through 6  
  
    //          to   from   size  direction  
    cudaMemcpy(dest, d_dest, size, cudaMemcpyDeviceToHost);  
  
    checkOnCPU(dest, nx);  
}
```

8. Post-process data on CPU

```
int main(int argc, char *argv[]) {  
    // steps 1 through 7  
  
    checkOnCPU(dest, nx);  
  
    // de-allocate _device_ arrays  
    cudaFree(d_src);  
    cudaFree(d_dest);  
  
    // de-allocate _host_ arrays  
    cudaFreeHost(src);  
    cudaFreeHost(dest);  
  
    return 0;  
}
```

Running Applications

- Compilation with either
 - NVCC
 - ‘old’ CUDA compiler, relays CPU code to host compiler (e.g. GCC)
 - `nvcc -arch=sm_80 -O3 -o my-cuda-app my-cuda-app.cu`
 - `-arch` sets the GPU to optimize for (https://en.wikipedia.org/wiki/CUDA#GPUs_supported)
 - or NVC++
 - C++ and CUDA compiler that also supports OpenACC, OpenMP, `std::par`, ...
 - `nvc++ -O3 -o my-cuda-app my-cuda-app.cu`
 - Auto-detects current GPU – compile on the system you want to execute on
- Execution as usual: `./my-cuda-app`

Profiling Applications

- Getting information about available hardware
 - `nvidia-smi`
- Profiling with results on command line
 - `nsight systems`
 - `nsys profile --stats=true \`
`./my-cuda-app`
 - `nsight compute`
 - `ncu ./my-cuda-app`
- Profiling with result files
 - `nsight systems`
 - `nsys profile -o output-file \`
`./my-cuda-app`
 - `nsight compute`
 - `ncu -o output-file \`
`./my-cuda-app`
- Open generated profiles with the corresponding GUIs

GPU Programming & Performance

GPU Benchmarks



Benchmarks

- Great work by Dominik Ernst: <https://github.com/te42kyfo/gpu-benches>

Aside: Latency

- Connecting parallelism, latency and bandwidth – Little's Law
 - $L \cdot b_s = N$, where
 - L is the latency, b_s is the performance, i.e. bandwidth, and N the number of customers, i.e. the total data volume requested
- Example: Execution time of a single thread at low occupancy on A100
 - $$L = \frac{64 \frac{\text{threads}}{\text{SM}} \cdot 108 \text{ SM} \cdot 16 \frac{\text{B}}{\text{thread}}}{157 \text{ GB/s}} \cdot F = 704 \text{ ns} \cdot 1.38 \text{ GHz} = 972 \text{ cyc}$$

Memory Coalescing

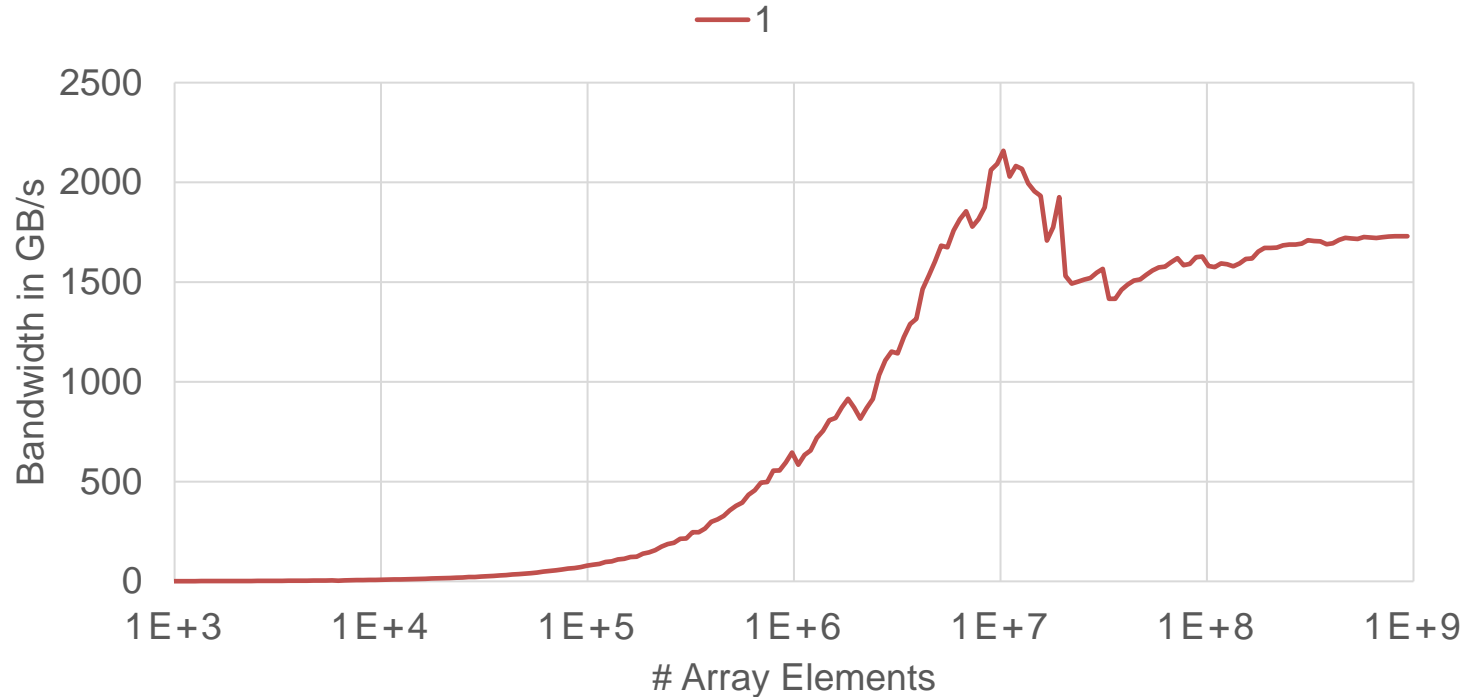
- Simple copy benchmark with strided access pattern

```
__global__ void copy(double *src, double *dest, size_t nx, int stride) {  
    size_t i = blockIdx.x * blockDim.x + threadIdx.x;  
  
    if (i < nx)  
        dest[stride * i] = src[stride * i];  
}
```

- All results are obtained on a single A100 80GB GPU in Alex

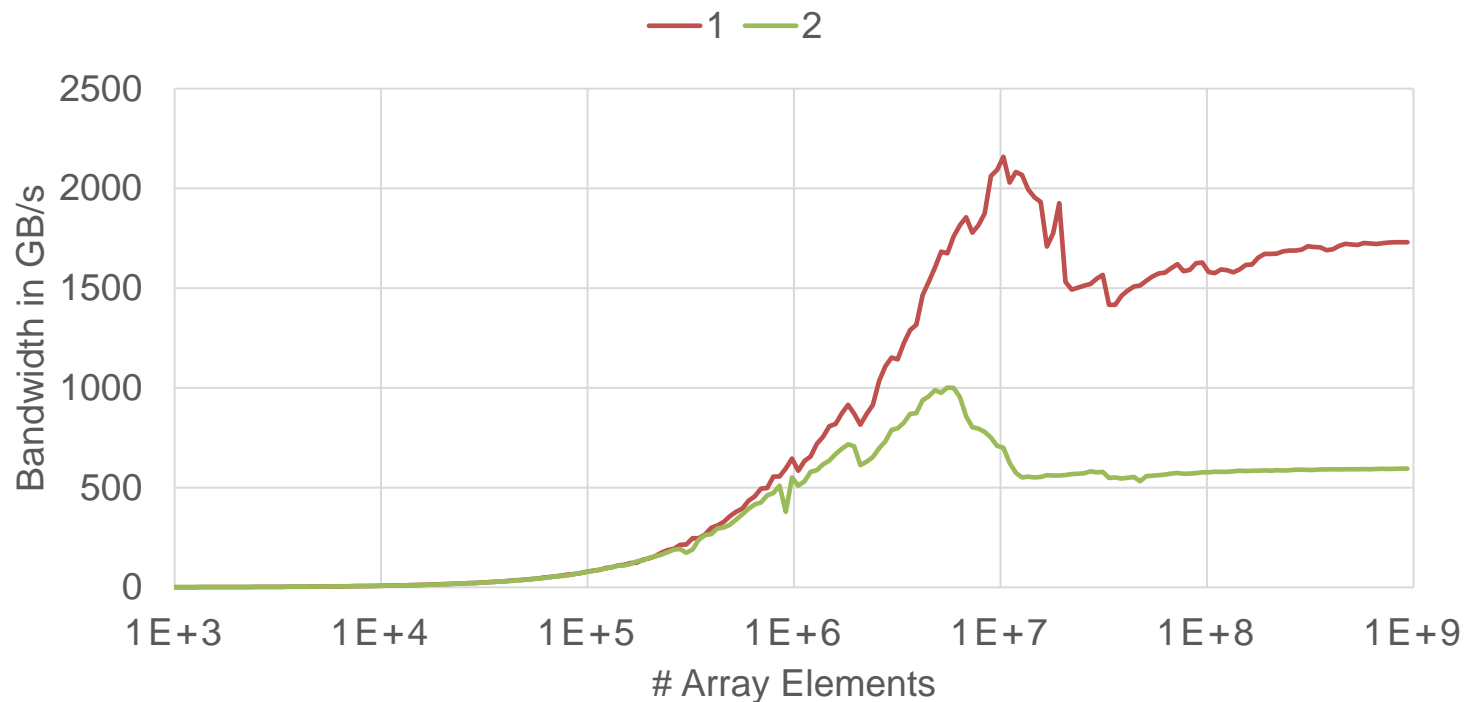
Memory Coalescing

- Simple copy benchmark with strided access pattern



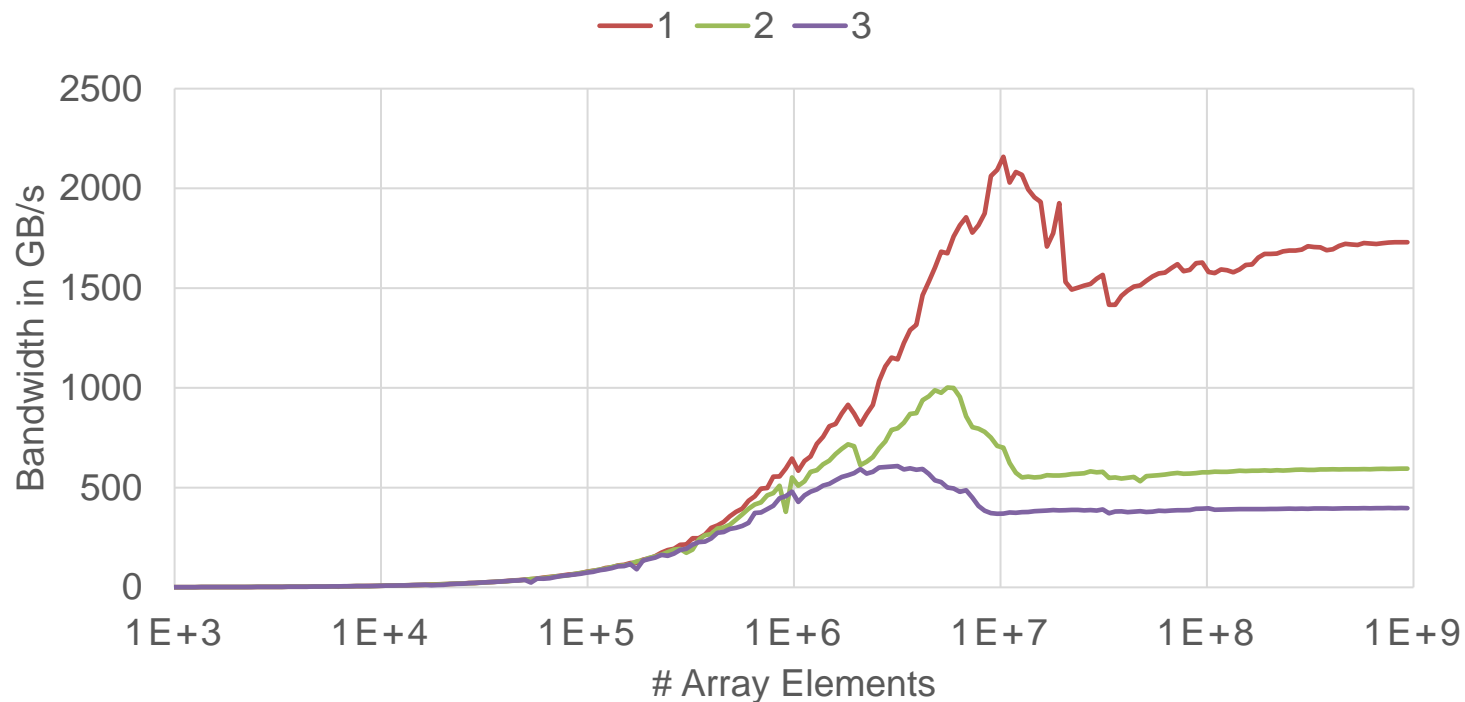
Memory Coalescing

- Simple copy benchmark with strided access pattern



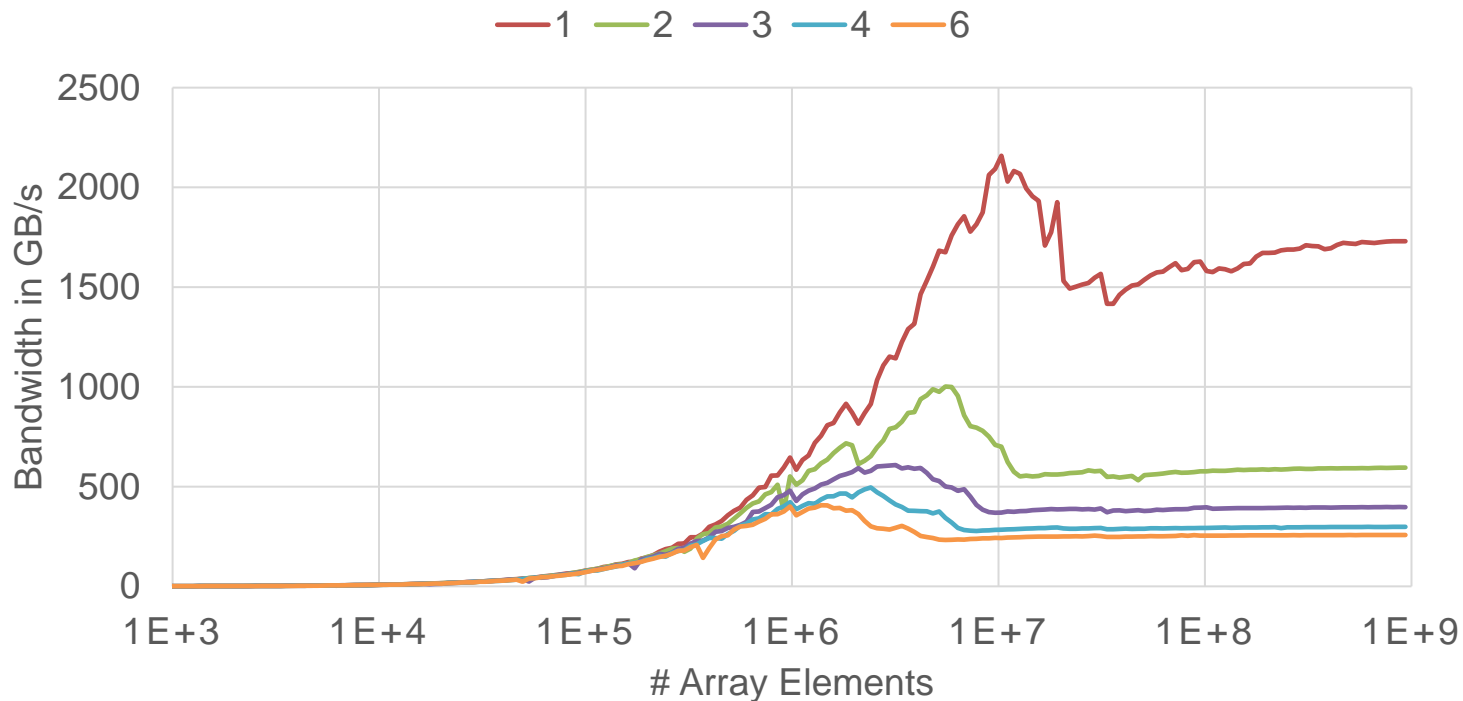
Memory Coalescing

- Simple copy benchmark with strided access pattern



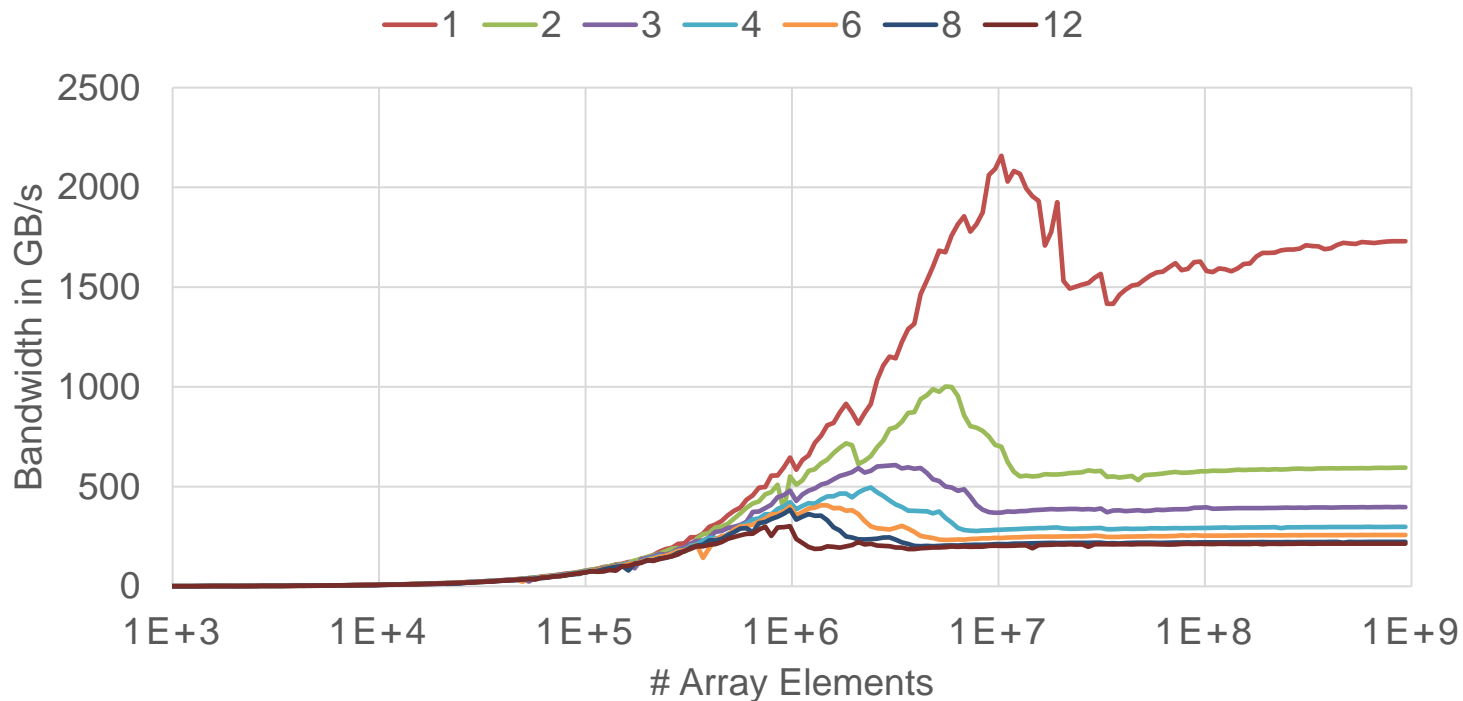
Memory Coalescing

- Simple copy benchmark with strided access pattern



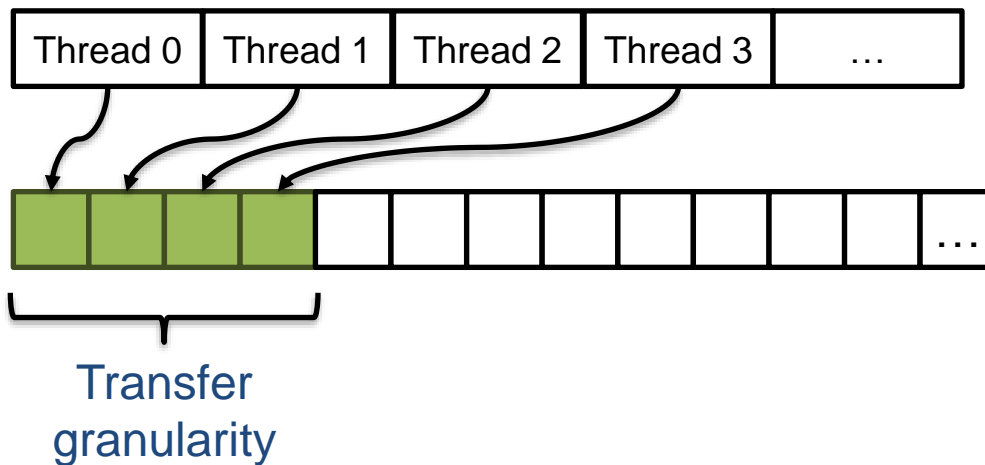
Memory Coalescing

- Simple copy benchmark with strided access pattern



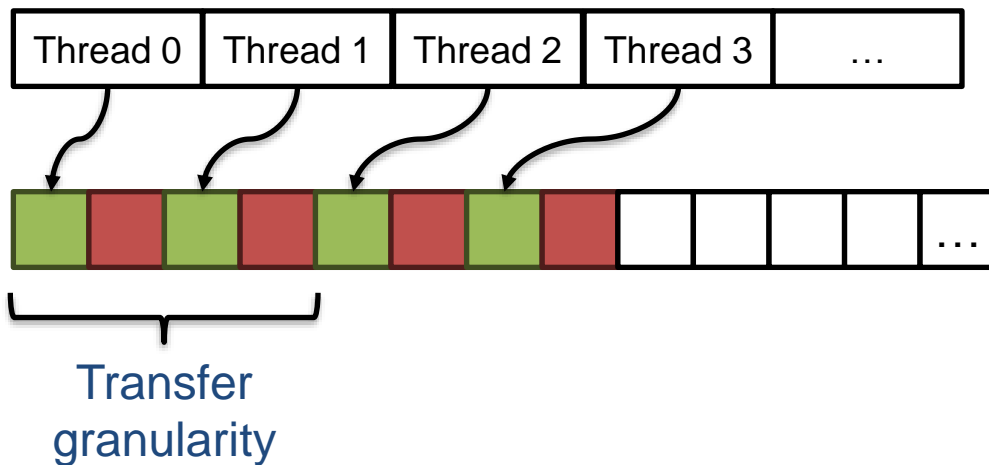
Memory Coalescing

- Coalesced access: consecutive threads access consecutive memory locations



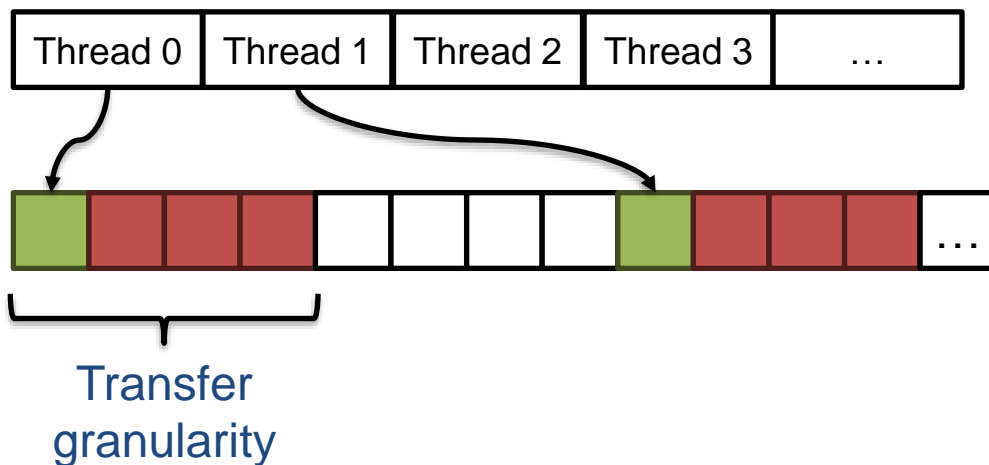
Memory Coalescing

- Uncoalesced access: additional, unused data has to be transferred
- Granularity depends on GPU and on memory space (DRAM, L2, L1, ...)



Memory Coalescing

- Uncoalesced access: additional, unused data has to be transferred
- Granularity depends on GPU and on memory space (DRAM, L2, L1, ...)



GPU Programming & Performance

OpenMP Target Offloading

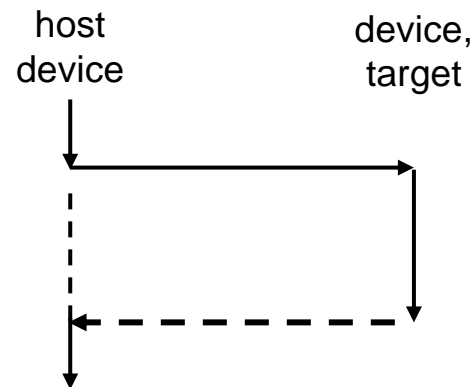


OpenMP Target Offloading

- OpenMP supports GPU offloading since version 4.0
- Remember: OpenMP standard != compiler implementation
- GPUs of all major vendors are supported
 - But performance may drastically vary from GPU to GPU and from compiler to compiler
- The following slides are based on *Introduction to OpenMP*, a course offered by NHR@FAU
- The full course covers concepts in more depth

Introduction

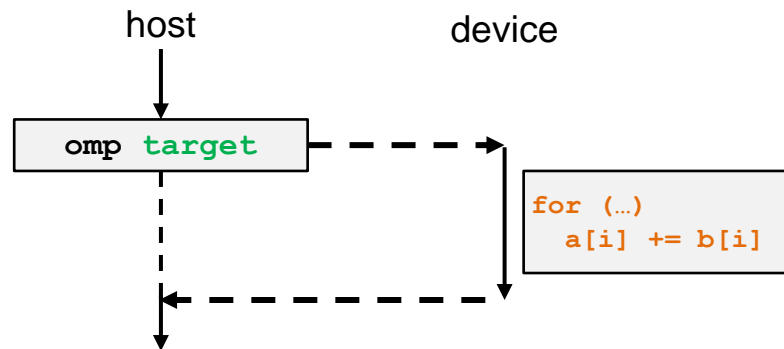
- Execute code on a device
 - Not necessarily a GPU, can also be an FPGA, DSP, ...
- **Target:** device where code and data is offloaded to
- Execution always starts on the **host** device



Offloading Code to the Target

- **target**: execute associated structured block on the device
- Execution on the target is initially single threaded
- Host waits until offloaded code completes

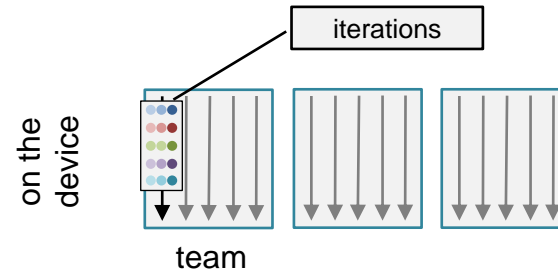
```
int a[1024], b[1024];  
/* init a and b */  
#pragma omp target  
{  
  for (int i = 0; i < 1024; ++i)  
    a[i] += b[i];  
} /* wait until complete */
```



Generating Parallelism on the Target

- Target construct alone does not generate parallelism

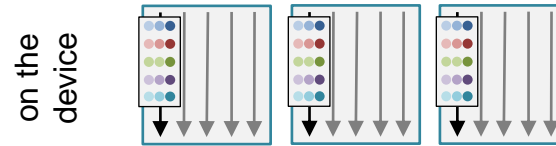
```
#pragma omp target  
for (int i = 0; i < 1024; ++i)  
    a[i] += b[i];
```



Generating Parallelism on the Target

- `teams` construct
 - generate **league of teams**
 - a team has only one initial thread
 - each team executes the same code

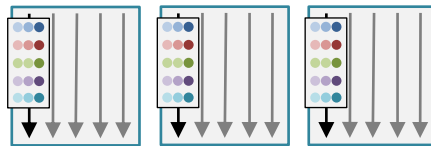
```
#pragma omp target teams  
for (int i = 0; i < 1024; ++i)  
    a[i] += b[i];
```



Generating Parallelism on the Target

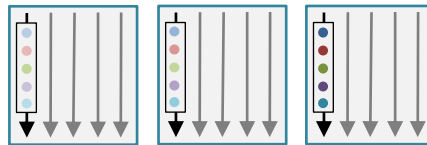
- **teams** construct
 - generate **league of teams**
 - a team has only one initial thread
 - each team executes the same code

```
#pragma omp target teams  
for (int i = 0; i < 1024; ++i)  
    a[i] += b[i];
```



- **distribute** construct
 - distributes iteration space of associated loop(s) over teams

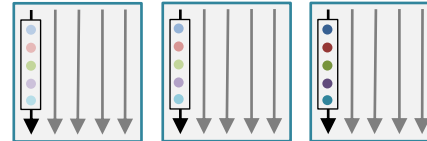
```
#pragma omp target teams distribute  
for (int i = 0; i < 1024; ++i)  
    a[i] += b[i];
```



Generating Parallelism on the Target

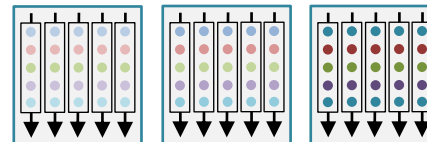
- **distribute** construct
 - Distributes iteration space of associated loop(s) over teams

```
#pragma omp target teams distribute
for (int i = 0; i < 1024; ++i)
    a[i] += b[i];
```



- **parallel** construct
 - Generate parallel region with multiple threads inside each team

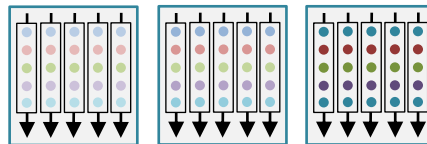
```
#pragma omp target teams distribute \
    parallel
for (int i = 0; i < 1024; ++i)
    a[i] += b[i];
```



Generating Parallelism on the Target

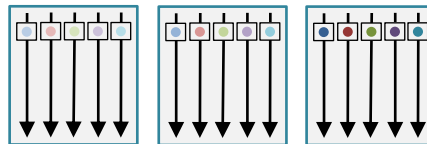
- `parallel` construct
 - Generate parallel region with multiple threads inside each team

```
#pragma omp target teams distribute \  
    parallel \  
for (int i = 0; i < 1024; ++i) \  
    a[i] += b[i];
```



- `for` construct
 - Worksharing loop
 - Distribute team's iteration space over all threads inside a team

```
#pragma omp target teams distribute \  
    parallel for \  
for (int i = 0; i < 1024; ++i) \  
    a[i] += b[i];
```



Generating Parallelism on the Target

- Additionally: `simd` construct – use SIMD lanes in each thread
- How each directive is mapped to GPU entities depends on the compiler
- Specializing the parallelization is possible (`num_teams`, `thread_limit`, ...)
- Reductions are supported as usual

Data Mapping

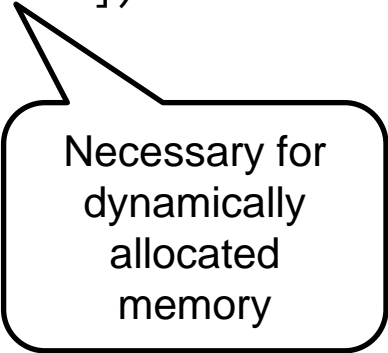
- Mapping data consists of three phases
 1. map-enter: on entry to target region
 - Storage allocation and copy of mapped variable to device
 2. Compute: execution of target region, device threads access mapped variable
 3. map-exit: on exist of target region, variable is unmapped from device
 - Copy of corresponding variable to host, deallocation

Data Mapping

- Mapping variables, etc. can be done implicit, or explicit, e.g.

```
#pragma omp target map(tofrom : a) map(to : b[0 : nx])  
// block
```

- Mapping causes a presence check
 - Copy to device only if not already present



Necessary for
dynamically
allocated
memory

Data Mapping

```
// Three options
// map as part of target region
#pragma omp target      map(tofrom : a) map(to : b[0 : nx])
{ /* ... */ }

// target data region
#pragma omp target data map(tofrom : a) map(to : b[0 : nx])
{
    #pragma omp target
    { /* ... */ }
}

// target data enter and exit
#pragma omp target data enter map(to : a) map(to : b[0 : nx])
/* ... */
#pragma omp target data exit map(from : a)
```

GPU Programming & Performance

Case Study: Dense Matrix Vector Multiplication

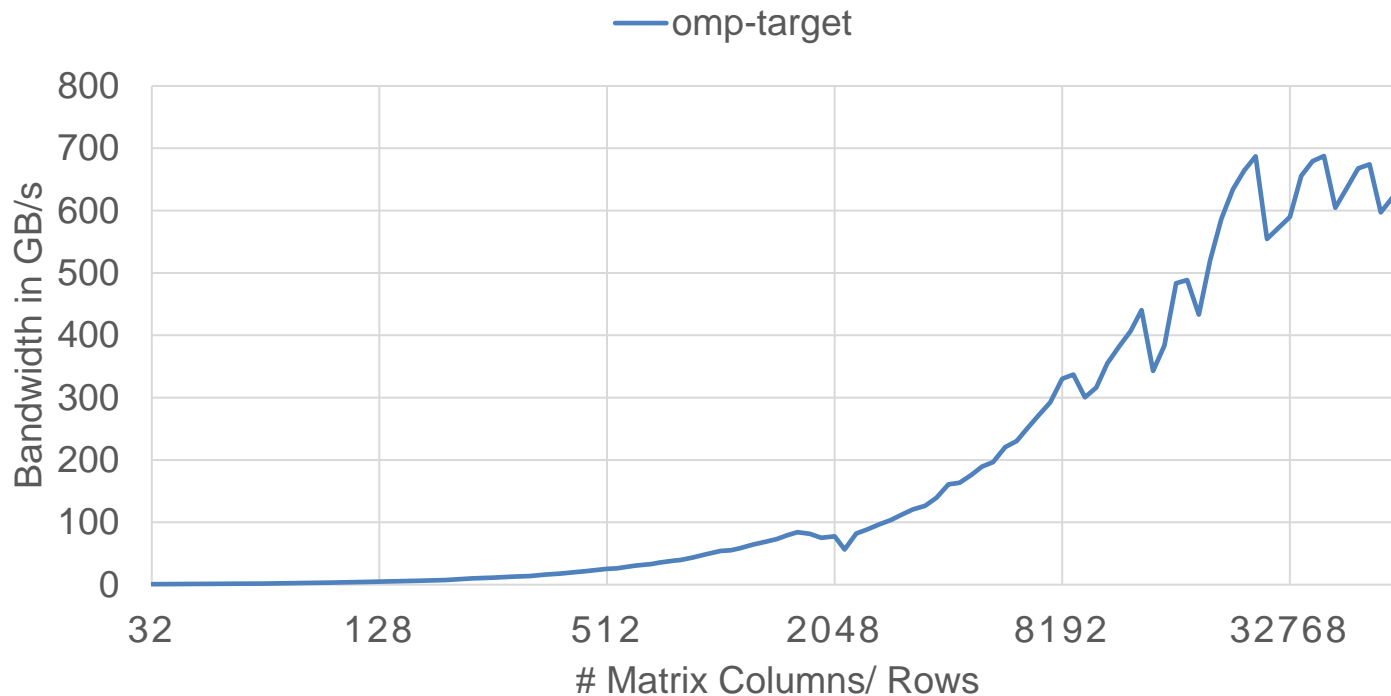


Case Study dMVM

- Live Demo
- All results are obtained on a single A100 80GB GPU in Alex

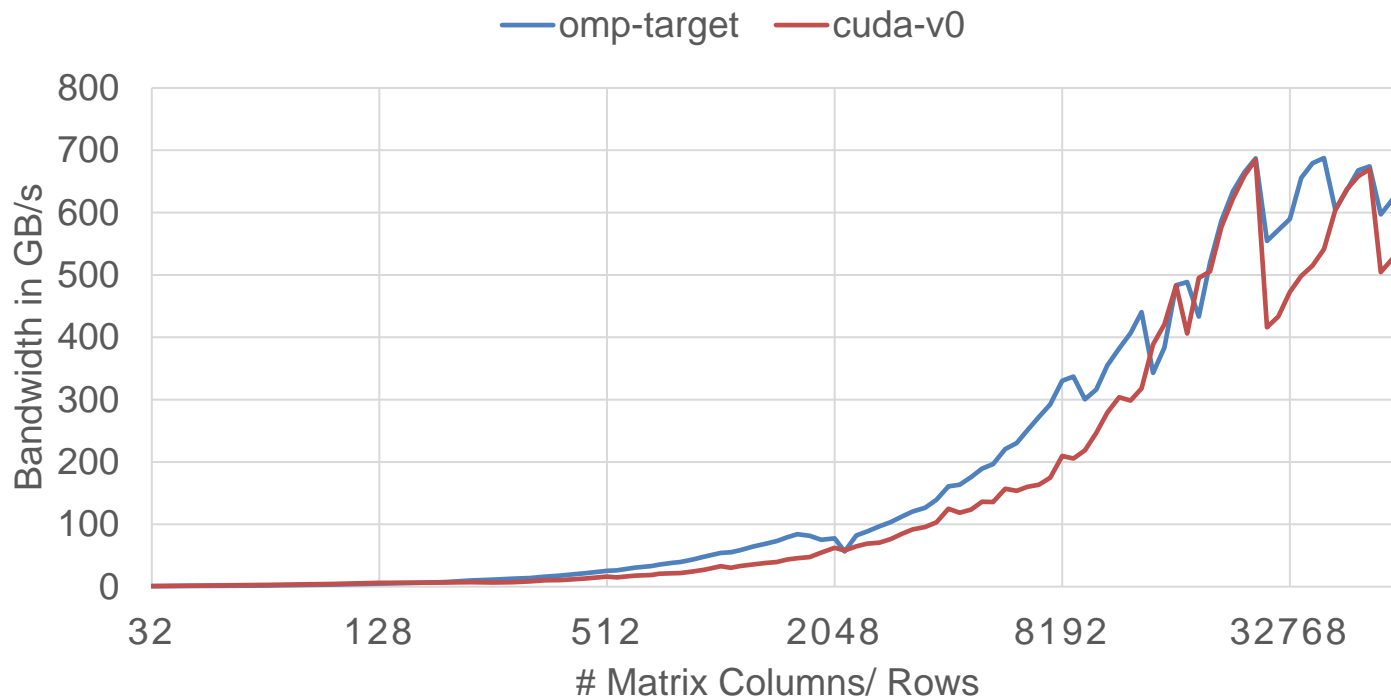
Case Study dMVM

- First Attempt: OpenMP target offload version



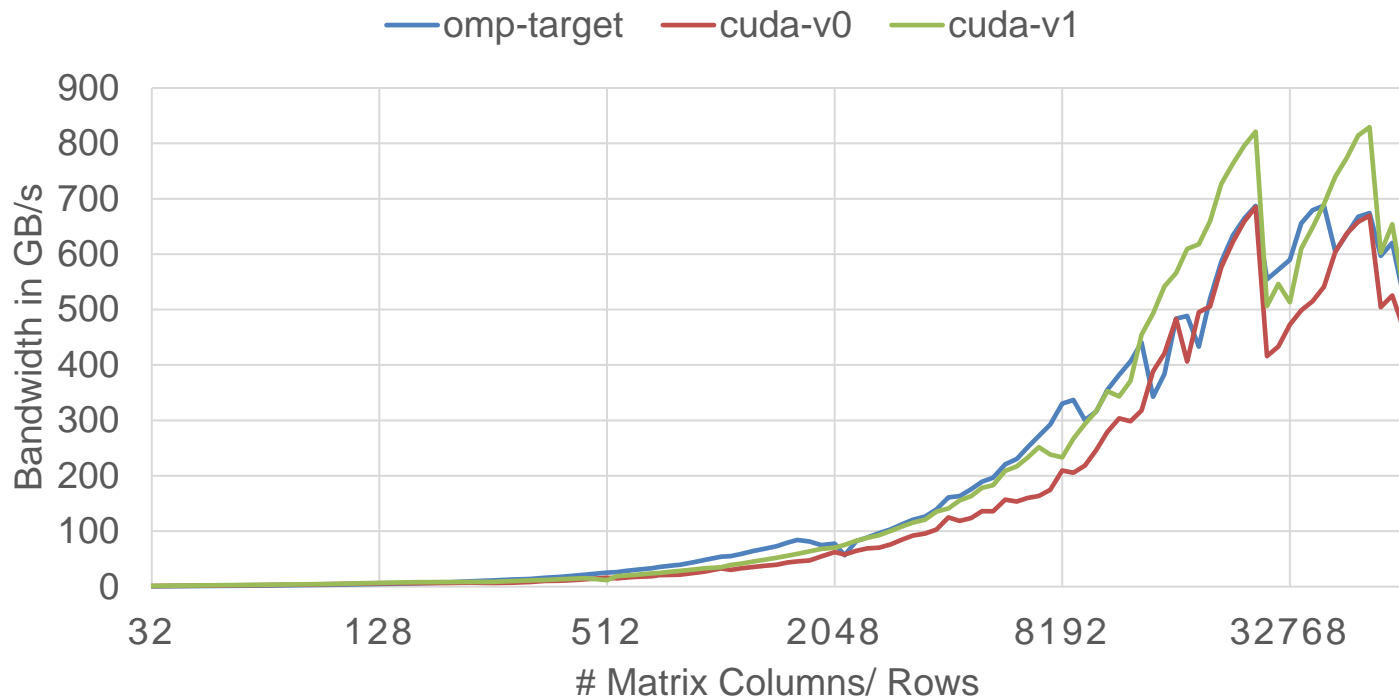
Case Study dMVM

- OpenMP target offload version shows subpar resource utilization
- Transition to CUDA



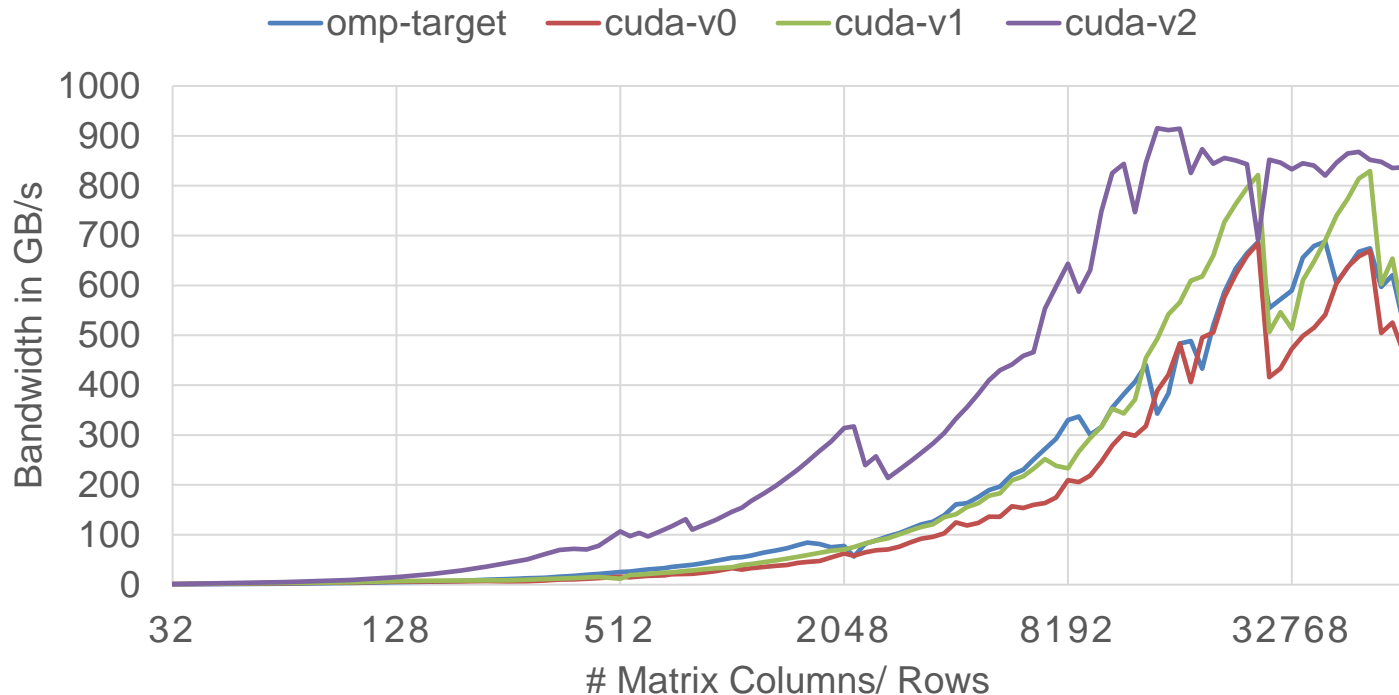
Case Study dMVM

- CUDA v0 frequently reads from/ writes to global memory (may be cached)
- Use local variable to minimize traffic



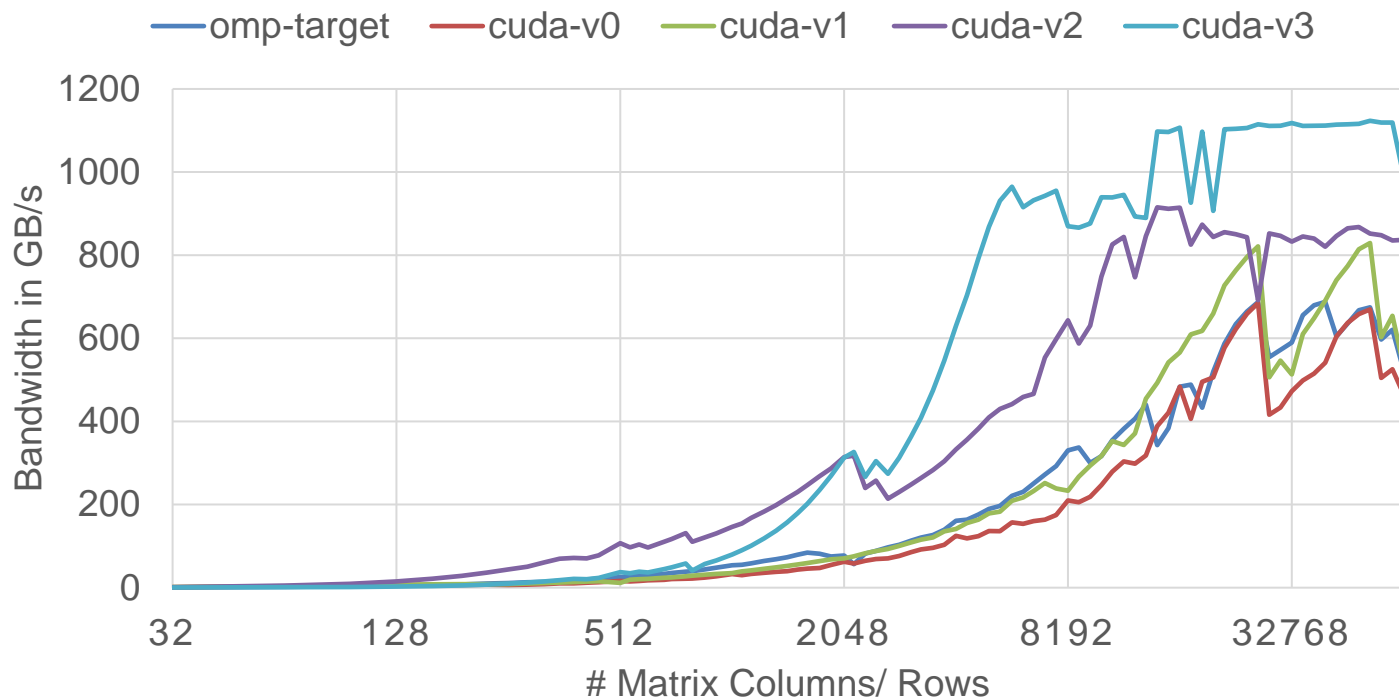
Case Study dMVM

- CUDA v0/1 suffer from a small number of threads
- Switch execution configuration to 2D and use atomics to avoid data races



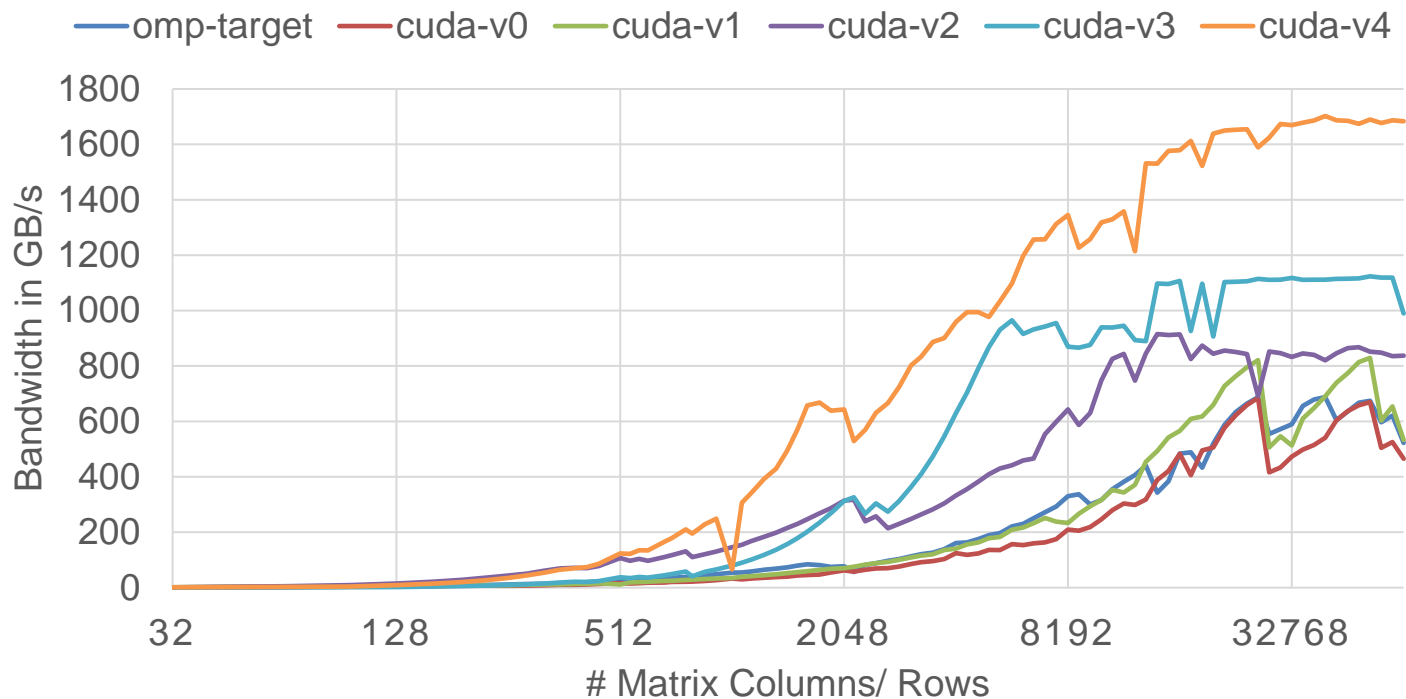
Case Study dMVM

- CUDA v2 experiences atomic congestion and high memory traffic
- Compute partial results and add grid-stride loops



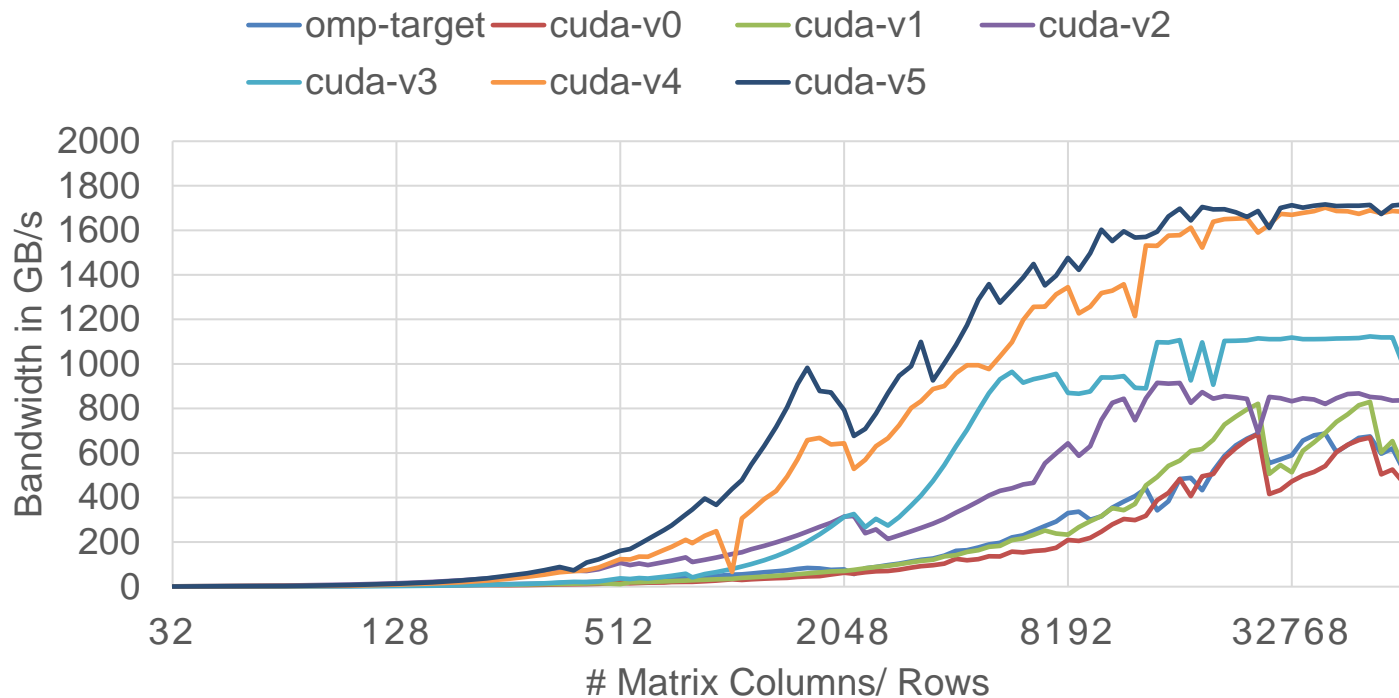
Case Study dMVM

- CUDA v3 writes coalesced but reads uncoalesced
- Transpose execution configuration



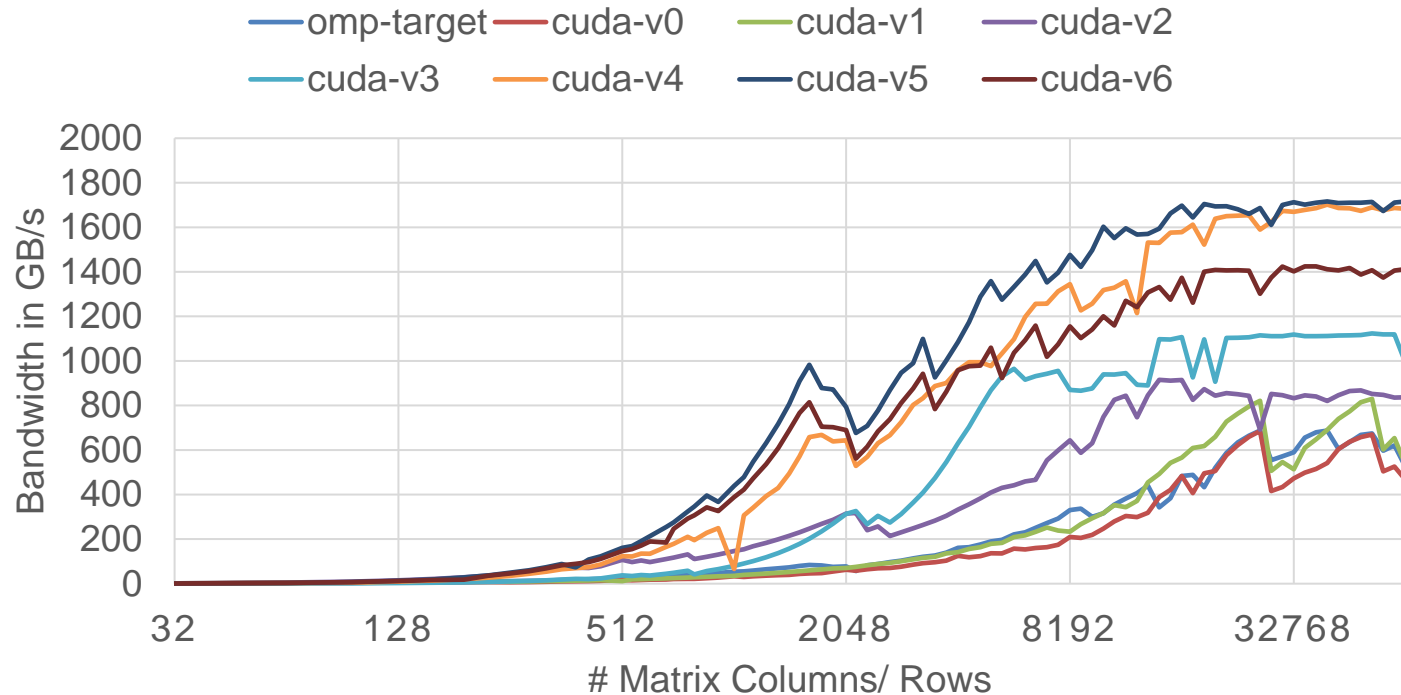
Case Study dMVM

- CUDA v4 doesn't aggregate results within CUDA blocks
- Use shared memory to buffer and aggregate output data



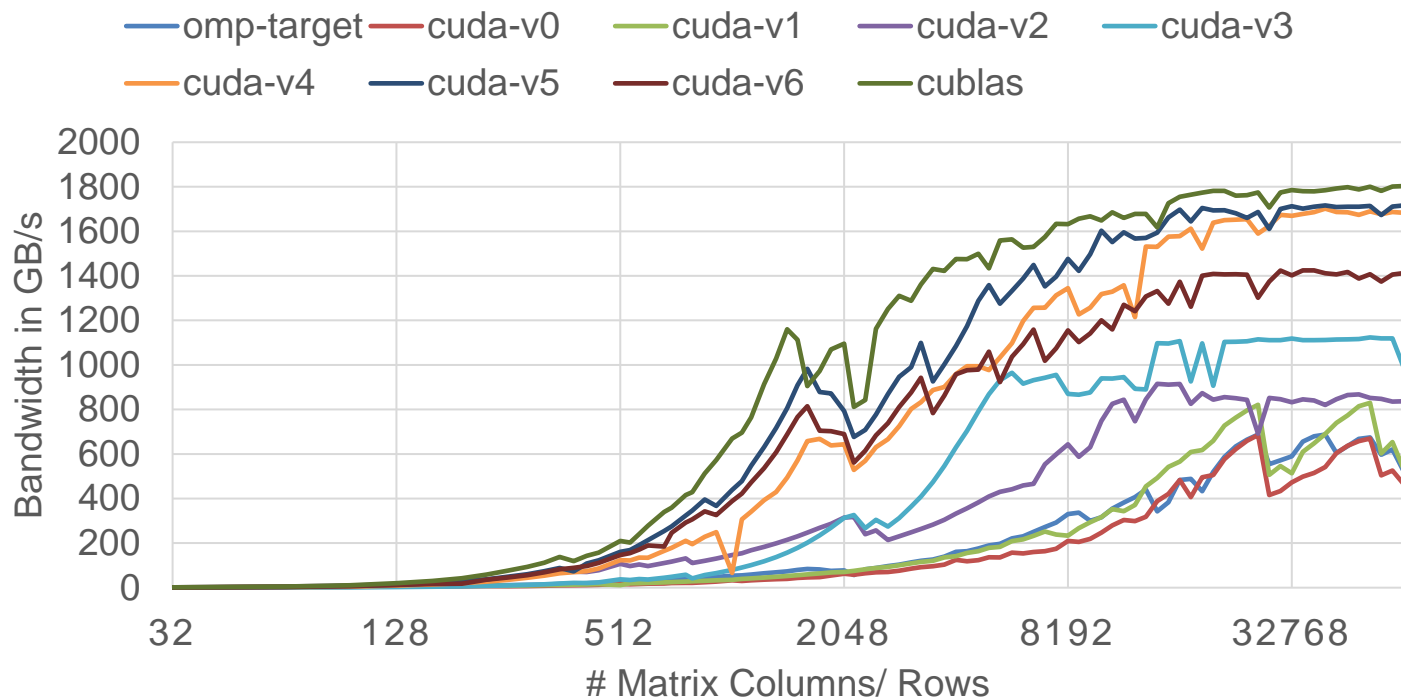
Case Study dMVM

- CUDA v5 doesn't buffer and share input data
- Use shared memory to buffer input data



Case Study dMVM

- CUDA v6 is slower (and becomes quite complex/ hard to maintain & tune)
- Use library implementation (cuBLAS) instead



GPU Programming & Performance

Outlook



Recommended Implementation Process

1. Implement CPU-only application, check correctness
2. Revise memory management to allocate GPU and CPU memory
3. Transform CPU functions to GPU kernels with grid-stride loops
 1. Execute serially
 2. Execute parallel
4. Repeat step 3 as necessary, then optimize
 1. Remove unnecessary synchronization
 2. Remove unnecessary copies

Recommended Implementation Process

5. Profile and optimize

- Nsight systems to get whole application view
- Nsight compute to zero in on performance behavior of single kernels
- Try to optimize for bottlenecks, e.g.
 - Long PCIe transfer periods – try to overlap computation and data transfers
 - Short kernel runtimes – try to fuse multiple kernels
 - Not enough parallelism – try running multiple kernels concurrently (c.f. CUDA streams)

Aside: Managed Memory

- 'Classic'
 - Distinct allocations for host and device
 - Explicit copies from H-D/ D-H
- Managed
 - One unified allocation shared between host and device
 - Migration between host and device on access
 - Small migration granularity (e.g. page size)

Outlook

- New HPC GPU systems with shared physical memory
 - NVIDIA GraceHopper super chip
 - AMD MI300A APU
 - Might require different ways of programming (for full performance)
- Additional resources
 - NHR courses (<https://hpc.fau.de/teaching/tutorials-and-courses/>)
 - High-End Simulation in Practice (HESP) by Prof. Köstler
 - GTC (<https://www.nvidia.com/gtc/>)