Programming Techniques for Supercomputers

Friedrich-Alexander-Universität
Erlangen-Nürnberg

GPU Programming – Performance

Sebastian Kuckuk

Erlangen National High Performance Computing Center (NHR@FAU)

NHR FAU

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

GPU Programming

GPU Benchmarks

Performance Modelling – General Approach

1. Obtain relevant metrics for relevant code regions

   - Execution time
   - Bytes read/ written
   - FLOPS performed
   - And many more …

   Somewhere between kernels/ loops and whole application; usually limited to ‘meaningful’ work

2. Relate observed performance to theoretical limits
   - … or to measured limits – see next slides

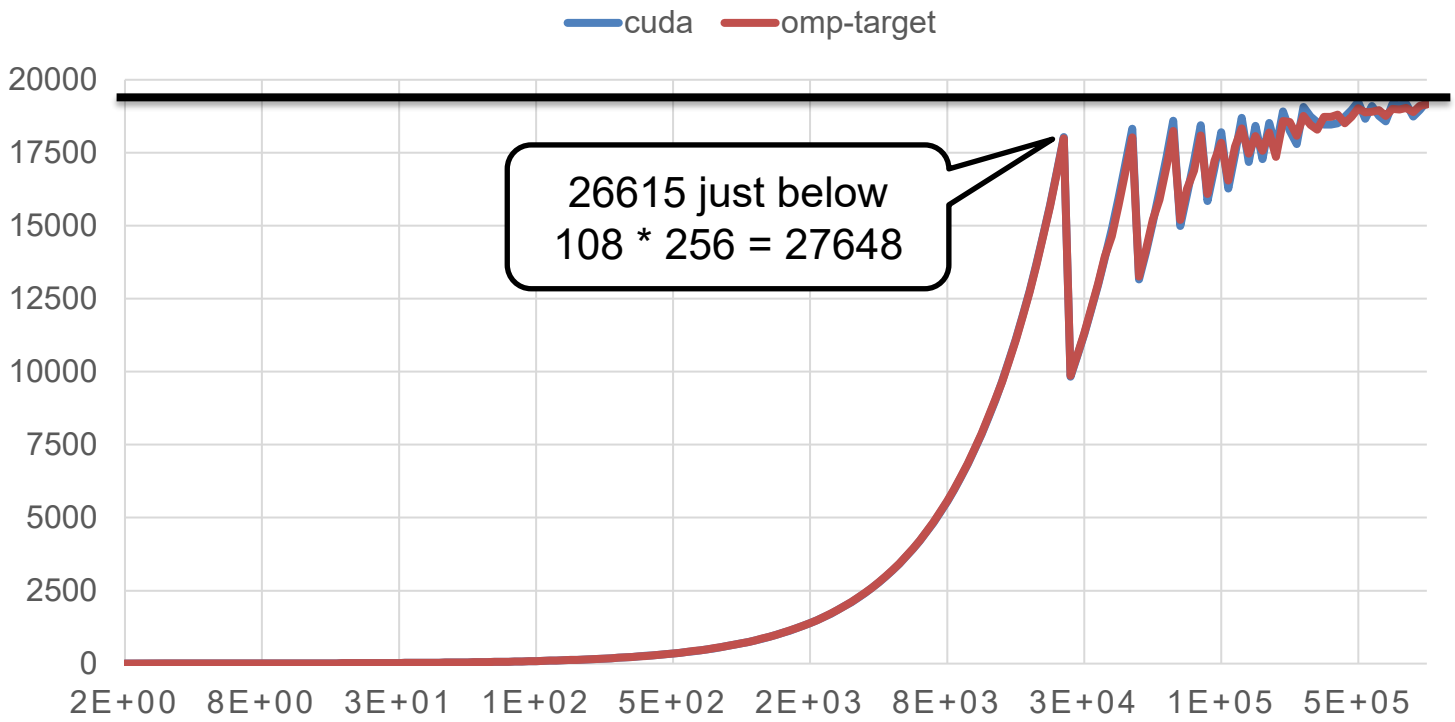
3. Figure out where optimization is possible/ reasonable
   - Try to hit at least one bottleneck, then shift to another bottleneck

Micro Benchmarks

- Motivation: theoretical limits can be too optimistic, micro benchmarks give a more realistic expectation
- Two examples: computational performance and sustained bandwidth
- This also represents a manual approach to performance modelling – doing explicit timing combined with manual code analysis
- The following results are obtained on a single A100 80GB GPU in Alex

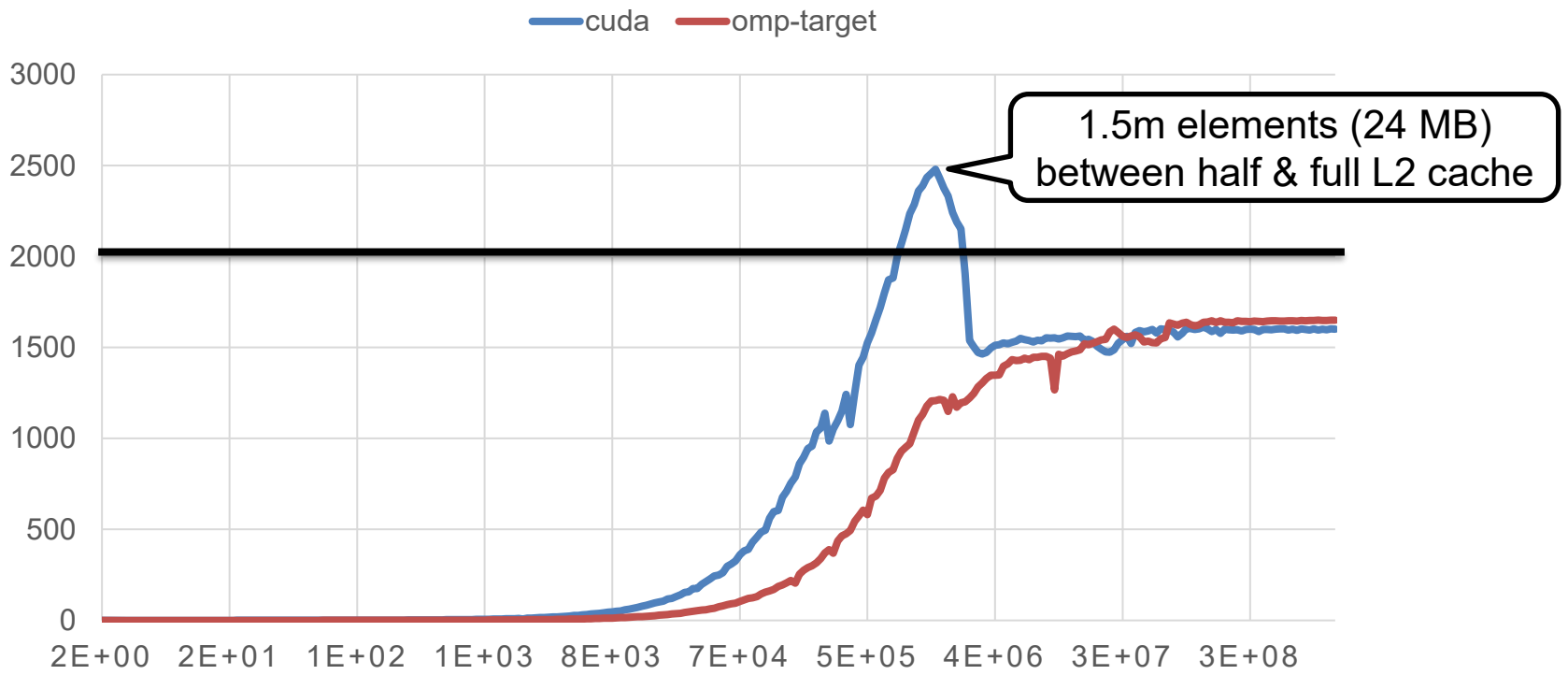
FMA Benchmark

A100, FMA, GFLOP/s over number of computational elements



Stream Benchmark

A100, stream, bandwidth in GB/s over number of array elements

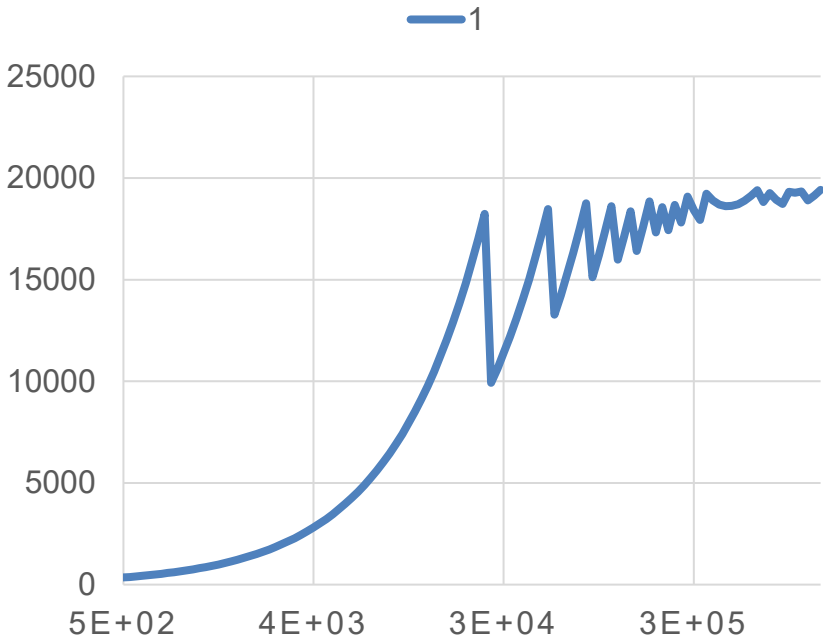


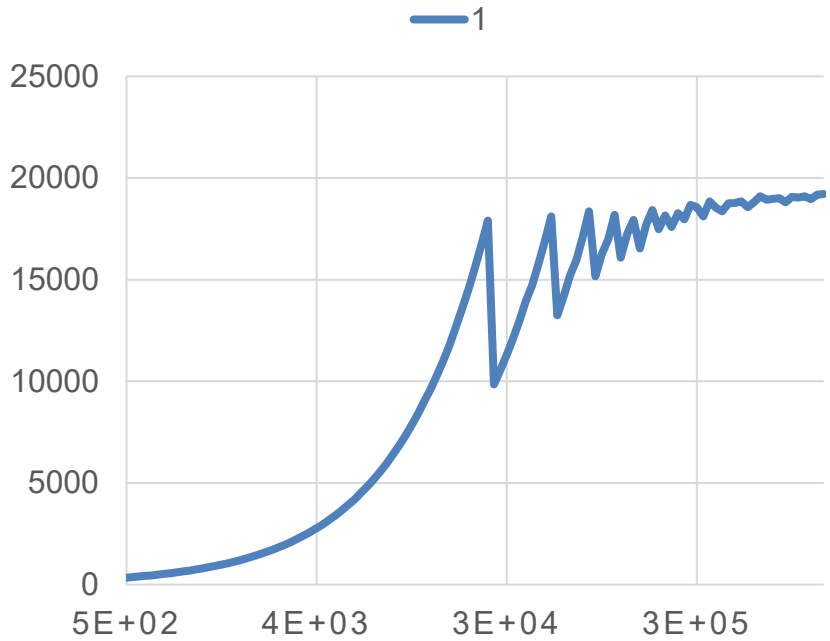
Benchmark Evaluation

- FMA performance is close to theoretical peak
- Stream performance is not perfect
  - Potentially different results for different data access patterns (#load and #store operations per thread)
  - Potentially different results for different data types
- What about non-uniform operations?
  - Strided Stream – each thread accesses data with an added stride
  - Strided FMA – each thread that is divisible by the stride computes

Strided FMA Benchmark

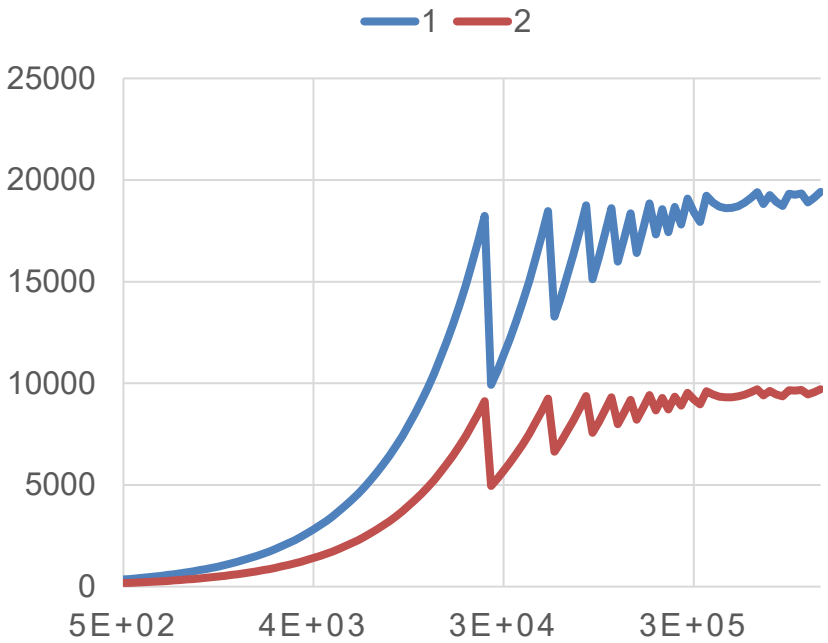
A100, strided FMA, GFLOP/s over number of computational elements
CUDA (left) and OpenMP target (right)

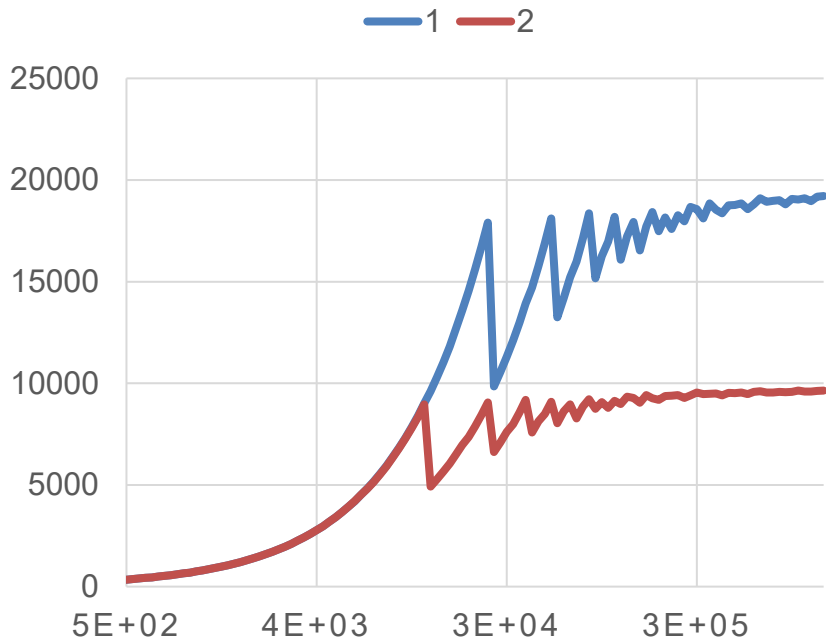




Strided FMA Benchmark

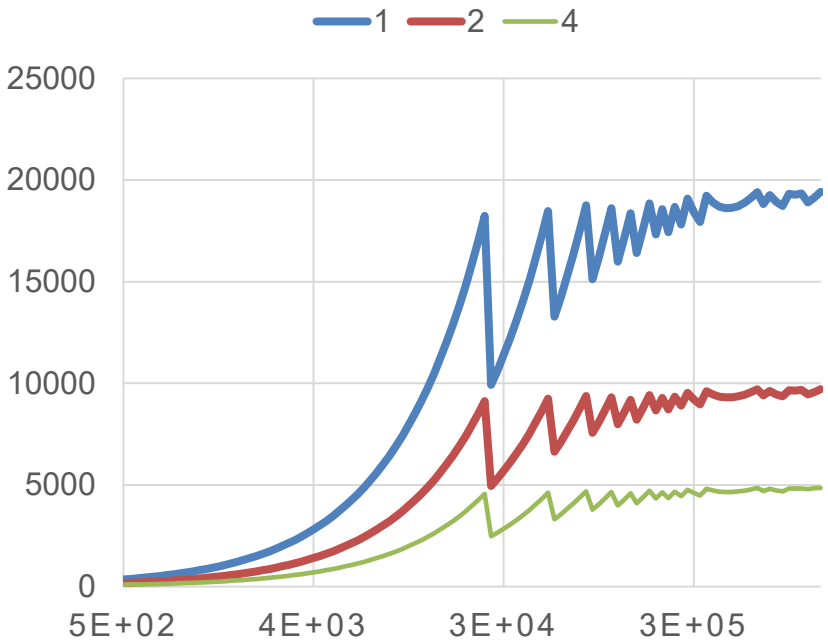
A100, strided FMA, GFLOP/s over number of computational elements
CUDA (left) and OpenMP target (right)

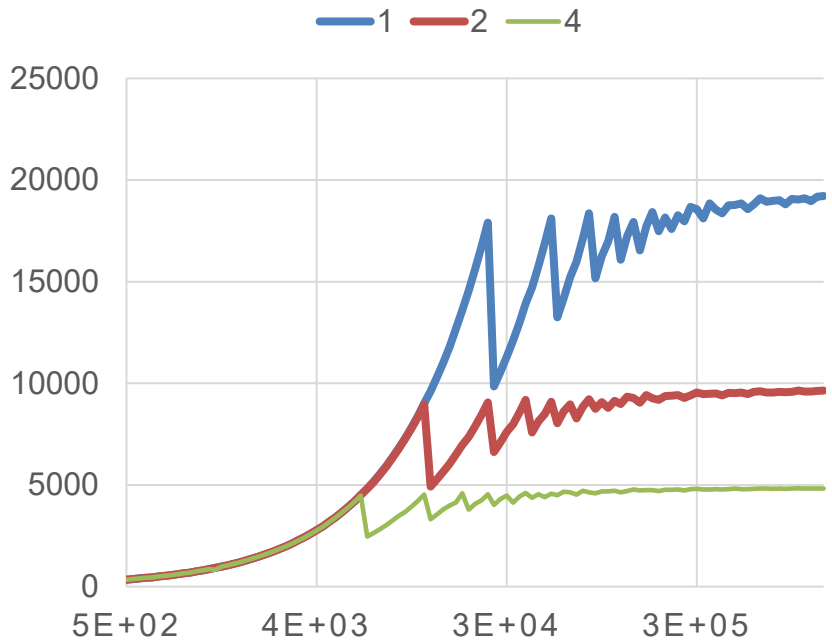




Strided FMA Benchmark

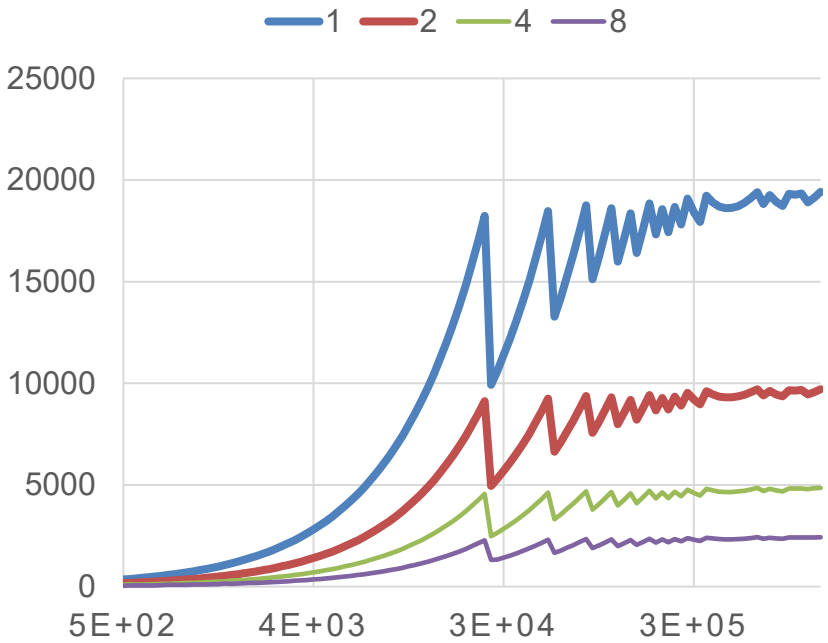
A100, strided FMA, GFLOP/s over number of computational elements
CUDA (left) and OpenMP target (right)

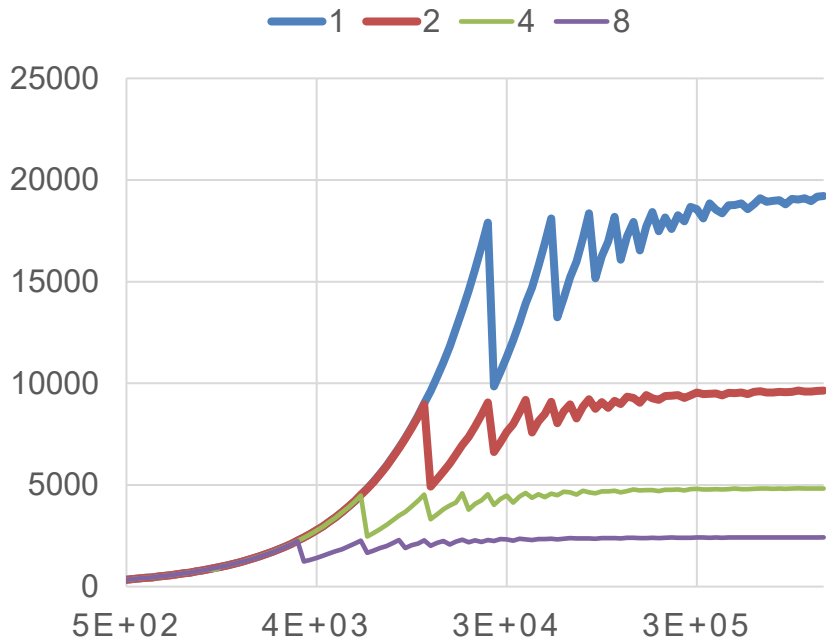




Strided FMA Benchmark

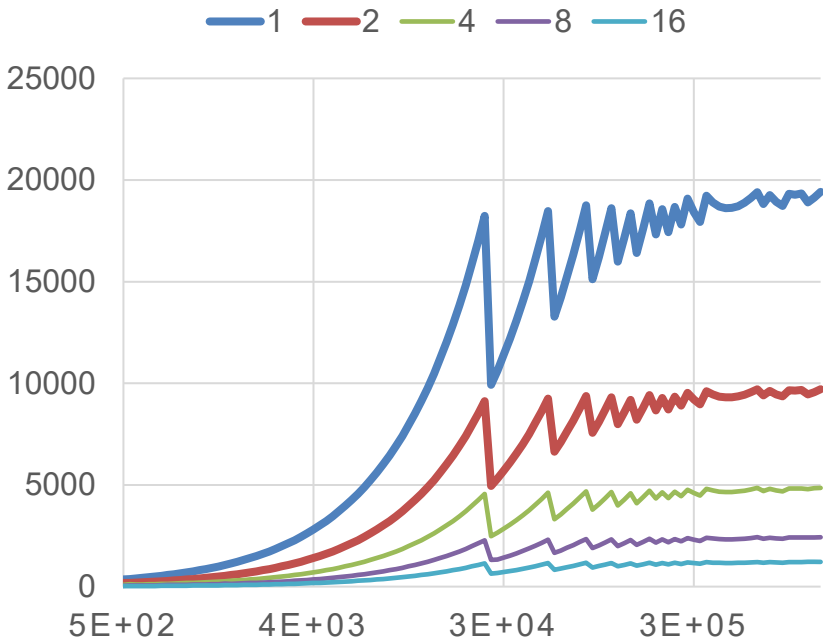
A100, strided FMA, GFLOP/s over number of computational elements
CUDA (left) and OpenMP target (right)

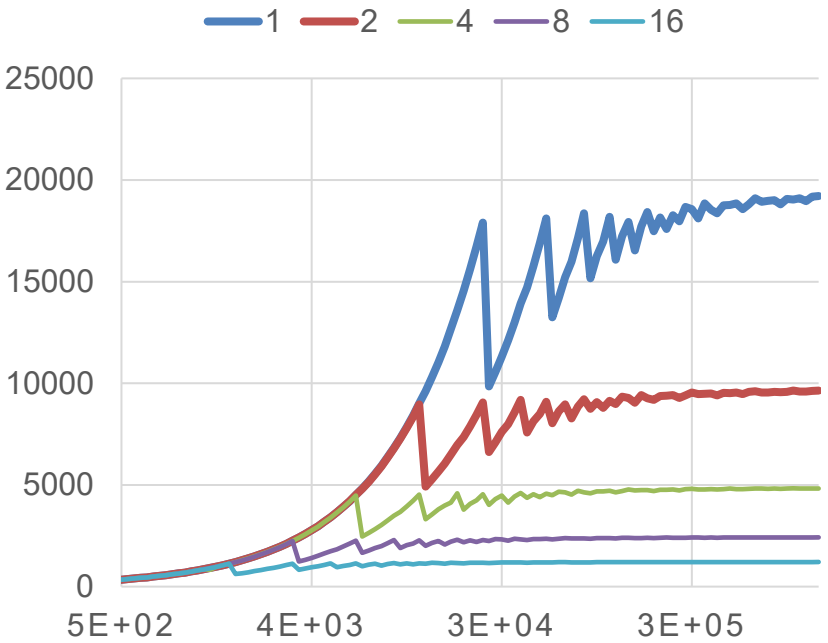




Strided FMA Benchmark

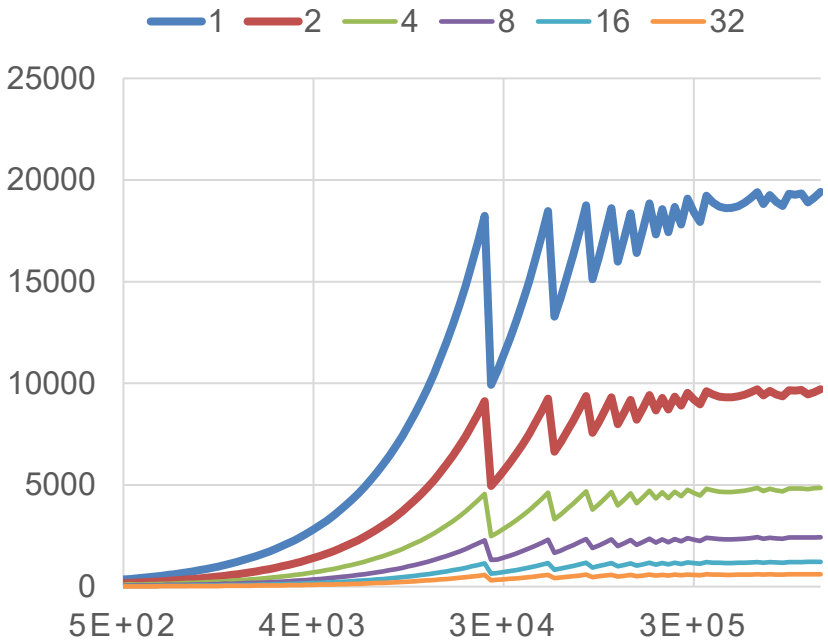
A100, strided FMA, GFLOP/s over number of computational elements
CUDA (left) and OpenMP target (right)

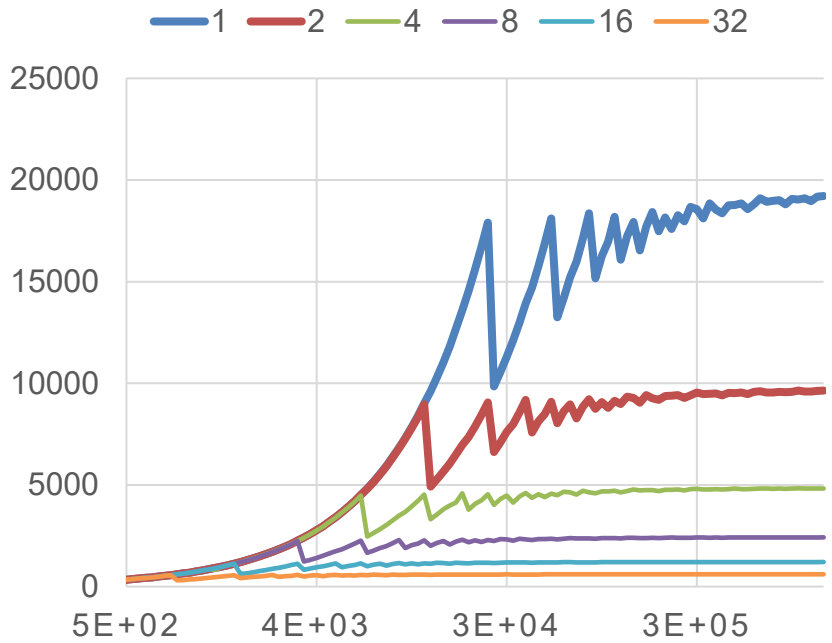




Strided FMA Benchmark

A100, strided FMA, GFLOP/s over number of computational elements
CUDA (left) and OpenMP target (right)





Memory Coalescing

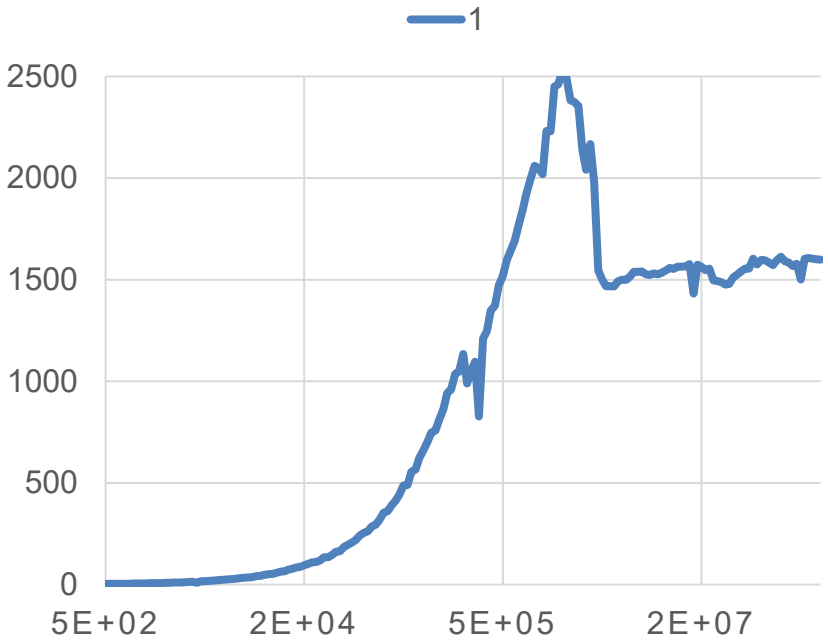
- Simple copy benchmark with strided access pattern

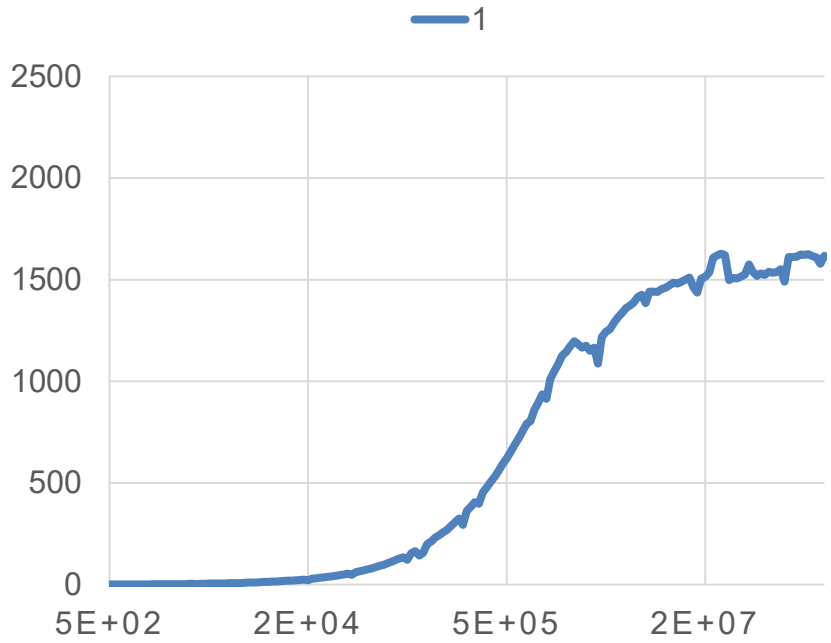
```
__global__ void copy(double *src, double *dest, size_t nx, int stride) {
  size_t i = blockIdx.x * blockDim.x + threadIdx.x;

  if (i < nx)
    dest[stride * i] = src[stride * i];
}
```

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

Strided Stream Benchmark

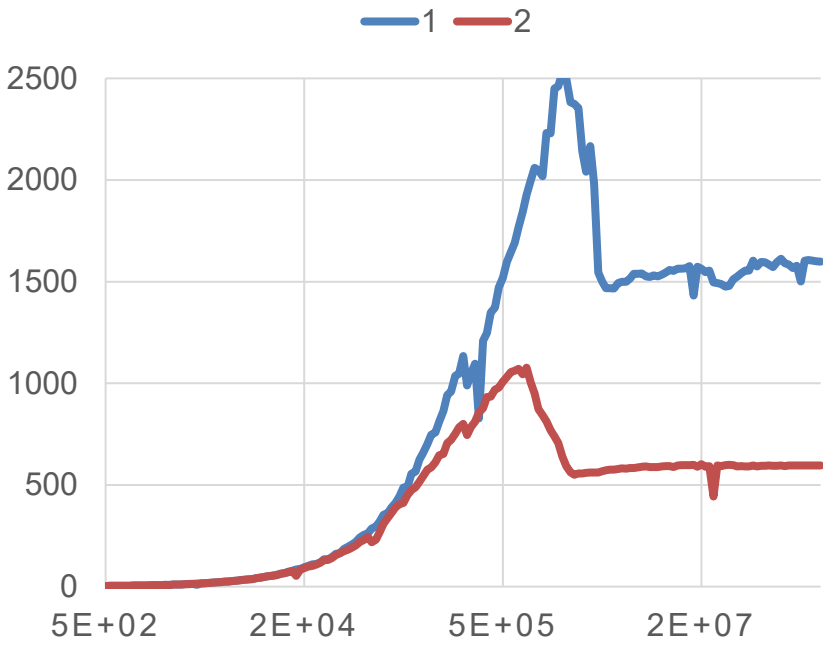
A100, strided stream, bandwidth in GB/s over number of array elements
CUDA (left) and OpenMP target (right)

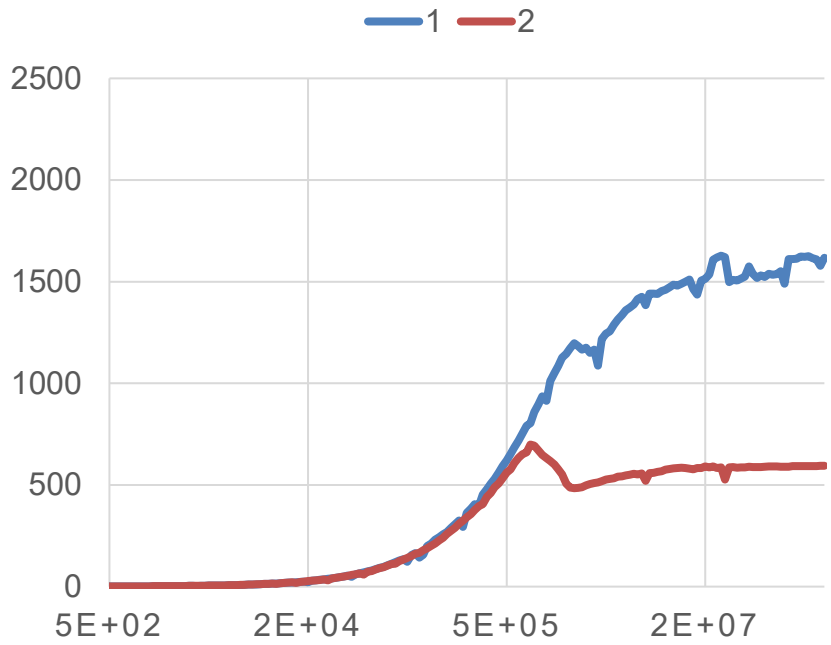




Strided Stream Benchmark

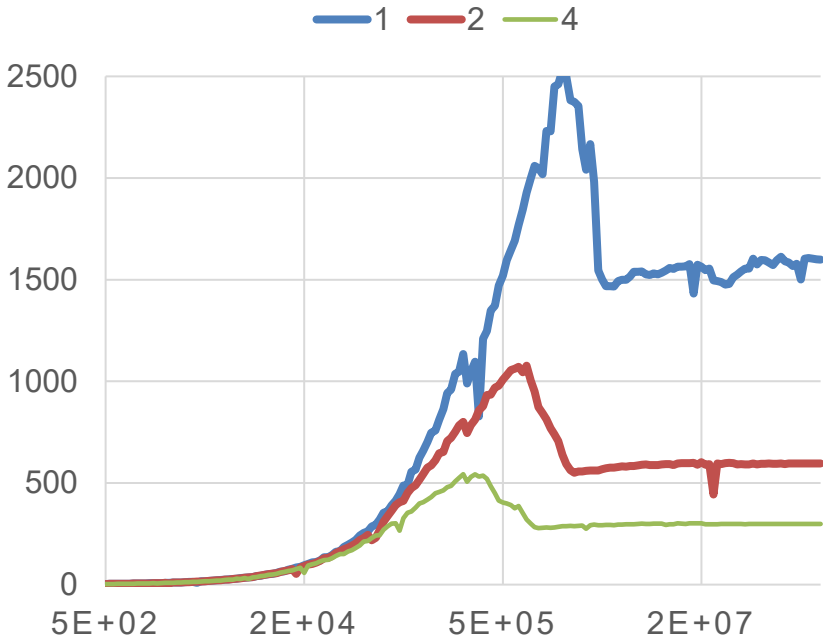
A100, strided stream, bandwidth in GB/s over number of array elements
CUDA (left) and OpenMP target (right)

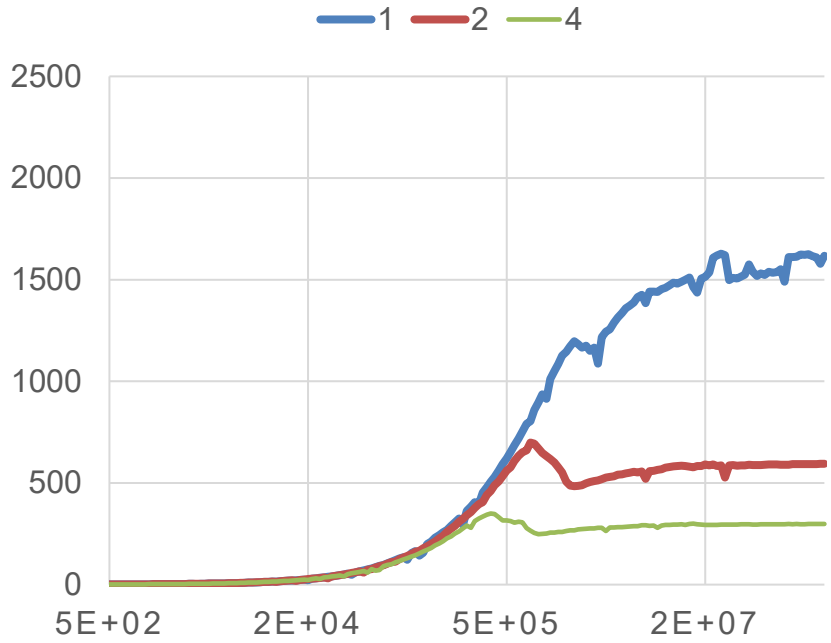




Strided Stream Benchmark

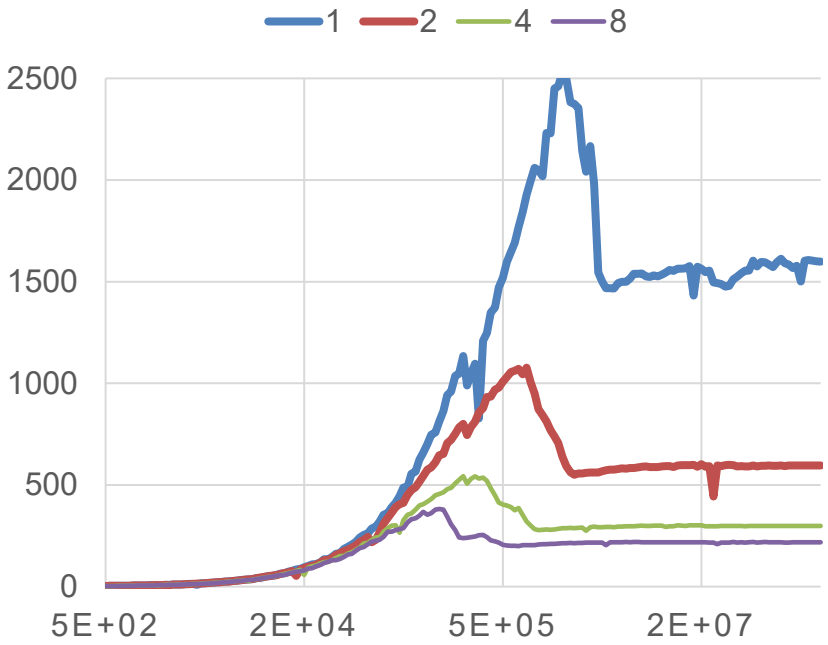
A100, strided stream, bandwidth in GB/s over number of array elements
CUDA (left) and OpenMP target (right)

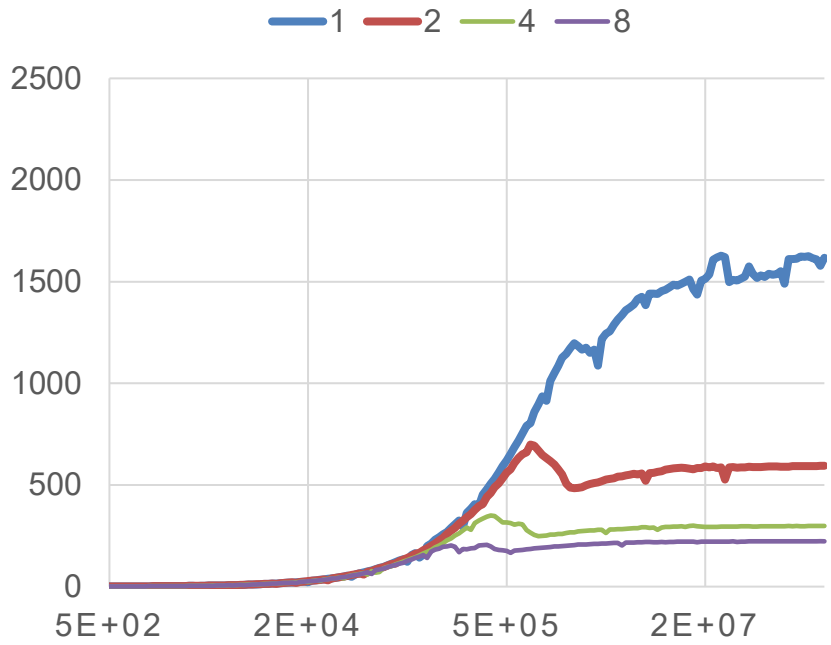




Strided Stream Benchmark

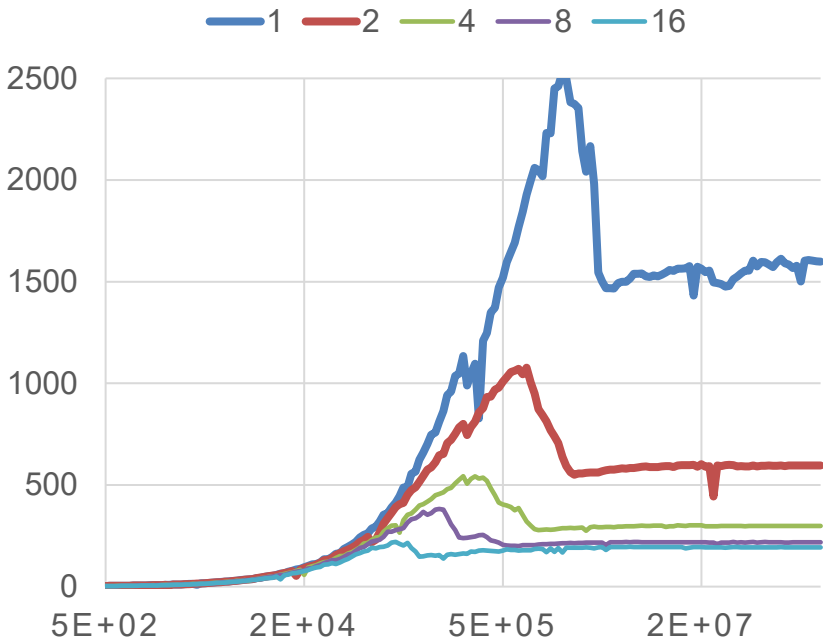
A100, strided stream, bandwidth in GB/s over number of array elements
CUDA (left) and OpenMP target (right)

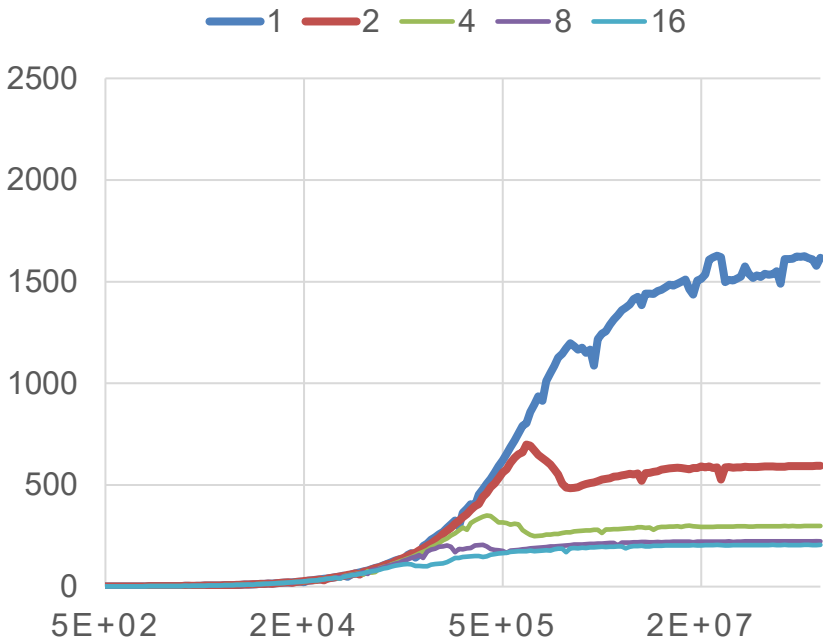




Strided Stream Benchmark

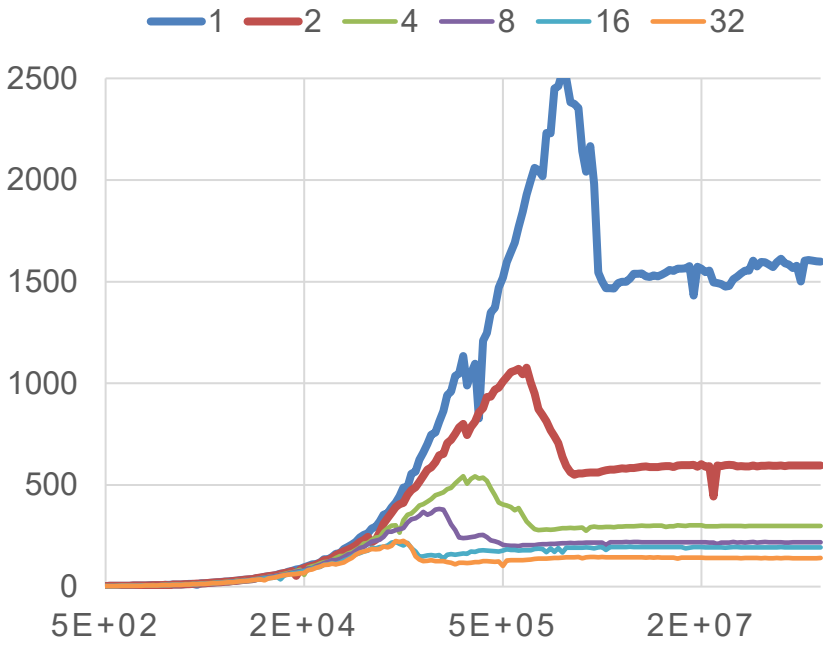
A100, strided stream, bandwidth in GB/s over number of array elements
CUDA (left) and OpenMP target (right)

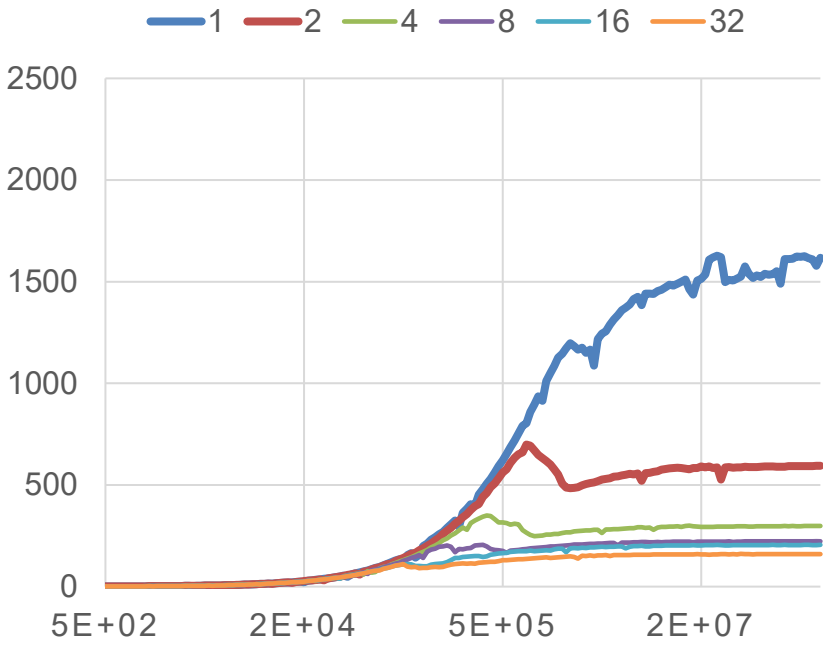




Strided Stream Benchmark

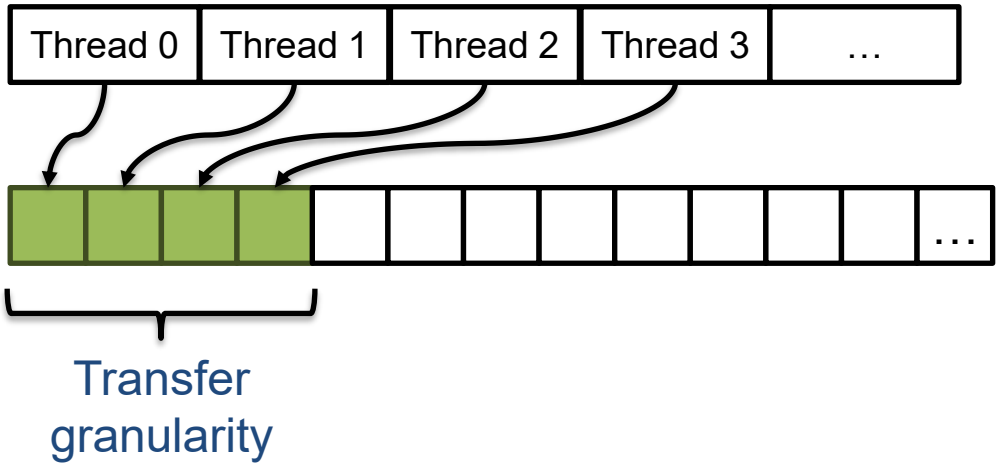
A100, strided stream, bandwidth in GB/s over number of array elements
CUDA (left) and OpenMP target (right)





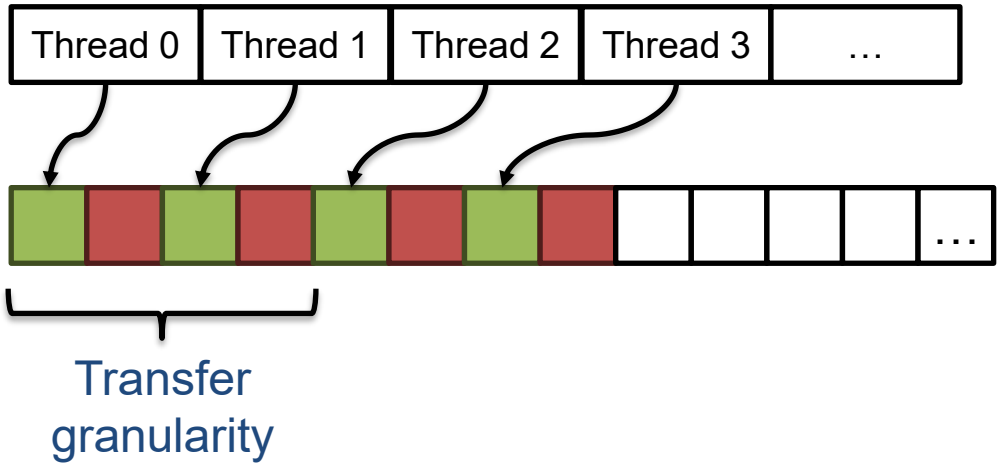
Memory Coalescing

- Coalesced access: consecutive threads access consecutive memory locations



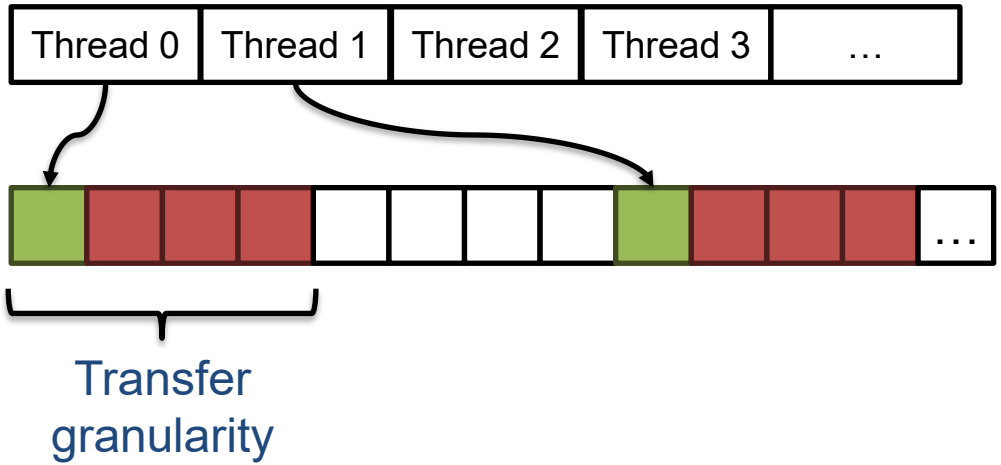
Memory Coalescing

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



Memory Coalescing

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



NHR FAU

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

Case Study

Dense Matrix Vector Multiplication

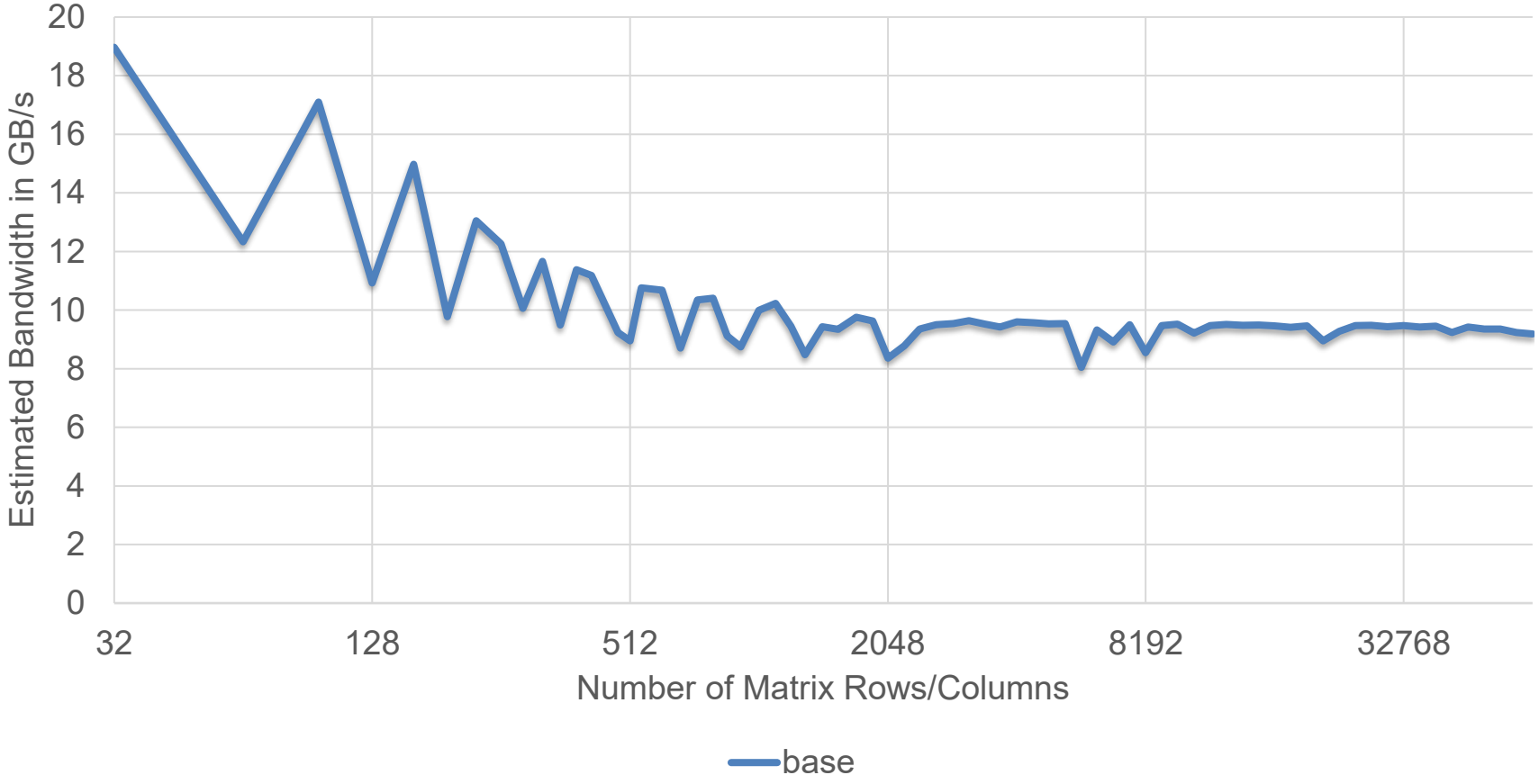
Case Study dMVM

- Code review
- All results are obtained on a single A100 80GB GPU in Alex
- Source code is available at https://github.com/SebastianKuckuk/apex/tree/main/src/teaching/dmvp

Case Study dMVM – Base Version – Main Kernel

```
for (int row = 0; row < nx; ++row) {
    dest[row] = 0.;
    for (int col = 0; col < nx; ++col)
        dest[row] += mat[row * nx + col] * src[col];
}
```

Case Study dMVM – Base Version

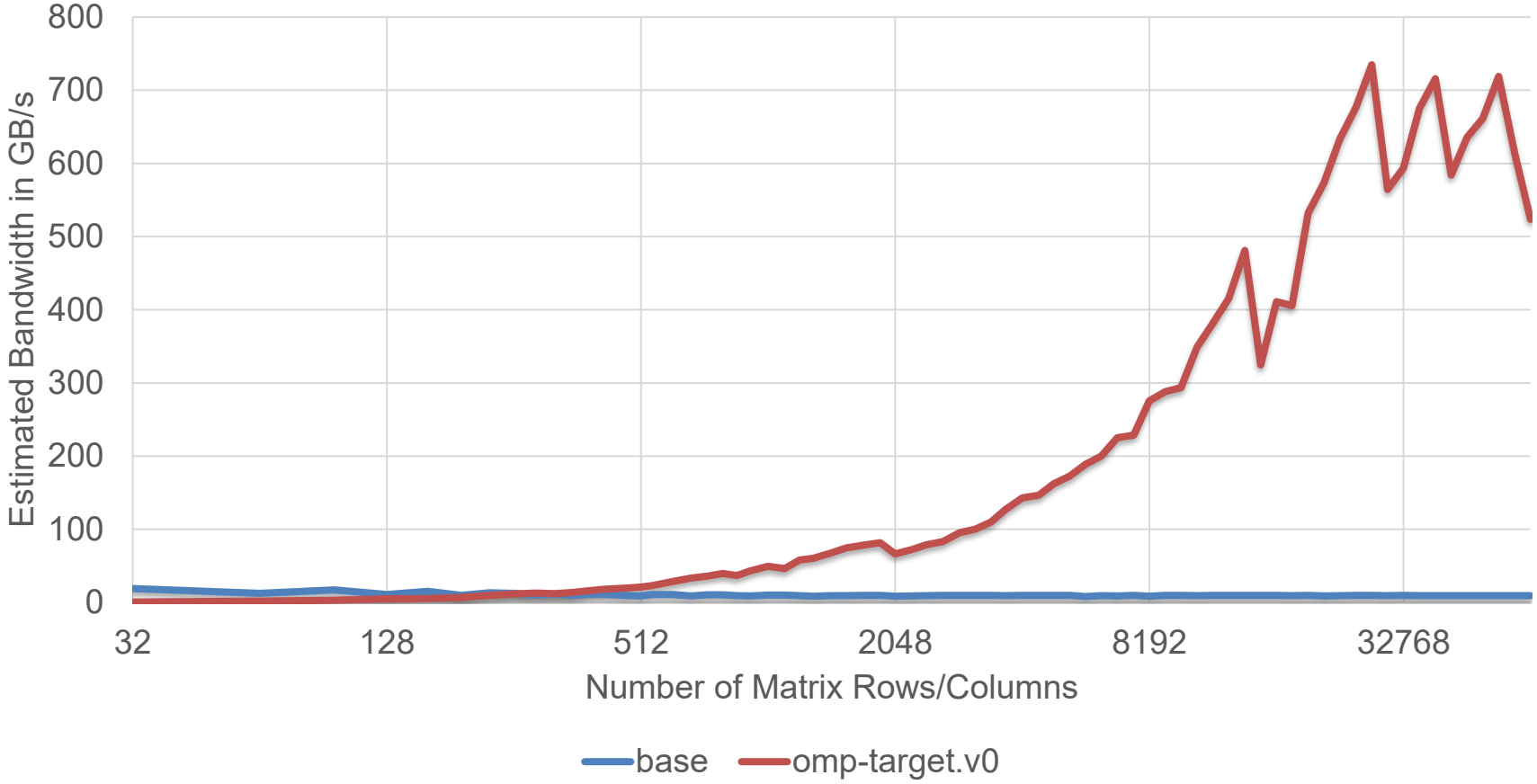


Case Study dMVM – OpenMP Target V0

- First Attempt: OpenMP target offload version

```
#pragma omp target teams distribute parallel for
for (int row = 0; row < nx; ++row) {
    dest[row] = 0.;
    for (int col = 0; col < nx; ++col)
        dest[row] += mat[row * nx + col] * src[col];
}
```

Case Study dMVM – OpenMP Target V0



Case Study dMVM – OpenMP Target V0

- OpenMP target offload version shows subpar resource utilization
- Transition to CUDA, but keep the same outer row parallelization for now

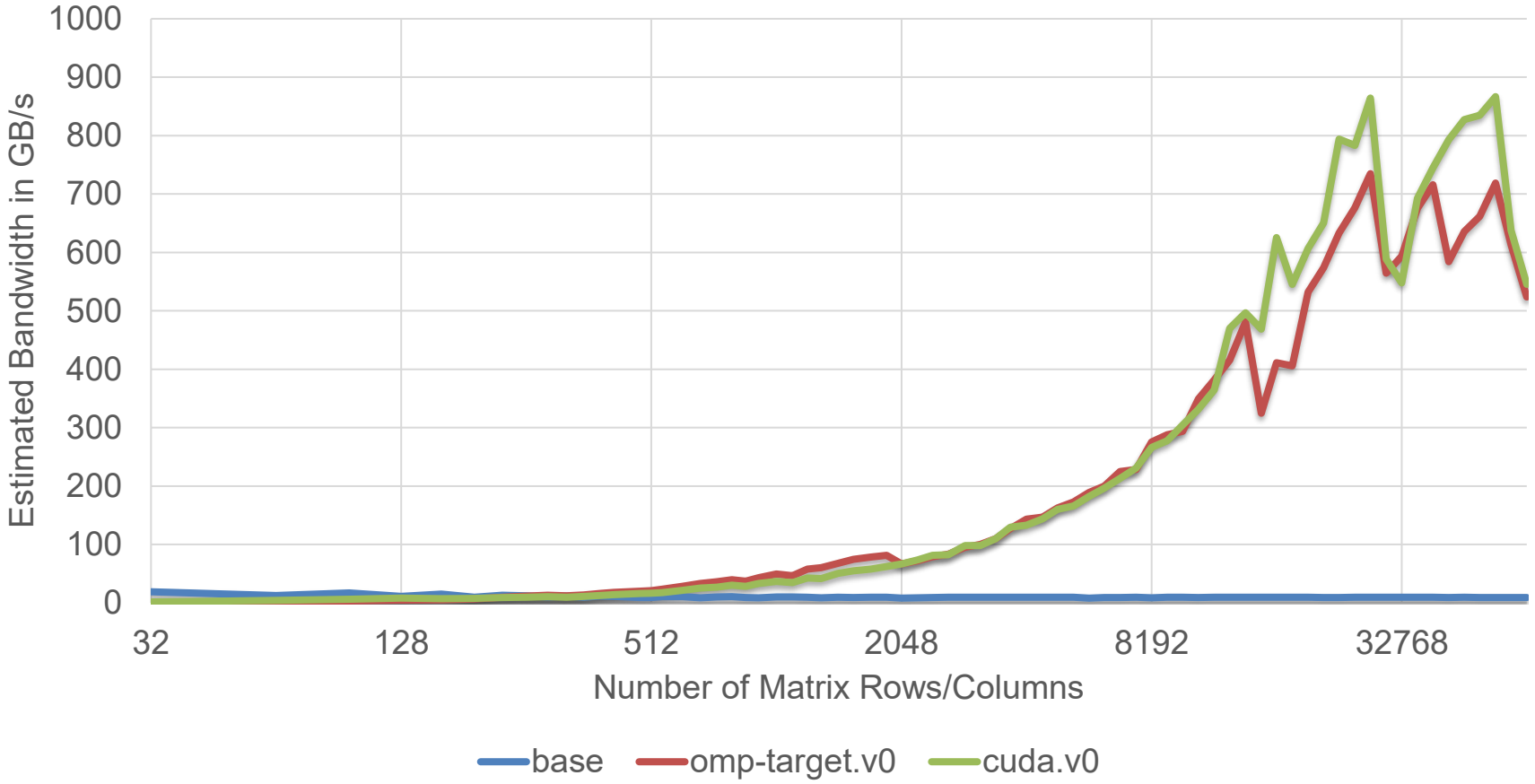
```
#pragma omp target teams distribute parallel for
for (int row = 0; row < nx; ++row) {
    dest[row] = 0.;
    for (int col = 0; col < nx; ++col)
        dest[row] += mat[row * nx + col] * src[col];
}
```

Case Study dMVM – CUDA V0

- Transition to CUDA, but keep the same outer row parallelization for now

```
int row = blockIdx.x * blockDim.x + threadIdx.x;

if (row < nx) {
    dest[row] = 0.;
    for (int col = 0; col < nx; ++col)
        dest[row] += mat[row * nx + col] * src[col];
}
```

Case Study dMVM – CUDA V0



Case Study dMVM – CUDA V0

- Frequent reads from/ writes to global memory (may be cached)

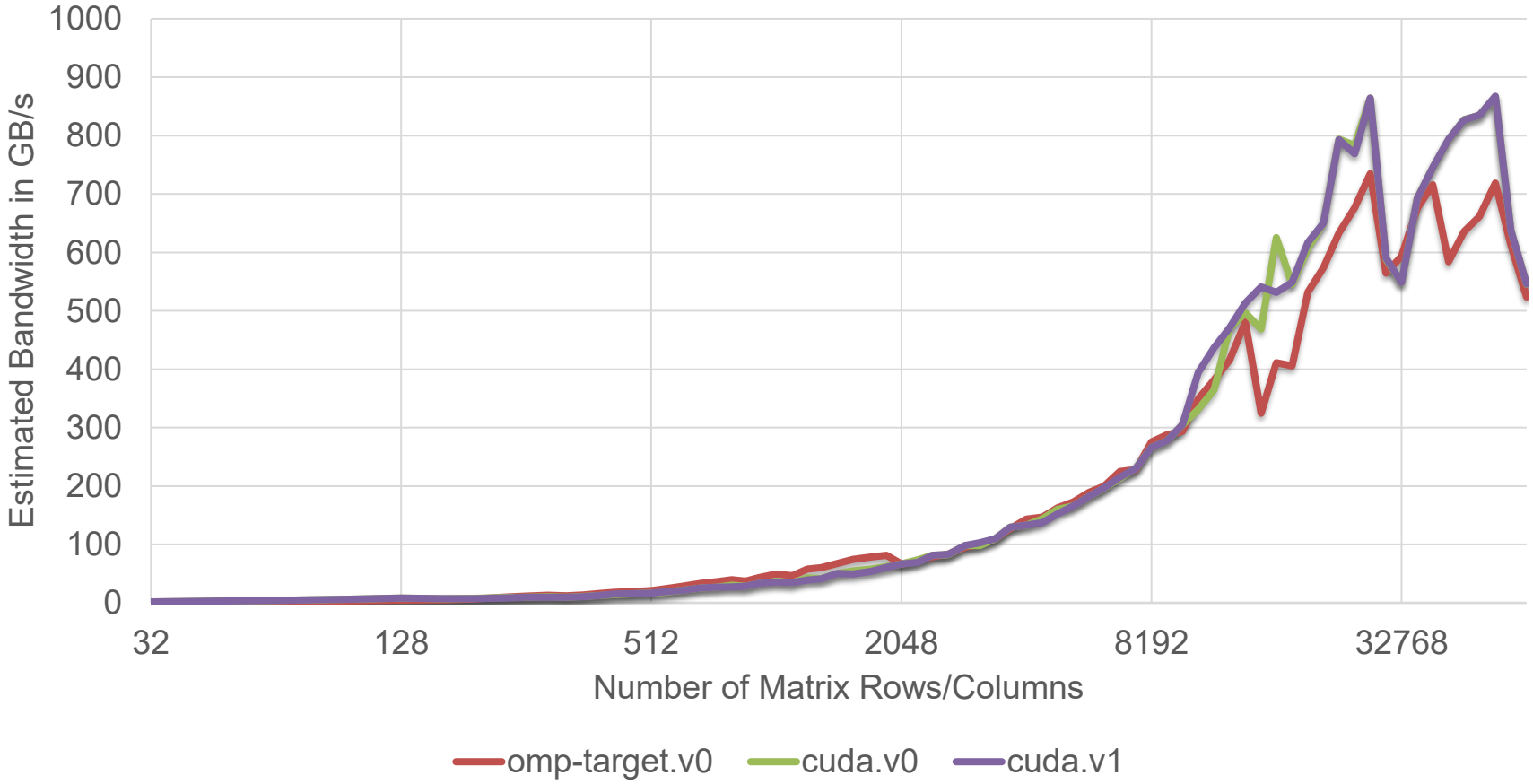
```
int row = blockIdx.x * blockDim.x + threadIdx.x;

if (row < nx) {
    dest[row] = 0.;
    for (int col = 0; col < nx; ++col)
        dest[row] += mat[row * nx + col] * src[col];
}
```

Case Study dMVM – CUDA V1

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

```
int row = blockIdx.x * blockDim.x + threadIdx.x;

if (row < nx) {
    tpe acc = (tpe)0;
    for (int col = 0; col < nx; ++col)
        acc += mat[row * nx + col] * src[col];
    dest[row] = acc;
}
```

Case Study dMVM – CUDA V1



Case Study dMVM – CUDA V1

- Outer loop parallelization leads to a small number of threads
- Switch execution configuration to 2D

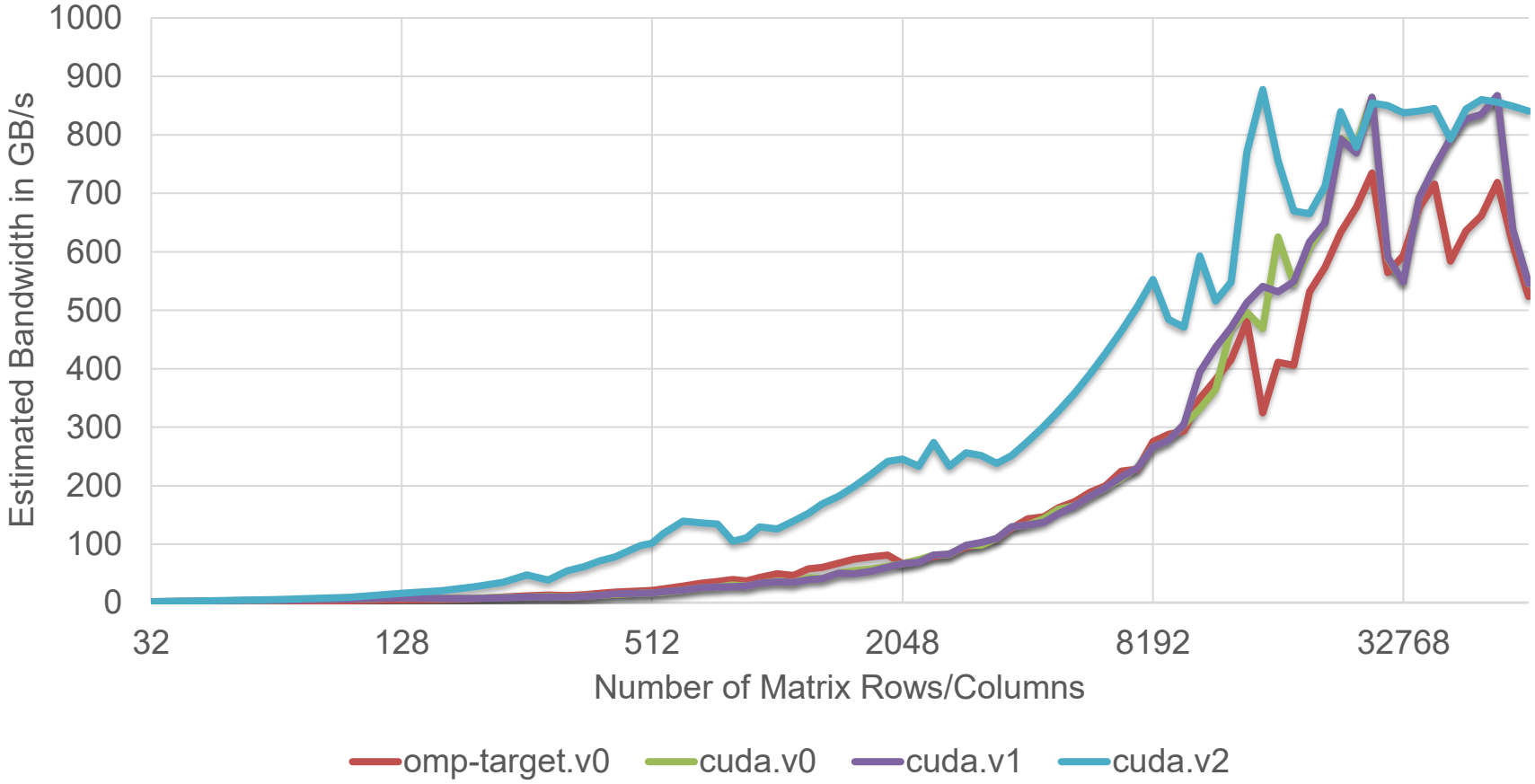
```
int row = blockIdx.x * blockDim.x + threadIdx.x;

if (row < nx) {
    tpe acc = (tpe)0;
    for (int col = 0; col < nx; ++col)
        acc += mat[row * nx + col] * src[col];
    dest[row] = acc;
}
```

Case Study dMVM – CUDA V2

- Switch execution configuration to 2D
- Multiple threads now write (concurrently) to the same output
- Use atomics to avoid data races

```
int row = blockIdx.x * blockDim.x + threadIdx.x;
int col = blockIdx.y * blockDim.y + threadIdx.y;

if (row < nx && col < nx)
    atomicAdd(&dest[row], mat[row * nx + col] * src[col]);
```

Case Study dMVM – CUDA V2



Case Study dMVM – CUDA V2

- Atomic congestion and high memory traffic
- Compute partial results and add grid-stride loops

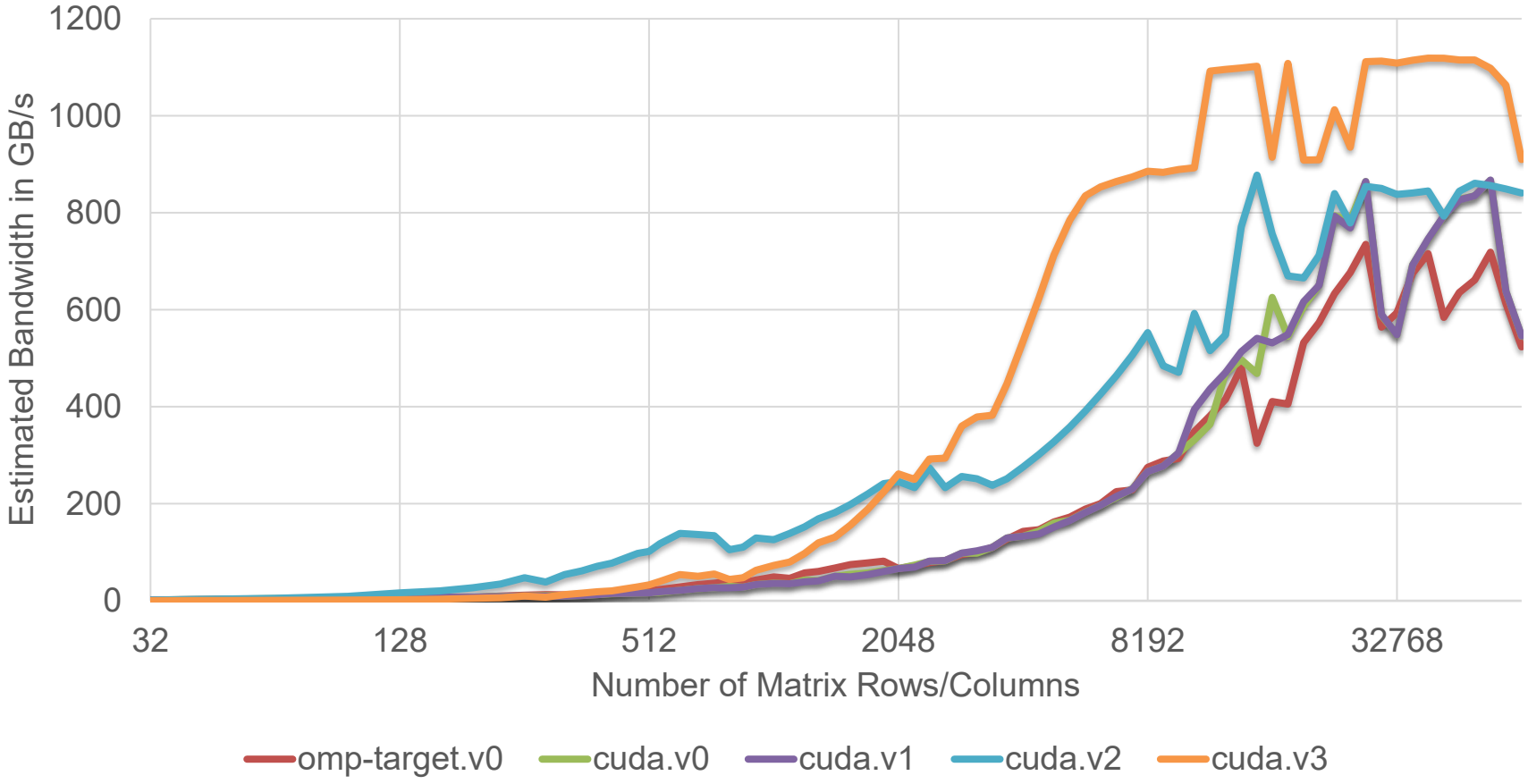
```
int row = blockIdx.x * blockDim.x + threadIdx.x;
int col = blockIdx.y * blockDim.y + threadIdx.y;

if (row < nx && col < nx)
    atomicAdd(&dest[row], mat[row * nx + col] * src[col]);
```

Case Study dMVM – CUDA V3

- Compute partial results and add grid-stride loops

```
int rowStart  = blockIdx.x * blockDim.x + threadIdx.x;
int rowStride = gridDim.x  * blockDim.x;
int colStart  = blockIdx.y * blockDim.y + threadIdx.y;
int colStride = gridDim.y  * blockDim.y;

for (int row = rowStart; row < nx; row += rowStride) {
    tpe acc = (tpe)0;
    for (int col = colStart; col < nx; col += colStride)
        acc += mat[row * nx + col] * src[col];
    atomicAdd(&dest[row], acc);
}
```

Case Study dMVM – CUDA V3



Case Study dMVM – CUDA V3

- Coalesced writes but uncoalesced reads

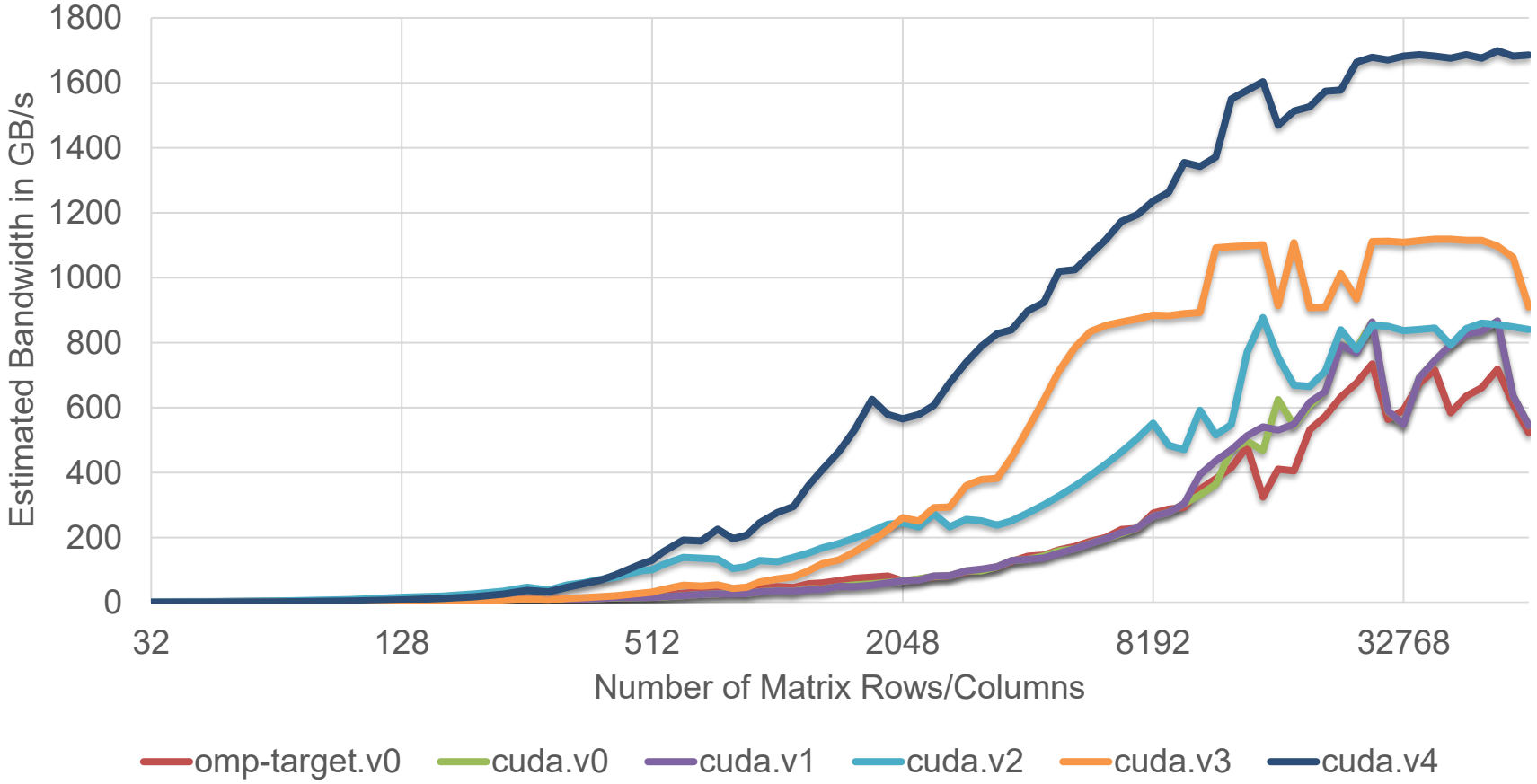
```
int rowStart  = blockIdx.x * blockDim.x + threadIdx.x;
int rowStride = gridDim.x  * blockDim.x;
int colStart  = blockIdx.y * blockDim.y + threadIdx.y;
int colStride = gridDim.y  * blockDim.y;

for (int row = rowStart; row < nx; row += rowStride) {
    tpe acc = (tpe)0;
    for (int col = colStart; col < nx; col += colStride)
        acc += mat[row * nx + col] * src[col];
    atomicAdd(&dest[row], acc);
}
```

Case Study dMVM – CUDA V4

- Transpose thread mapping (and adapt execution configuration)

```
int colStart  = blockIdx.x * blockDim.x + threadIdx.x;
int colStride = gridDim.x  * blockDim.x;
int rowStart  = blockIdx.y * blockDim.y + threadIdx.y;
int rowStride = gridDim.y  * blockDim.y;

for (int row = rowStart; row < nx; row += rowStride) {
    tpe acc = (tpe)0;
    for (int col = colStart; col < nx; col += colStride)
        acc += mat[row * nx + col] * src[col];
    atomicAdd(&dest[row], acc);
}
```

Case Study dMVM – CUDA V4



Case Study dMVM – CUDA V4

- Multiple threads of the same block perform atomic updates into the same location

```
for (int row = rowStart; row < nx; row += rowStride) {
    tpe acc = (tpe)0;
    for (int col = colStart; col < nx; col += colStride)
        acc += mat[row * nx + col] * src[col];
    atomicAdd(&dest[row], acc);
}
```

Case Study dMVM – CUDA V5

- Use shared memory to buffer and aggregate output data
- Assume `blockDim.x == blockDim.y == 32`
- Start by allocating shared memory

```
__shared__ double out[32];
```

Case Study dMVM – CUDA V5

- Zero the shared memory in each iteration of the row loop

```
__shared__ double out[32];

for (int row = rowStart; row < nx; row += rowStride) {
    if (threadIdx.x < 32 && 0 == threadIdx.y)
        out[threadIdx.x] = 0;
```

Case Study dMVM – CUDA V5

- Wait for the completion of this operation before continuing – other threads might accumulate into an unprepped shared memory location otherwise

```
__shared__ double out[32];

for (int row = rowStart; row < nx; row += rowStride) {
    if (threadIdx.x < 32 && 0 == threadIdx.y)
        out[threadIdx.x] = 0;

    __syncthreads();
```

Case Study dMVM – CUDA V5

- Accumulate into shared memory

```
__shared__ double out[32];

for (int row = rowStart; row < nx; row += rowStride) {
    // ...
    __syncthreads();

    tpe acc = (tpe)0;
    for (int col = colStart; col < nx; col += colStride)
        acc += mat[row * nx + col] * src[col];
    atomicAdd(&out[threadIdx.y], acc);
```

Case Study dMVM – CUDA V5

- Synchronize again to make sure that all contributions have been added to shared memory

```
__shared__ double out[32];

for (int row = rowStart; row < nx; row += rowStride) {
    // init buffer
    __syncthreads();

    // fill buffer
    __syncthreads();
```

Case Study dMVM – CUDA V5

- Write back the accumulated result coalesced with a reduced number of threads

```
__shared__ double out[32];

for (int row = rowStart; row < nx; row += rowStride) {
    // init buffer, sync
    // fill buffer, sync

    if (threadIdx.x < 32 && 0 == threadIdx.y)
        atomicAdd(&dest[row + threadIdx.x],
                  out[threadIdx.x]);
}
```

Case Study dMVM – CUDA V5

- Bonus question: is there another thread sync required at the loop's end?

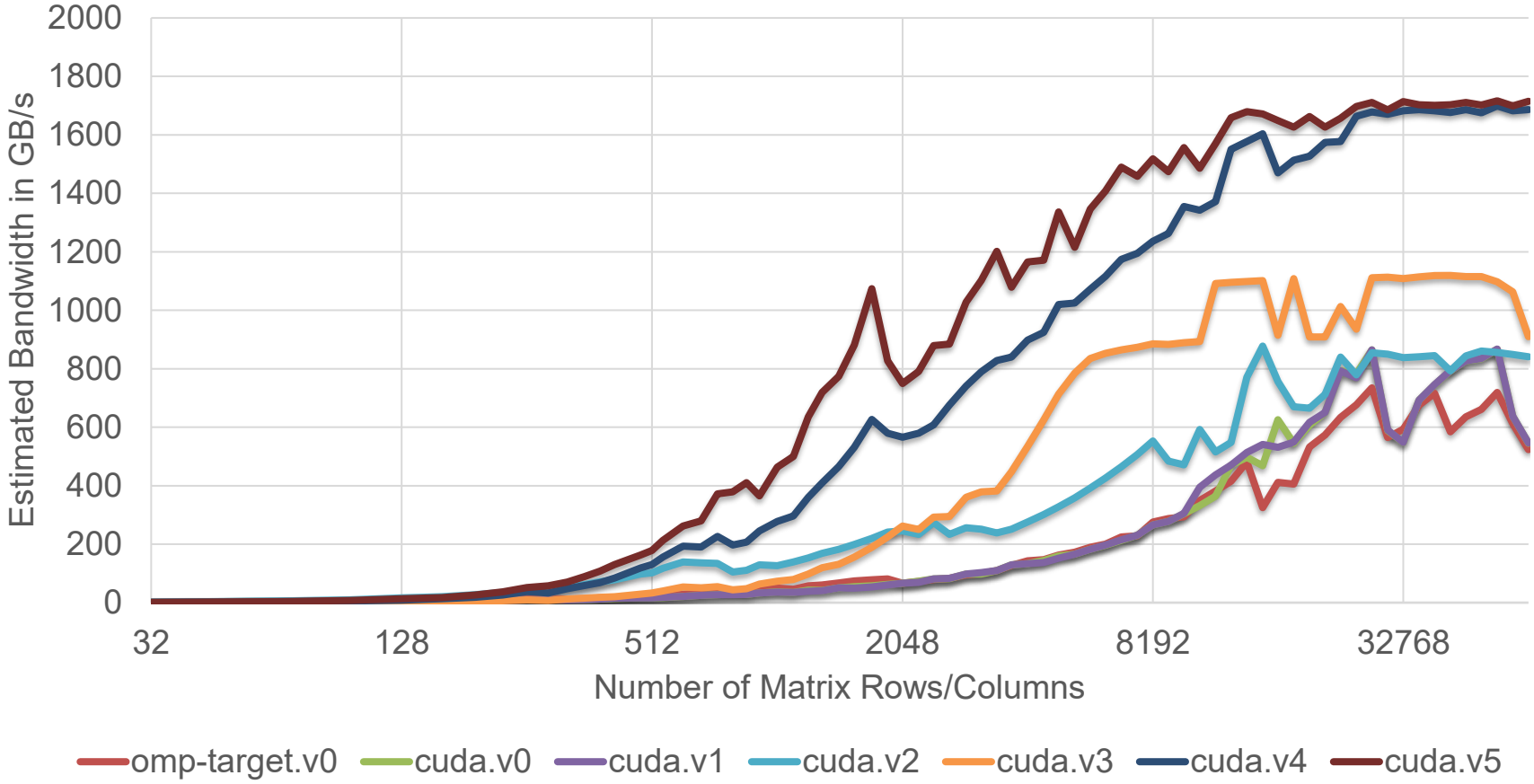
```
__shared__ double out[32];

for (int row = rowStart; row < nx; row += rowStride) {
    // init buffer, sync
    // fill buffer, sync

    if (threadIdx.x < 32 && 0 == threadIdx.y)
        atomicAdd(&dest[row + threadIdx.x],
                  out[threadIdx.x]);
}
```

Case Study dMVM – CUDA V5

- Bonus question: is there another thread sync required at the loop's end?
- Answer: no, the threads that init and write to global memory are the same.

```
__shared__ double out[32];

for (int row = rowStart; row < nx; row += rowStride) {
    // init buffer, sync
    // fill buffer, sync

    if (threadIdx.x < 32 && 0 == threadIdx.y)
        atomicAdd(&dest[row + threadIdx.x],
                  out[threadIdx.x]);
}
```

Case Study dMVM – CUDA V5



Case Study dMVM – CUDA V5

- Multiple threads of a block read the same values from `src`
- This should be cached in L1, but using shared memory is also possible

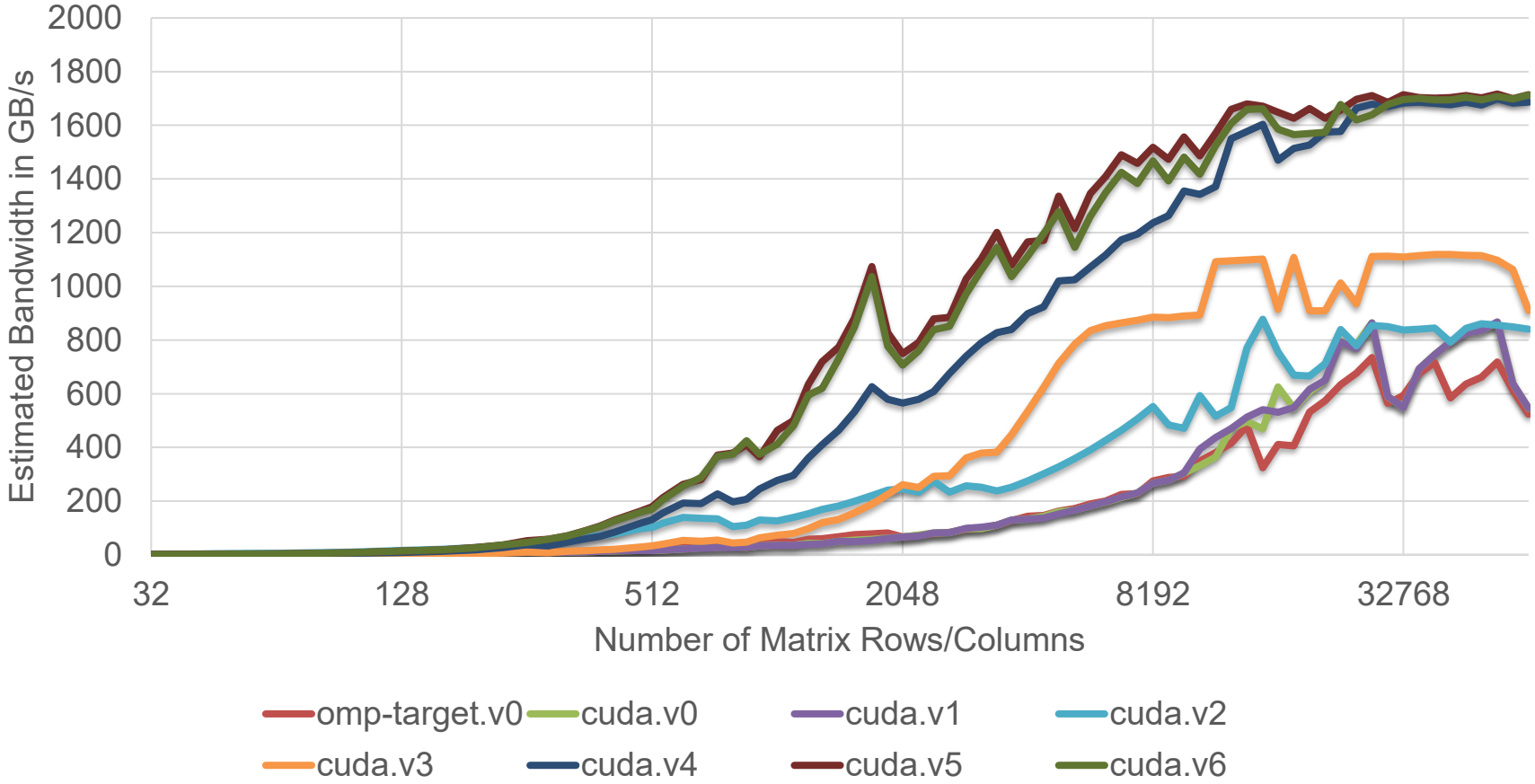
```
// ...

tpe acc = (tpe)0;
for (int col = colStart; col < nx; col += colStride)
    acc += mat[row * nx + col] * src[col];
atomicAdd(&out[threadIdx.y], acc);

// ...
```

Case Study dMVM – CUDA V6

- Use shared memory to buffer *input*

```
tpe acc = (tpe)0;
for (int col = colStart; col < nx; col += colStride) {
    if (threadIdx.x < 32 && 0 == threadIdx.y)
        in[threadIdx.x] = src[col];
    __syncthreads();

    acc += mat[row * nx + col] * in[threadIdx.x];
}
atomicAdd(&out[threadIdx.y], acc);
```

Case Study dMVM – CUDA V6

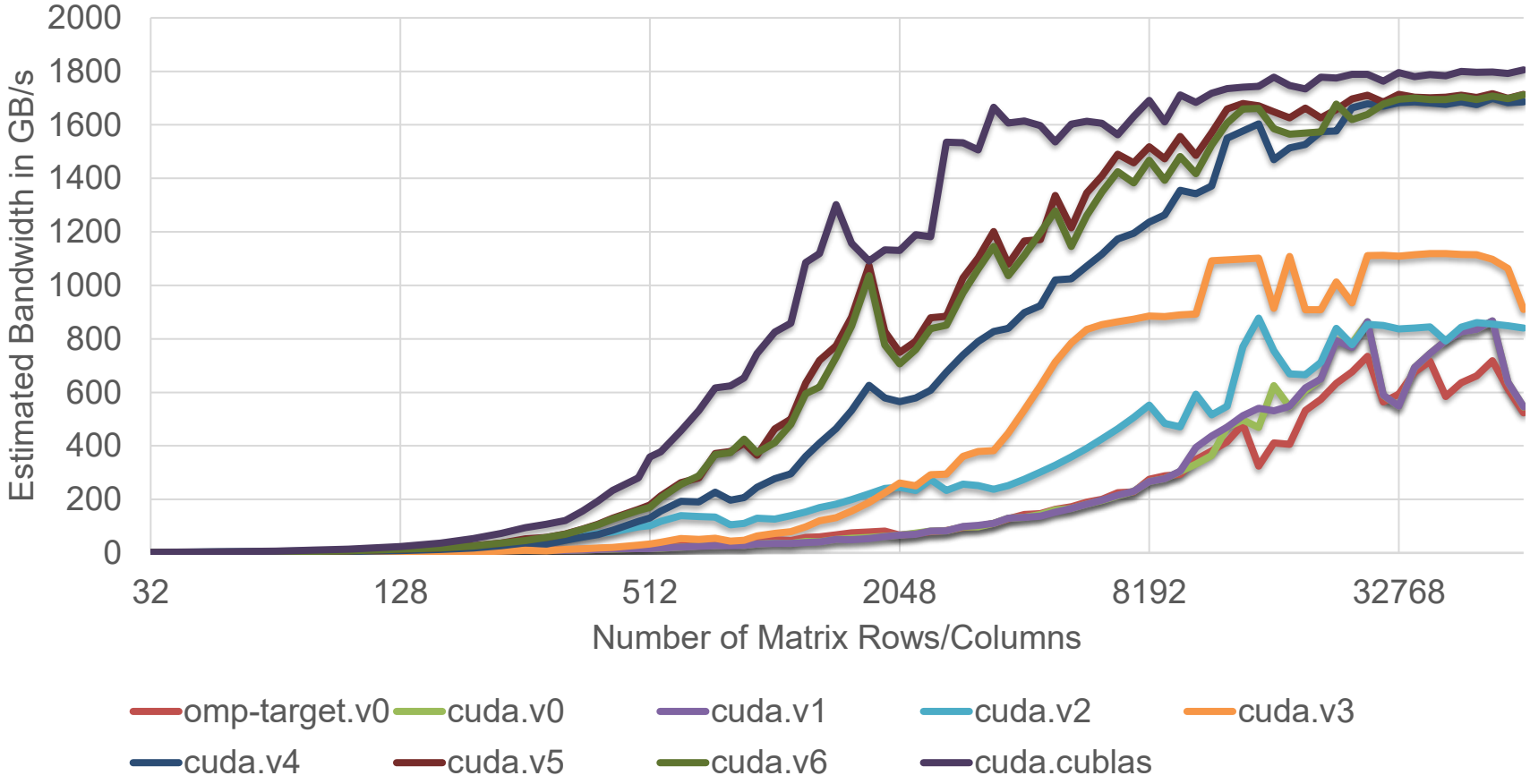


Case Study dMVM – Library

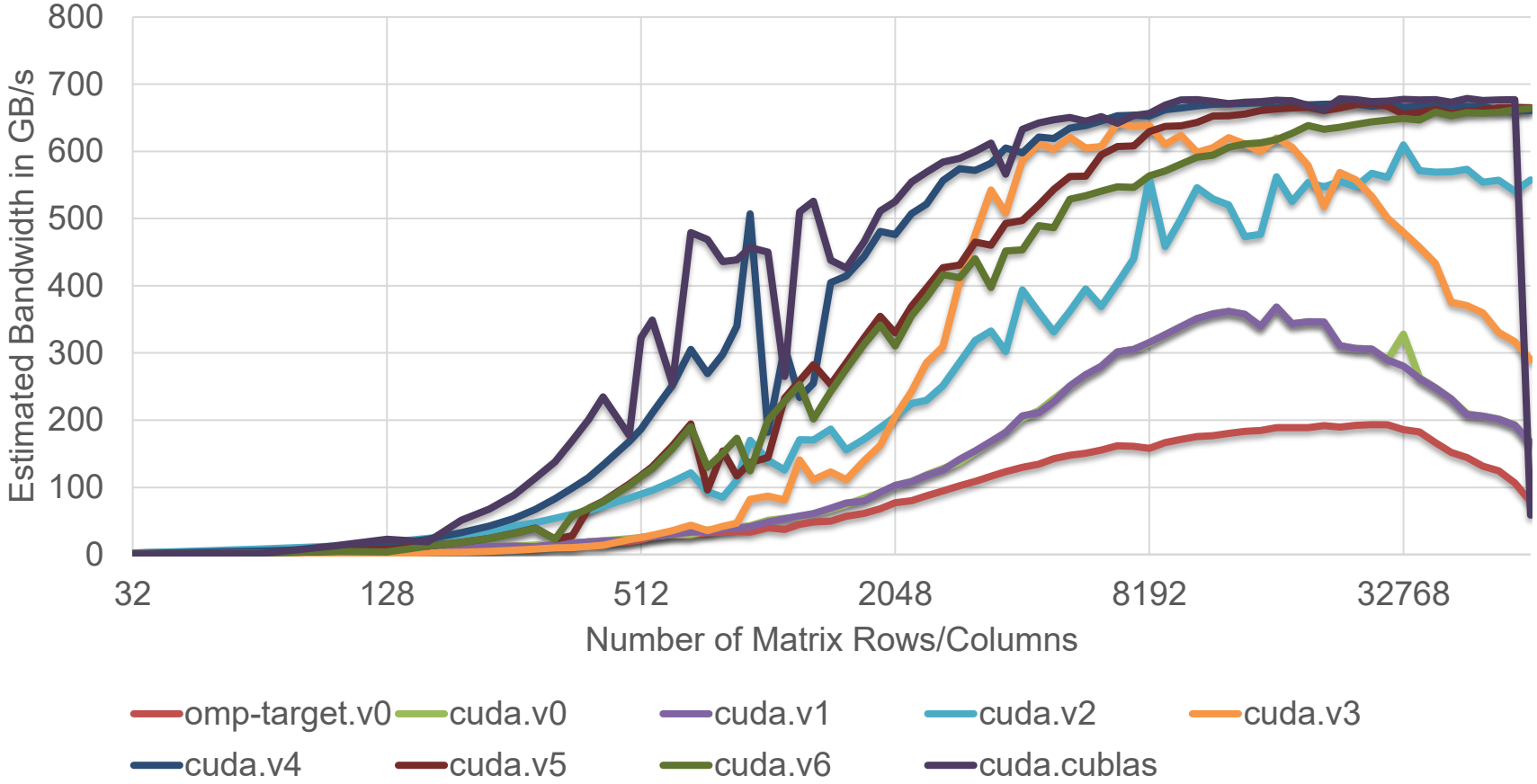
- Next steps: parameter tuning, warp primitives, …
- OR: rely on the hard work of others (in this case cuBLAS)
- https://docs.nvidia.com/cuda/cublas/#cublas-t-gemv

```
cublasHandle_t handle;
cublasCreate_v2(&handle);

const auto alpha = 1.;
const auto beta = 0.;
cublasDgemv(handle, CUBLAS_OP_N, nx, nx,
            &alpha, d_mat, nx, d_src, 1, &beta,
            d_dest, 1);
```

Case Study dMVM – cuBLAS



Case Study dMVM – **A40** Comparison



Case Study dMVM – OpenMP V0 Revisited

- With what we learned – can we make the OpenMP version better?
- Main issue: lack of threads/parallel work due to outer loop parallelization

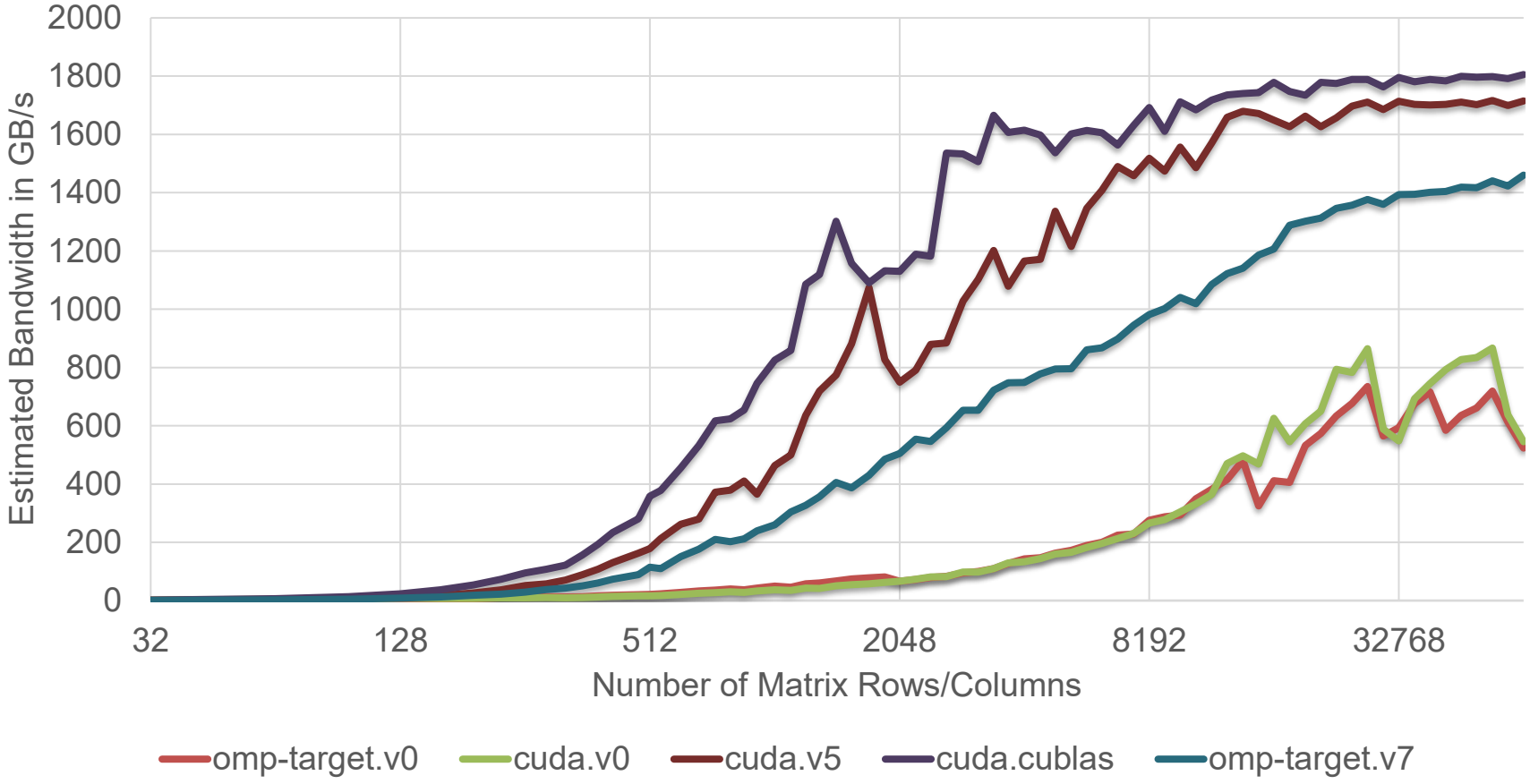
```
#pragma omp target teams distribute parallel for
for (int row = 0; row < nx; ++row) {
    dest[row] = 0.;
    for (int col = 0; col < nx; ++col)
        dest[row] += mat[row * nx + col] * src[col];
}
```

Case Study dMVM – OpenMP V7

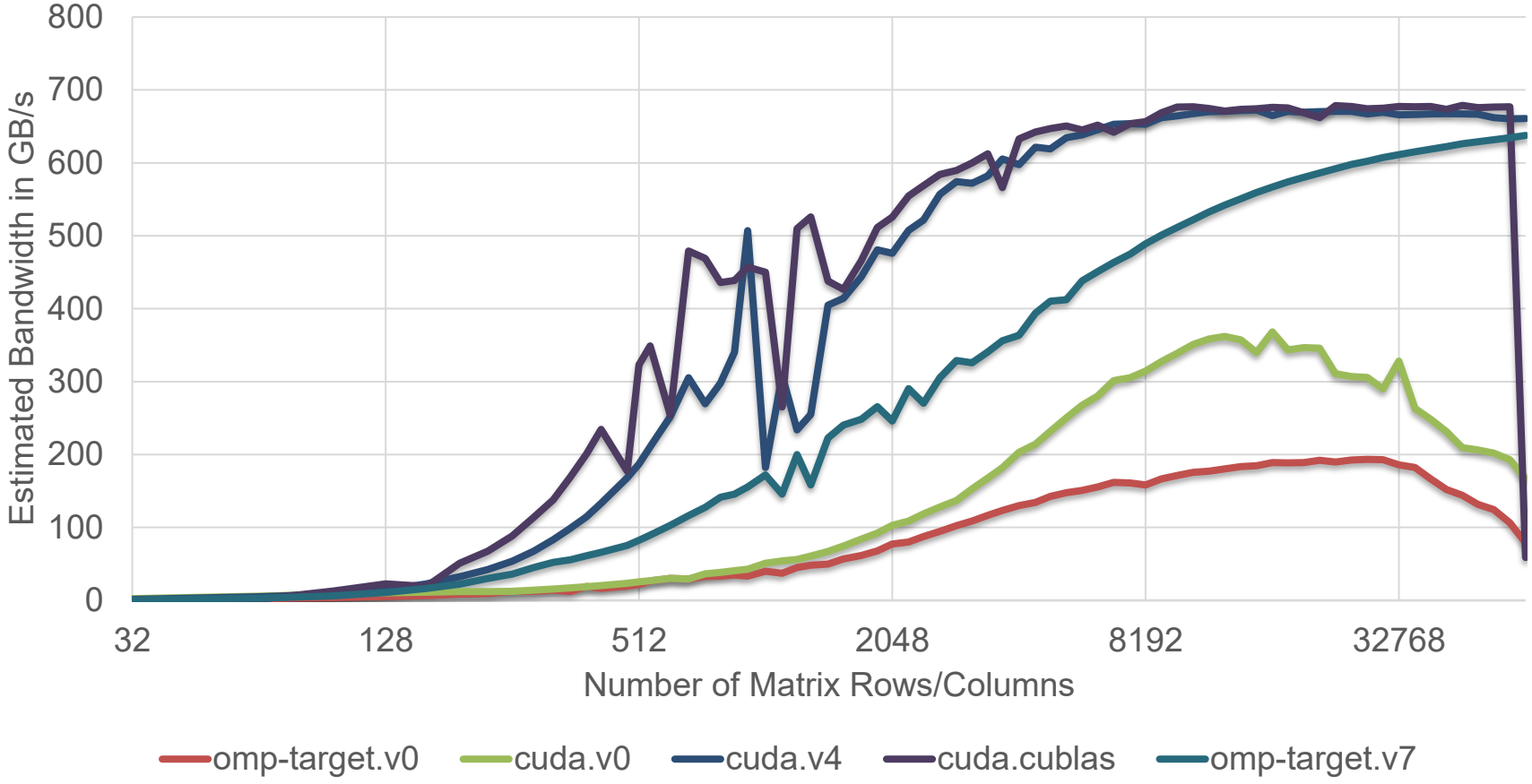
- Hierarchical parallelization + reduction
- No atomics required since each block will write to exactly one output element

```
#pragma omp target teams distribute
for (int row = 0; row < nx; ++row) {
    tpe acc = (tpe)0;
    #pragma omp parallel for reduction(+ : acc)
    for (int col = 0; col < nx; ++col)
        acc += mat[row * nx + col] * src[col];
    dest[row] = acc;
}
```

Case Study dMVM – OpenMP V7



Case Study dMVM – **A40** Comparison



Case Study dMVM – OpenMP V7

- No sharing of elements from `src` within on block (team) anymore

```
#pragma omp target teams distribute
for (int row = 0; row < nx; ++row) {
    tpe acc = (tpe)0;
    #pragma omp parallel for reduction(+ : acc)
    for (int col = 0; col < nx; ++col)
        acc += mat[row * nx + col] * src[col];
    dest[row] = acc;
}
```

Case Study dMVM – OpenMP V8

- Partial unrolling of the outer loop

```
#pragma omp target teams distribute
for (int row = 0; row < nx; row += 4) {
```

Case Study dMVM – OpenMP V8

- Multiple accumulators

```
#pragma omp target teams distribute
for (int row = 0; row < nx; row += 4) {
    tpe acc0 = (tpe)0;
    tpe acc1 = (tpe)0;
    tpe acc2 = (tpe)0;
    tpe acc3 = (tpe)0;
```

Case Study dMVM – OpenMP V8

- Multiple reduction targets

```
#pragma omp target teams distribute
for (int row = 0; row < nx; row += 4) {
    // acc0 ... acc3
    #pragma omp parallel for \
            reduction(+ : acc0, acc1, acc2, acc3)
    for (int col = 0; col < nx; ++col) {
```

Case Study dMVM – OpenMP V8

- Multiple accumulations

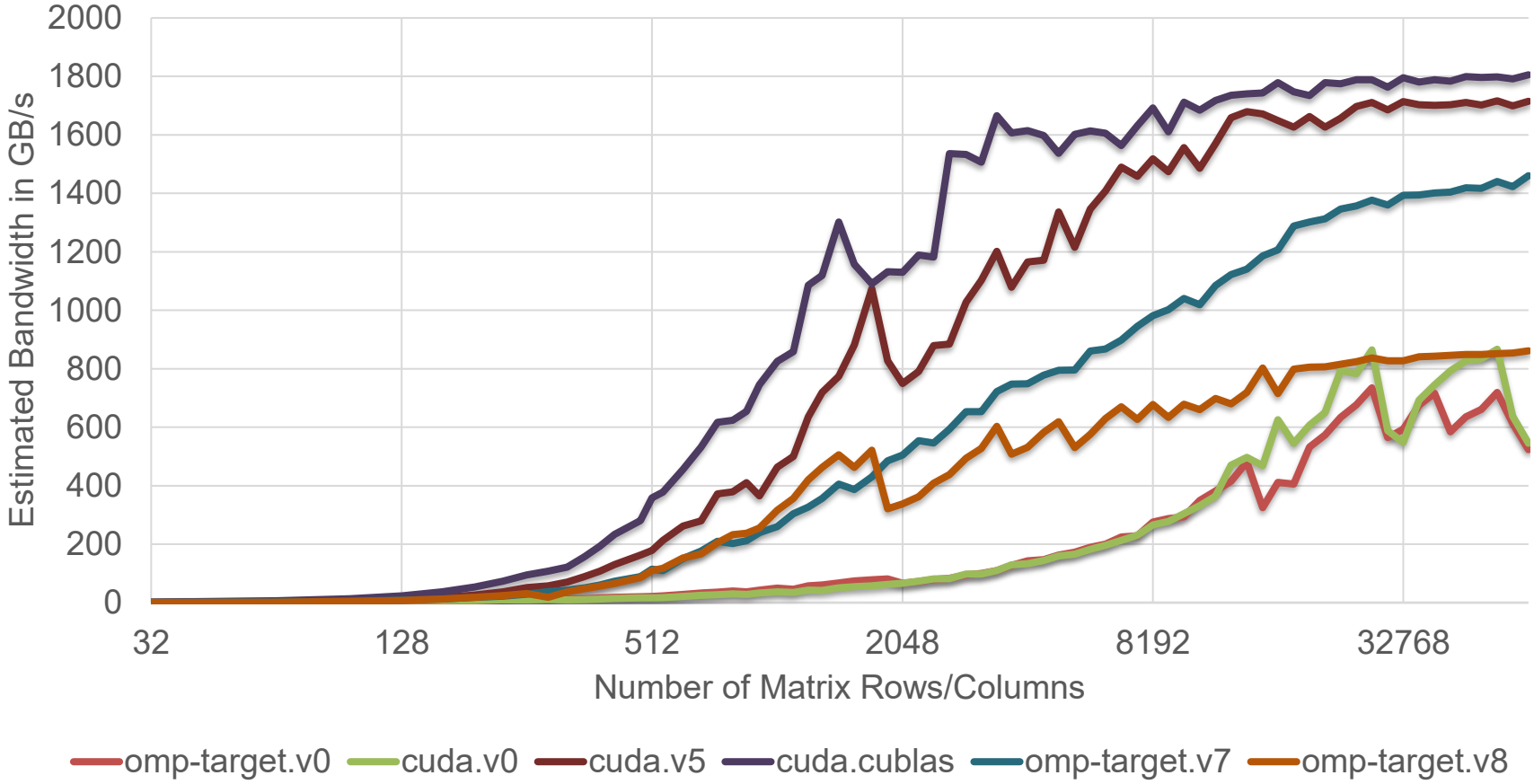
```
for (int col = 0; col < nx; ++col) {
    acc0 += mat[row * nx + col] * src[col];
    if (row + 1 < nx)
        acc1 += mat[(row + 1) * nx + col] * src[col];
    if (row + 2 < nx)
        acc2 += mat[(row + 2) * nx + col] * src[col];
    if (row + 3 < nx)
        acc3 += mat[(row + 3) * nx + col] * src[col];
```

Case Study dMVM – OpenMP V8

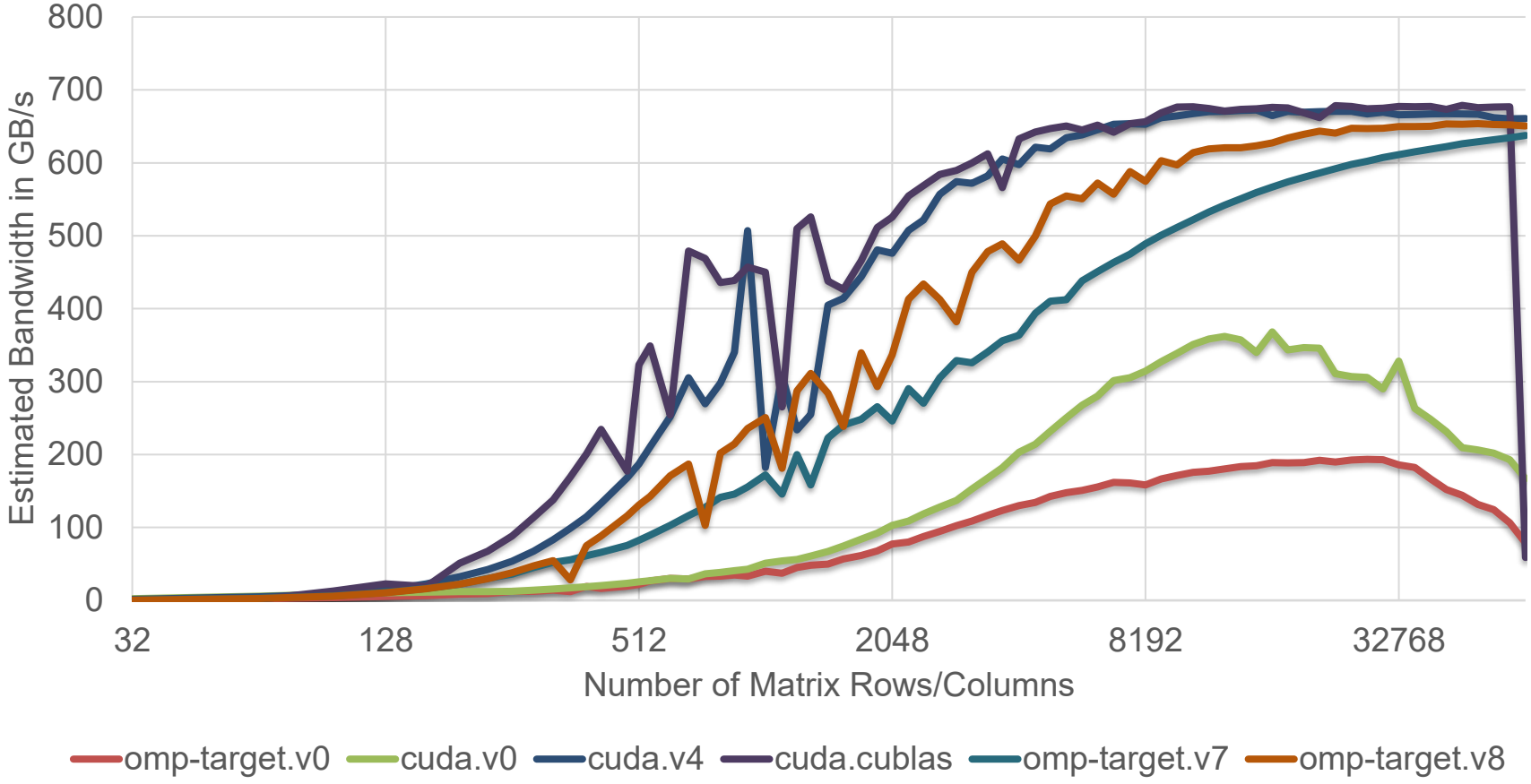
- Write back multiple values (one thread only)

```
// row loop
    // init acc0 ... acc3
    // col loop
        // fill acc0 ... acc3

    dest[row] = acc0;
    if (row + 1 < nx) dest[row + 1] = acc1;
    if (row + 2 < nx) dest[row + 2] = acc2;
    if (row + 3 < nx) dest[row + 3] = acc3;
```

Case Study dMVM – OpenMP V8



Case Study dMVM – **A40** Comparison



Case Study dMVM – OpenMP V9

- Employ loops + templating (`blk` as template parameter)

```
#pragma omp target teams distribute
for (int row = 0; row < nx; row += blk) {
    tpe acc[blk] = {};
```

Case Study dMVM – OpenMP V9

- Do the local multi-value accumulation in a fixed-length loop
- Use OpenMP array reductions

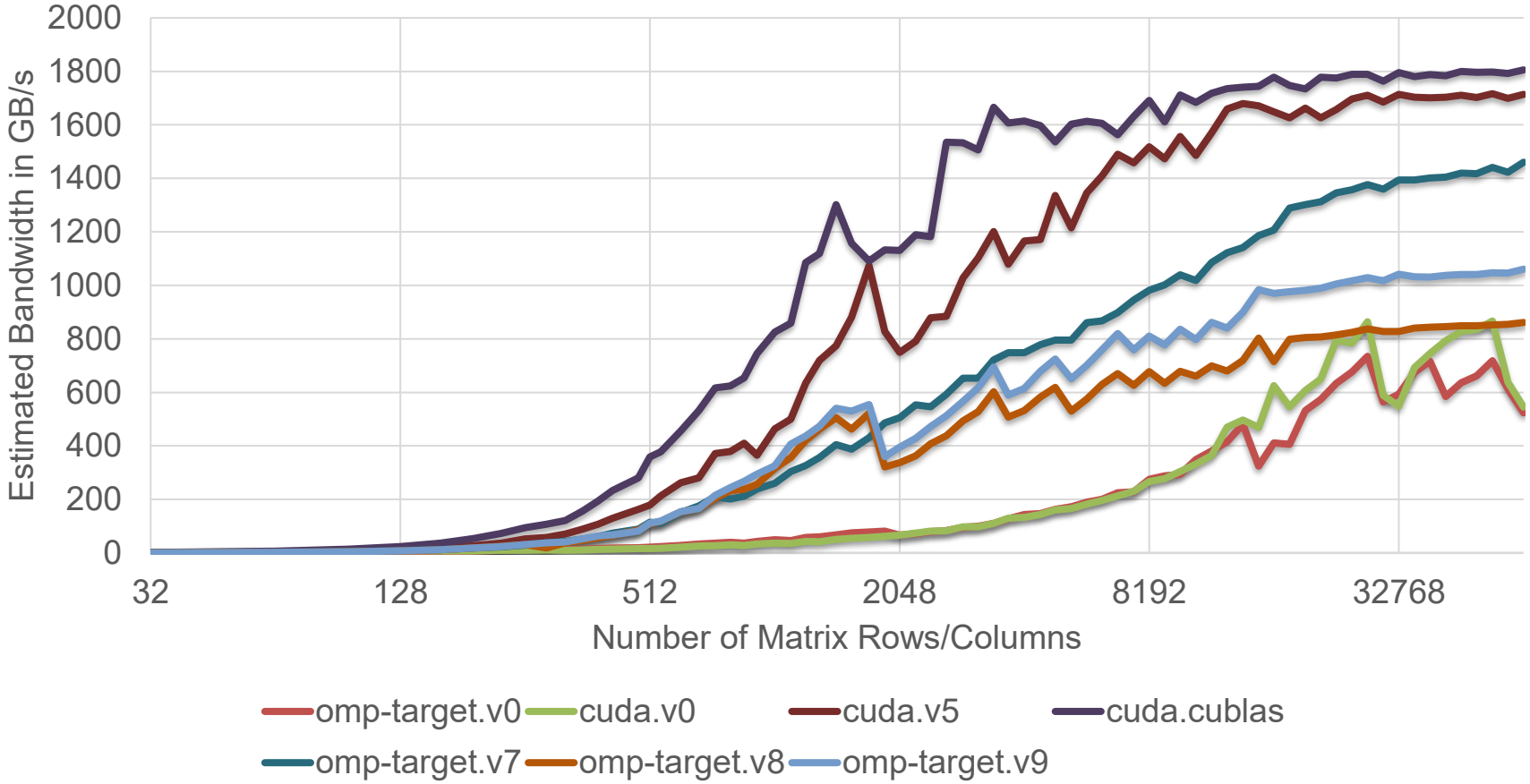
```
tpe acc[blk] = {};

#pragma omp parallel for reduction(+ : acc[:blk])
for (int col = 0; col < nx; ++col)
    for (int b = 0; b < blk; ++b)
        if (row + b < nx)
            acc[b] += mat[(row + b) * nx + col] * src[col];
```

Case Study dMVM – OpenMP V9

- Write back accumulator array in a fixed-length loop

```
#pragma omp target teams distribute
for (int row = 0; row < nx; row += blk) {
    tpe acc[blk] = {};

    // fill acc

    for (int b = 0; b < blk; ++b)
        if (row + b < nx)
            dest[row + b] = acc[b];
}
```

Case Study dMVM – OpenMP V9



Case Study dMVM – **A40** Comparison