Optimization of a Solver for Computational Materials and Structures Problems on NVIDIA Volta and AMD Instinct GPUs

Mohammad Zubair
Old Dominion University
Norfolk, Virginia

James Warner
NASA Langley Research Center (LaRC)
Hampton, Virginia

David Wagner
NASA Langley Research Center (LaRC)
Hampton, Virginia

This research was supported in part by the
NASA Langley Research Center High Performance Computing Incubator program.

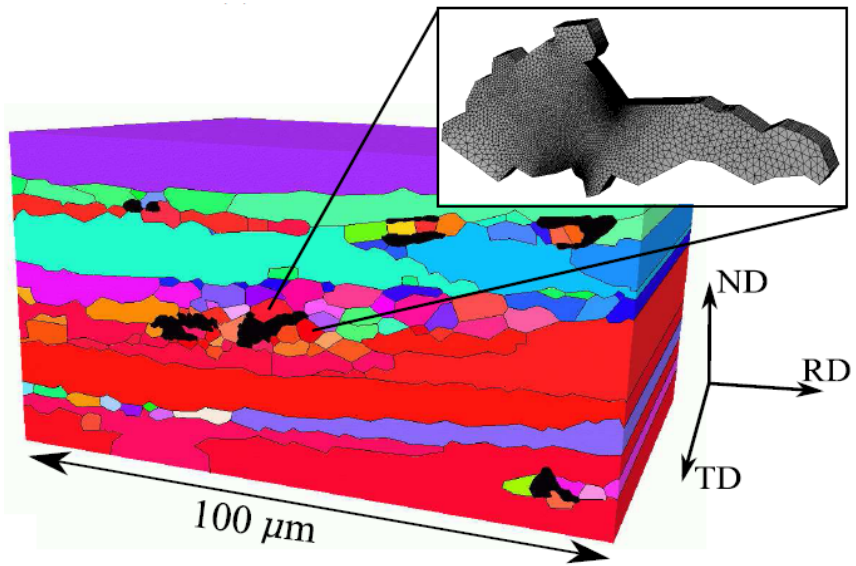
Motivation and Background

SCIFEN (software.nasa.gov/software/LAR-18720-1)

Scalable Implementation of Finite Elements by NASA

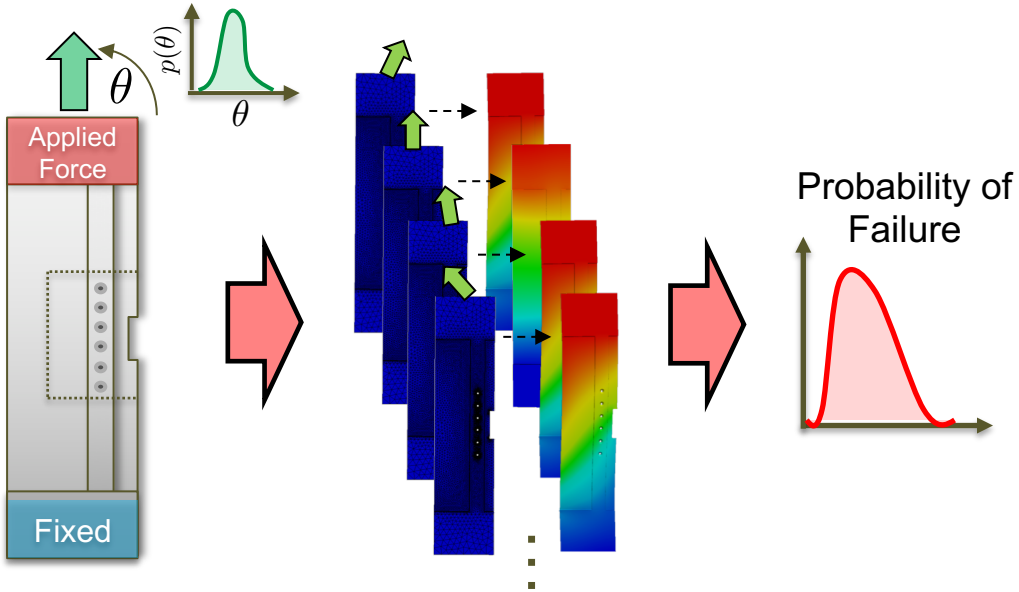
- Leverages high performance computing to solve complex computational materials and structures problems with target applications ranging from:

Solution of single large-scale problem



e.g., Microstructure Analysis

Solution of thousands of moderate-sized problems

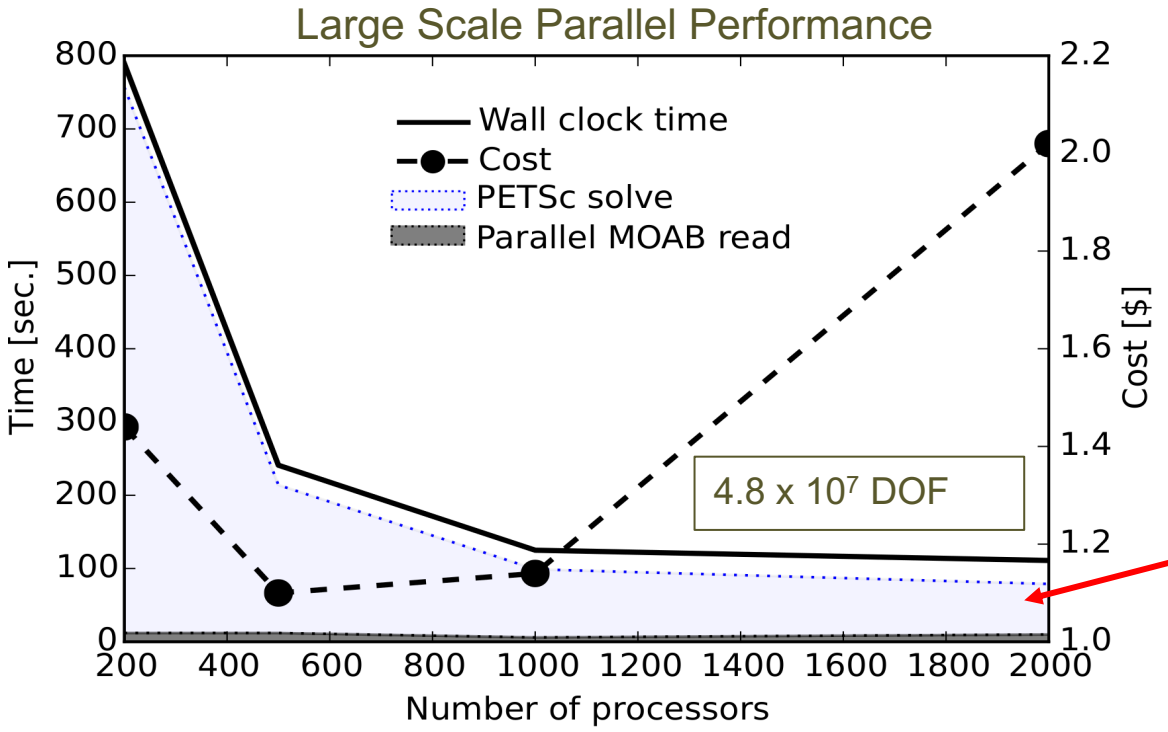


e.g., Uncertainty Quantification

SCIFEN (software.nasa.gov/software/LAR-18720-1)

Scalable Implementation of Finite Elements by NASA

- Relies on the Portable Extensible Toolkit for Scientific Computation (PETSc) libraries for parallel linear algebra routines (e.g., the solution of a linear system of equations)



Linear solver dominates overall computation time

The linear solver was isolated in a mini-app to focus code optimization (https://software.nasa.gov/software/LAR-19417-1)

Porting ScIFEN Solver on Emerging HPC Architectures

- Many cores CPU (e.g. Intel Skylake, ARM ThunderX2)
  - Parallelization across many cores
  - Memory and cache utilization
  - Vectorization
  - NUMA issues

- Accelerator: GPU (e.g. NVIDIA V100 and AMD MI25/60)
  - Fine grain data parallelism
  - Memory and cache utilization
  - SIMT architecture (Warp/Subgroup)
  - Accelerator Issue (Heterogeneous computing)

NVIDIA GPU Volta V100 is used in the Summit system deployed at Oak Ridge National Laboratory.

The new exascale system, Frontier, will be using AMD Radeon Instinct GPU (MI60?)

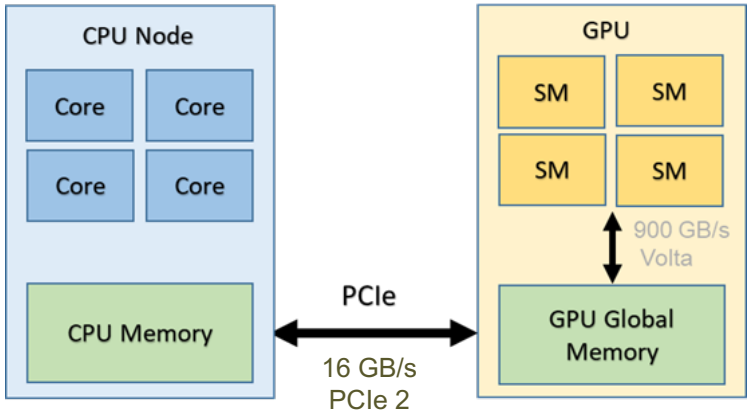
NOTE: AMD MI25 GPU is a lower-end model of AMD Instinct series based on the Vegas architecture, and for a fair comparison we should consider MI60 model. Unfortunately, we do not have access to MI60, but as our computation is memory-bound, the percentage of theoretical peak bandwidth metric is a good indicator of performance on the higher model.

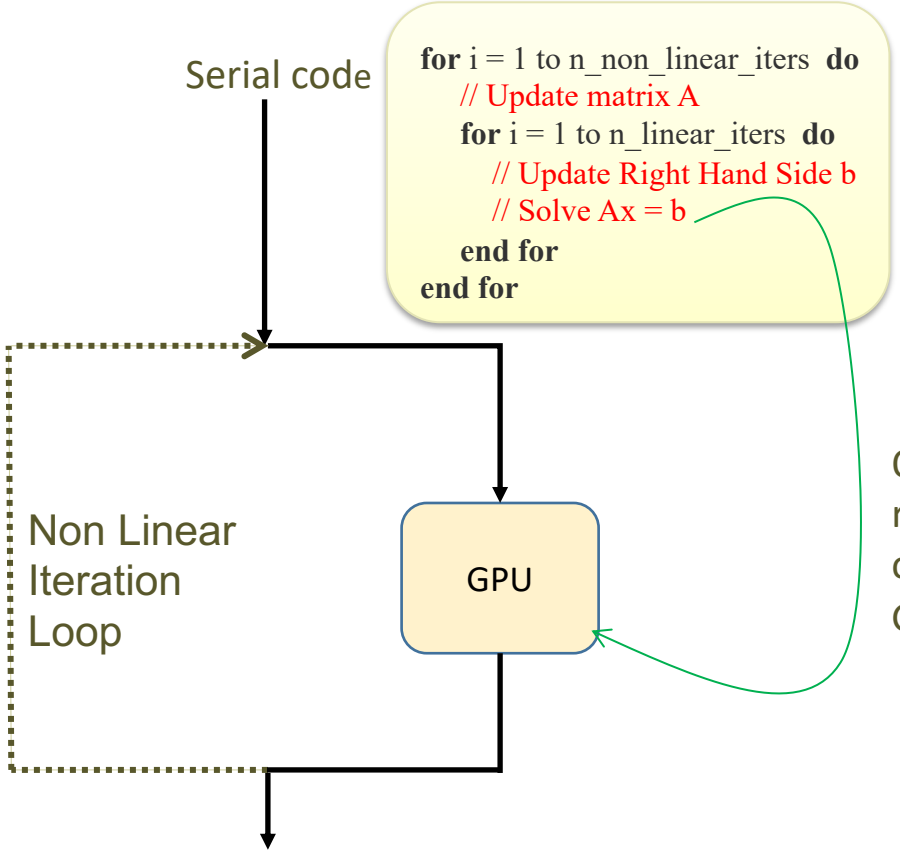
NVIDIA GPU Vs AMD GPU

- Programming Environment
  - NVIDIA has CUDA, a C/C++ API for programming their GPUs
  - AMD has developed HIP, a C/C++ API to program their GPUs

- Target GPUs NVIDIA V100 GPU and AMD MI 25.
  - Ratio between double precision and single precision performance on V100 is 0.5, and on MI 25 it is 0.0625
  - Memory bandwidth on V100 is 900 GB/s and on MI 25 it is 484 GB/s
  - Warp size is 32 on V100 and it is 64 on MI 25,
  - Atomic operations are not supported in the hardware on AMD MI25.

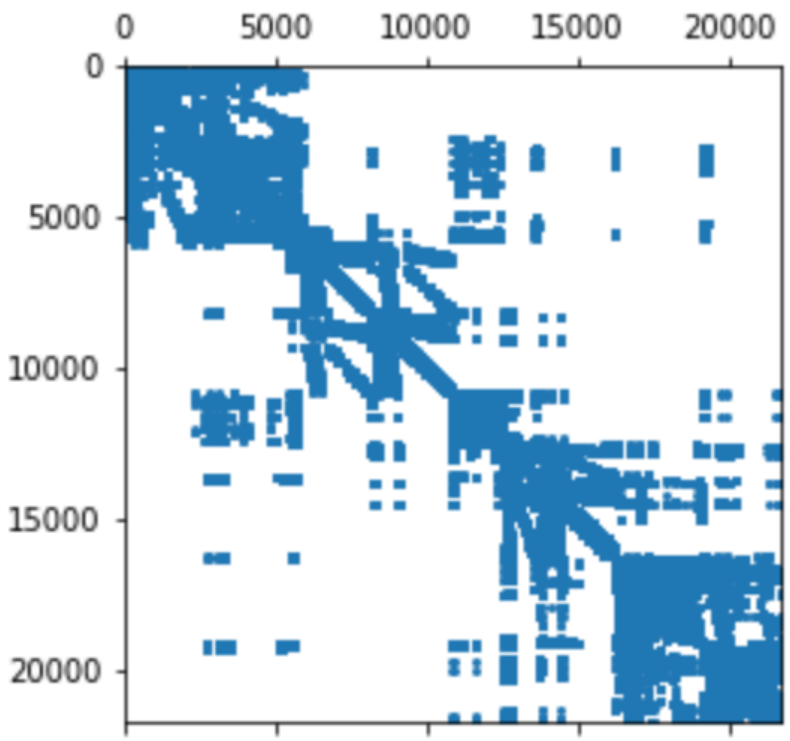
Accelerator Issues-ScIFEN

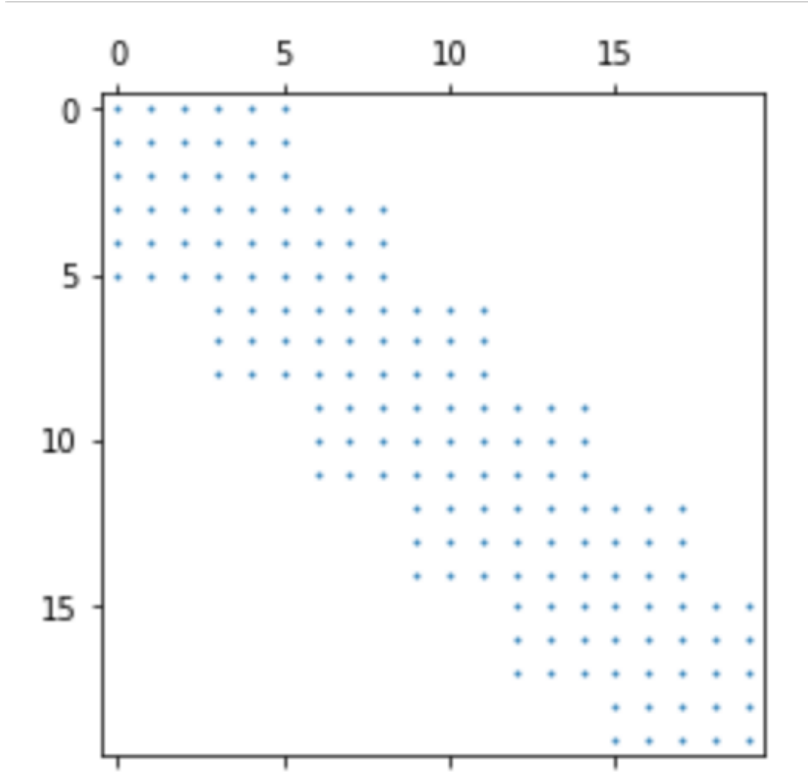




Offload linear solver – requires frequent movement of data between CPU and GPU

ScIFEN Solver (Ax = b): Matrix Exploration





Non-zero appears in blocks – this can be exploited for efficient matrix vector multiplication on HPC architectures

Iterative ScIFEN Solver

Algorithm 1 PRECONDITIONED CR($\mathbf{A}$, $\mathbf{M}$, $\mathbf{b}$, tol)

$\mathbf{d_x} = 0$
$\mathbf{r} = \mathbf{M} * \mathbf{b}$ (*custom GPU kernel*)
$\mathbf{a_r} = \mathbf{A}\mathbf{r}$ (*cusparseDcsrmv*)
$r_{old} = \mathbf{r^t}\mathbf{a_r}$ (*cublasDdot*)
$bnorm = \sqrt{\mathbf{b^t}\mathbf{b}}$ (*cublasDdot*)
$\mathbf{p} = \mathbf{r}$
$\mathbf{a_p} = \mathbf{A}\mathbf{p}$ (*cusparseDcsrmv*)
$r_2 = \mathbf{r^t}\mathbf{r}$ (*cublasDdot*)
while $\sqrt{r_2} > tol * bnorm$ **do**
    $\mathbf{m_p} = \mathbf{M} * \mathbf{a_p}$ (*custom GPU kernel*)
    $denom = \mathbf{a_p^t}\mathbf{m_p}$ (*cublasDdot*)
    $\alpha = r_{old}/denom$
    $\mathbf{d_x} = \mathbf{d_x} + \alpha\mathbf{p}$ (*cublasDaxpy*)
    $\mathbf{r} = \mathbf{r} - \alpha\mathbf{m_p}$ (*cublasDaxpy*)
    $\mathbf{a_r} = \mathbf{A}\mathbf{r}$ (*cusparseDcsrmv*)
    $r_{new} = \mathbf{r^t}\mathbf{a_r}$ (*cublasDdot*)
    $\beta = r_{new}/r_{old}$
    $\mathbf{p} = \mathbf{r} + \beta\mathbf{p}$ (*cublasDscal, cublasDaxpy*)
    $\mathbf{a_p} = \mathbf{a_r} + \beta\mathbf{a_p}$ (*cublasDscal, cublasDaxpy*)
    $r_{old} = r_{new}$
    $r_2 = \mathbf{r^t}\mathbf{r}$ (*cublasDdot*)
end while
return $\mathbf{d_x}$

- Conjugate residual iterative scheme works well for a class of ScIFEN applications
- Can be easily formulated using dense and sparse BLAS

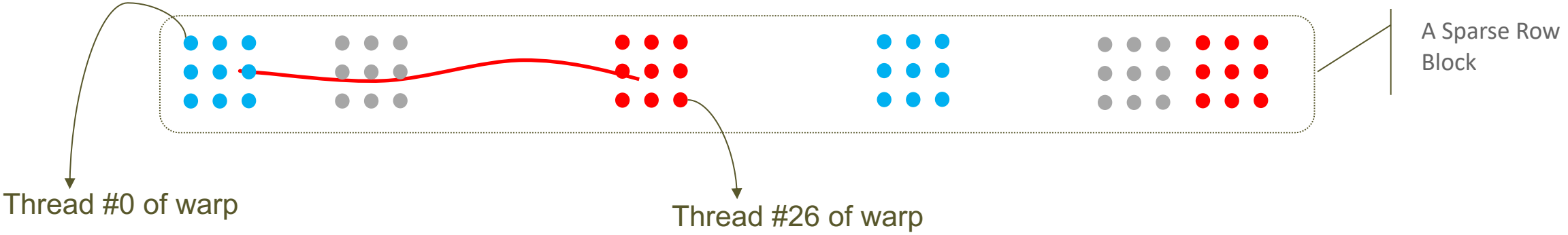
Performance Optimization of Iterative Solver

- Iterative solver computation is dominated by sparse matrix-vector operation
  - Storage provided in the ScIFEN Solver Mini-App is in CSR format
    - Converted storage format to a more memory efficient block CSR format
    - Our experimentations indicated NVIDIA library for sparse matrix vector operations can be improved. For the AMD GPU, we did not find a sparse library to multiply a block CSR matrix with a vector.
    - We developed an efficient implementation of a block CSR matrix with a vector operation for NVIDIA and AMD GPU
- Merge Level-1 BLAS routines wherever possible and developed a custom implementation of them that minimizes the memory traffic.

Block Sparse Matrix Vector Operation on NVIDIA GPU

Warp Size: 32

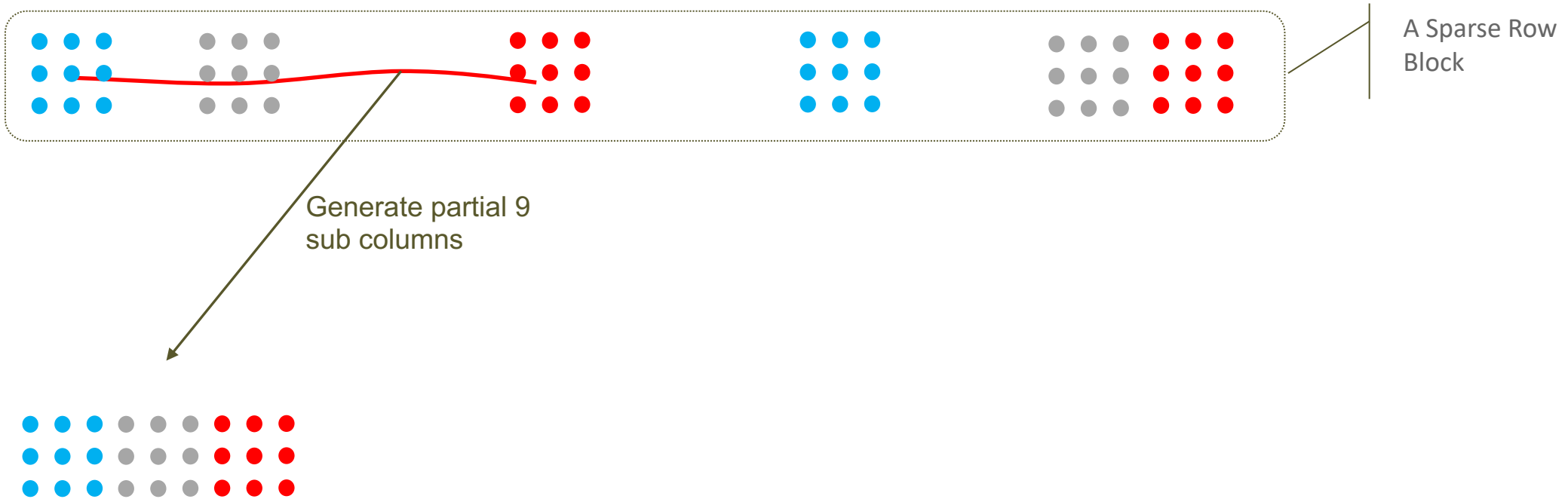
Block sparse matrix vector operation involve multiplying a row block with appropriate elements of the vector



Assign a warp to three consecutive non-zero blocks. This enables consecutive threads in the warp to load consecutive locations in the memory and process these points to create partial results

Block Sparse Matrix Vector Operation on NVIDIA GPU

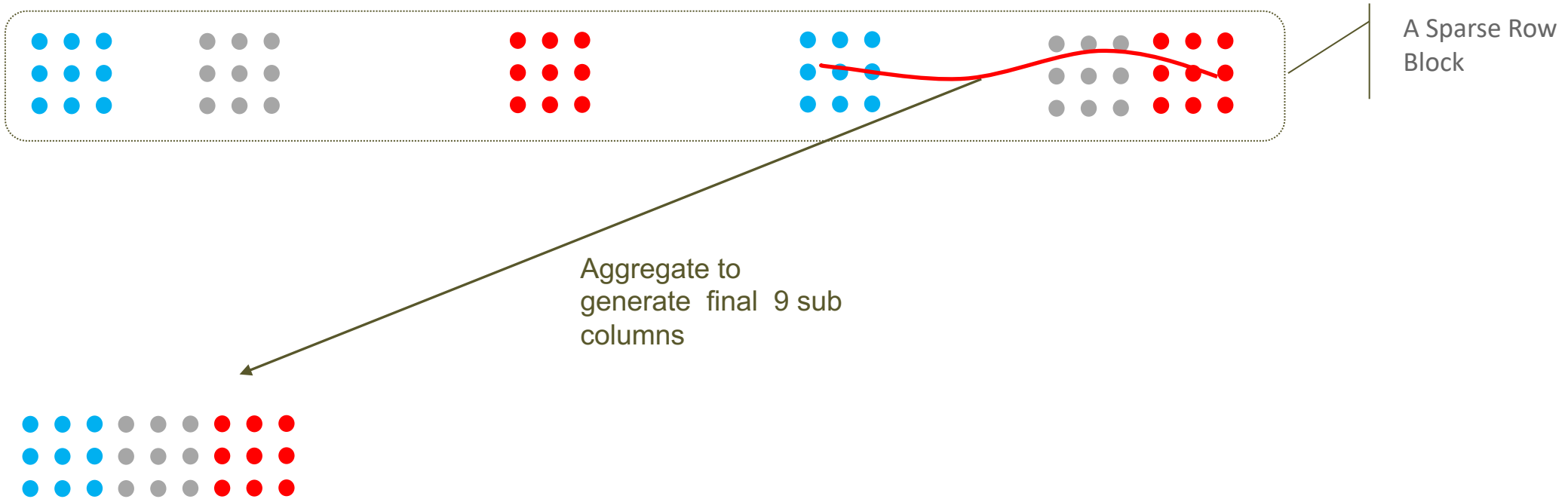
```
//  loop over a block row while processing three non-zero blocks at a time
for (j = istart; j < iend-nbk+1; j+=nbk) {
   fk += A(k, l, j+nbki) * x(l, jam[j+nbki]);
}
```



First iteration of the `for` loop

Block Sparse Matrix Vector Operation on NVIDIA GPU

```
//  loop over a block row while processing three non-zero blocks at a time
for (j = istart; j < iend-nbk+1; j+=nbk) {
   fk += A(k, l, j+nbki) * x(l, jam[j+nbki]);
}
```



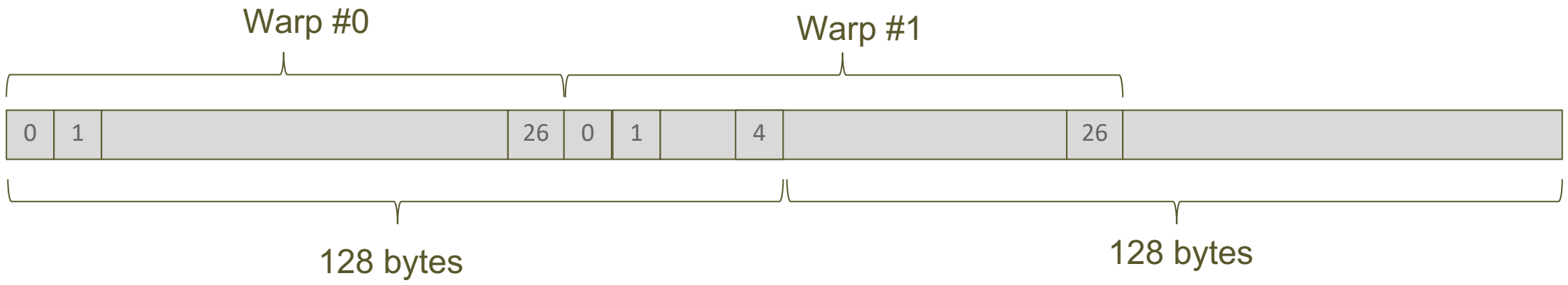
Second iteration of the `for` loop

Block Sparse Matrix Vector Operation on NVIDIA GPU

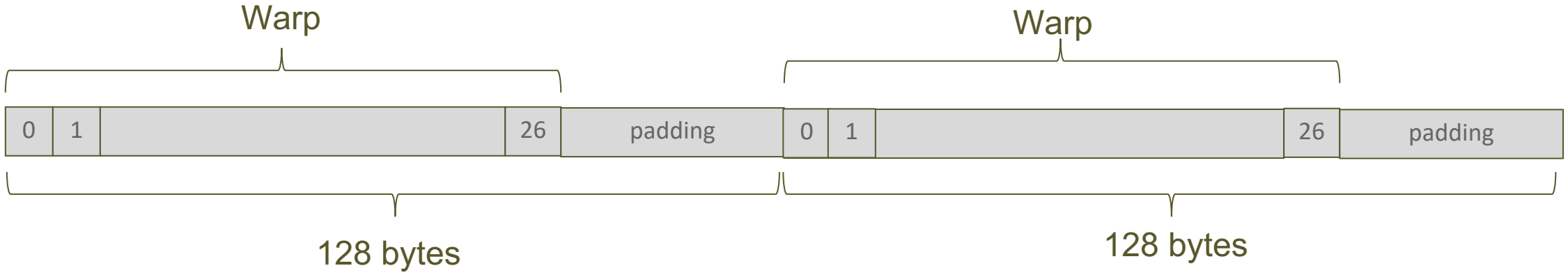
```
//  use shuffle to reduce 9 columns into a single column
f1 = fk;
f1 = f1 + __shfl_down_sync(0xffffffe0, f1, 1 * 3);
f1 = f1 + __shfl_down_sync(0xffffffe0, f1, 2 * 3);
f1 = f1 + __shfl_down_sync(0xffffffe0, f1, 4 * 3);
f1 = f1 + __shfl_down_sync(0xffffffe0, fk, 8 * 3);
```

Memory Alignment Issue

Unaligned Access



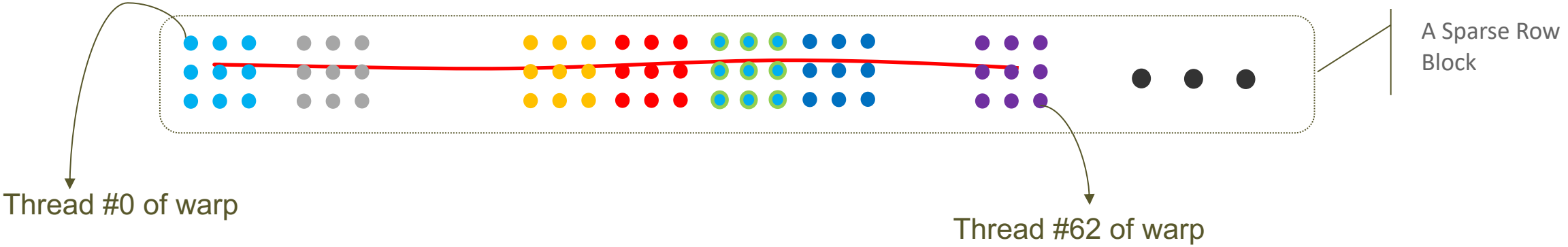
Can padding help?



Block Sparse Matrix Vector Operation on AMD GPU

Warp Size: 64

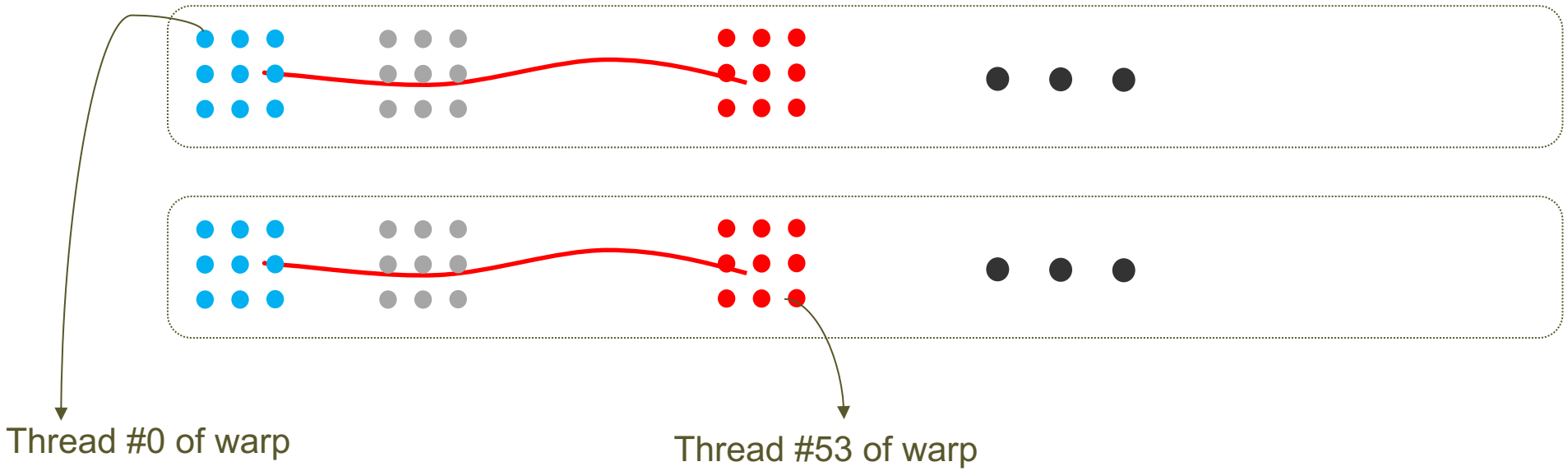
One block row per warp



Assign a warp to seven consecutive non-zero blocks. This enables consecutive threads in the warp to load consecutive locations in the memory and process these points to create partial results

Block Sparse Matrix Vector Operation on AMD GPU

Warp Size: 64
Two blocks row per warp



A half-warp processes three consecutive non-zero blocks. Note that in this algorithm, it is not necessary that the two consecutive block-rows have an identical number of non-zero blocks. Consequently, not all the 54 threads of a warp will be active all the time.

Merge Level-1 BLAS

Algorithm 1 PRECONDITIONED CR($\mathbf{A}$, $\mathbf{M}$, $\mathbf{b}$, tol)

$\mathbf{d_x} = 0$
$\mathbf{r} = \mathbf{M} * \mathbf{b}$ (*custom GPU kernel*)
$\mathbf{a_r} = \mathbf{A}\mathbf{r}$ (*cusparseDcsrmv*)
$r_{old} = \mathbf{r^t}\mathbf{a_r}$ (*cublasDdot*)
$bnorm = \sqrt{\mathbf{b^t}\mathbf{b}}$ (*cublasDdot*)
$\mathbf{p} = \mathbf{r}$
$\mathbf{a_p} = \mathbf{A}\mathbf{p}$ (*cusparseDcsrmv*)
$r_2 = \mathbf{r^t}\mathbf{r}$ (*cublasDdot*)
while $\sqrt{r_2} > tol * bnorm$ **do**
  $\mathbf{m_p} = \mathbf{M} * \mathbf{a_p}$ (*custom GPU kernel*)
  $denom = \mathbf{a_p^t}\mathbf{m_p}$ (*cublasDdot*)
  $\alpha = r_{old}/denom$
  $\mathbf{d_x} = \mathbf{d_x} + \alpha\mathbf{p}$ (*cublasDaxpy*)
  $\mathbf{r} = \mathbf{r} - \alpha\mathbf{m_p}$ (*cublasDaxpy*)
  $\mathbf{a_r} = \mathbf{A}\mathbf{r}$ (*cusparseDcsrmv*)
  $r_{new} = \mathbf{r^t}\mathbf{a_r}$ (*cublasDdot*)
  $\beta = r_{new}/r_{old}$
  $\mathbf{p} = \mathbf{r} + \beta\mathbf{p}$ (*cublasDscal, cublasDaxpy*)
  $\mathbf{a_p} = \mathbf{a_r} + \beta\mathbf{a_p}$ (*cublasDscal, cublasDaxpy*)
  $r_{old} = r_{new}$
  $r_2 = \mathbf{r^t}\mathbf{r}$ (*cublasDdot*)
end while
return $\mathbf{d_x}$

Merge Vector-Vector with Ddot

Merge two Daxpys and a Ddot

Merge Dscal with Daxpy

Merge Level-1 BLAS

- Reduction operation is one of the critical operations in merging Level-1 BLAS which has to be implemented efficiently
  - Different implementations on NVIDIA GPU and AMD GPU because of different warp sizes (32 Vs 64)

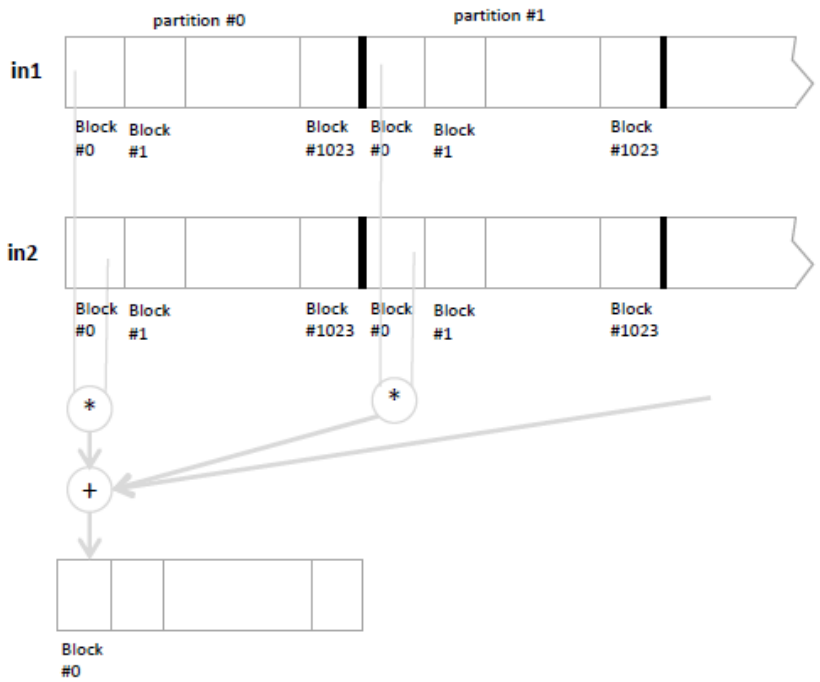
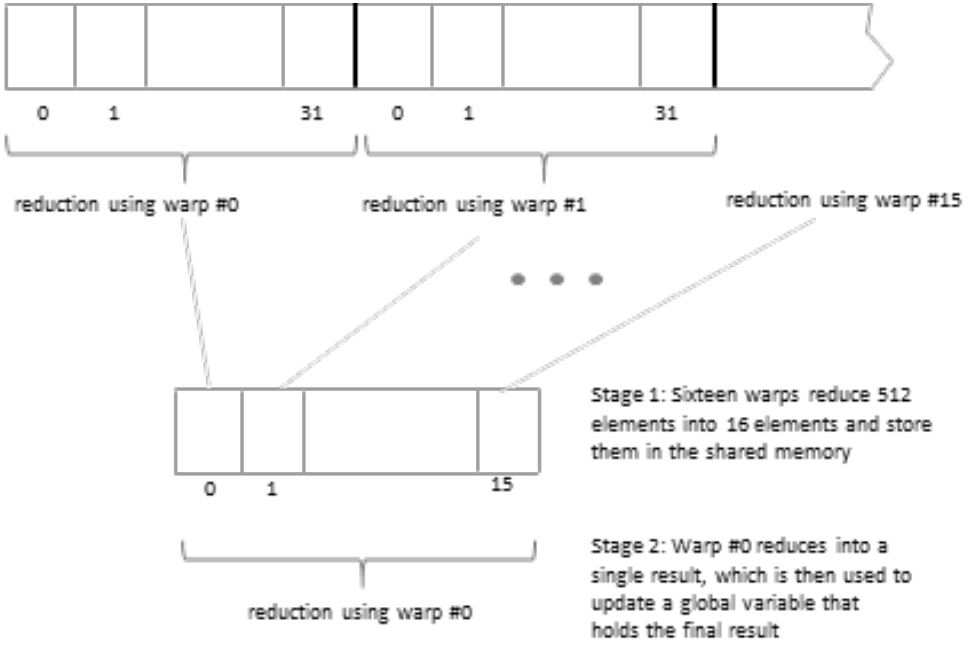


Fig. 2: Grid-stride loop to multiply the two arrays and perform partial reduction to create a single partition.

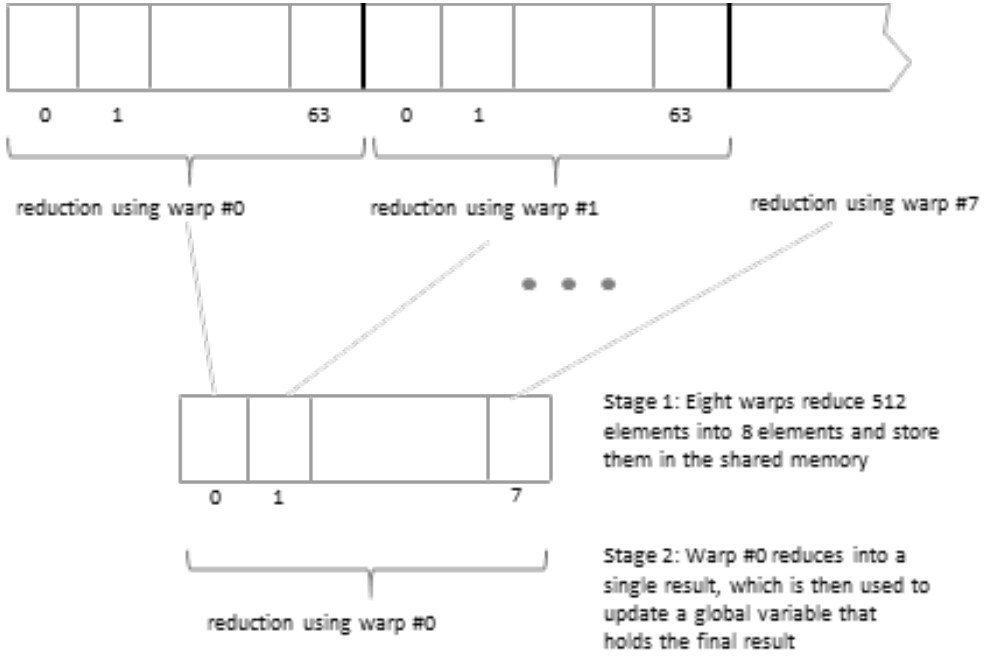
Merge Level-1 BLAS

- Reduction operation is one of the critical operations in merging Level-1 BLAS which has to be implemented efficiently
  - Different implementations on NVIDIA GPU and AMD GPU because of different warp sizes (32 Vs 64)



Warp Size: 32 (NVIDIA GPU) A block of 512 elements is aggregated using 16 warps. The 512 elements in a block are partitioned in 16 chunks each of size 32. A chunk is aggregated by a warp.

NVIDIA GPU



Warp Size: 64 (AMD GPU) A block of 512 elements is aggregated using 8 warps. The 512 elements in a block are partitioned in 8 chunks each of size 64. A chunk is aggregated by a warp.

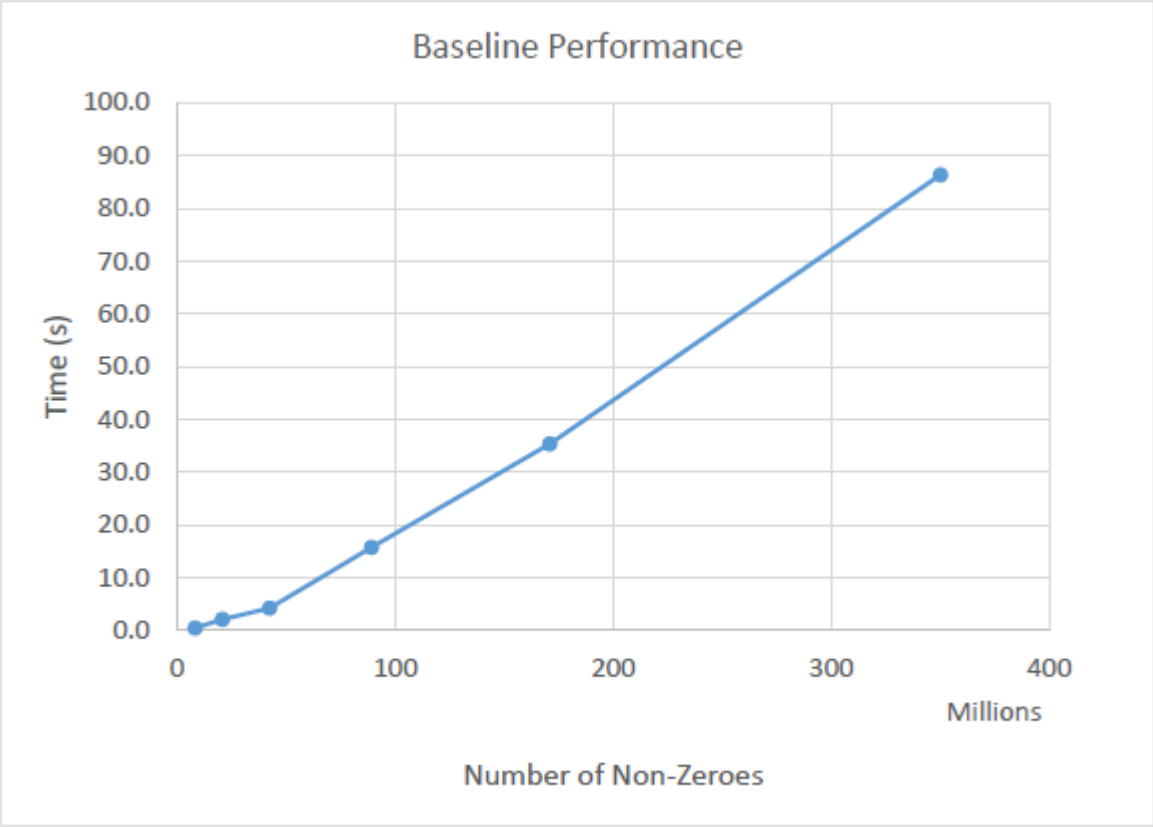
AMD GPU

Test Matrices

Matrix	N	Total Non-Zeros	Avg Non-Zeros Per Row
100K	100,704	8,201,934	81
250K	251,733	20,732,877	82
500K	511,167	42,448,329	83
1M	1,070,391	89,131,761	83
2M	2,046,246	170,856,450	83
4M	4,185,654	350,095,572	84

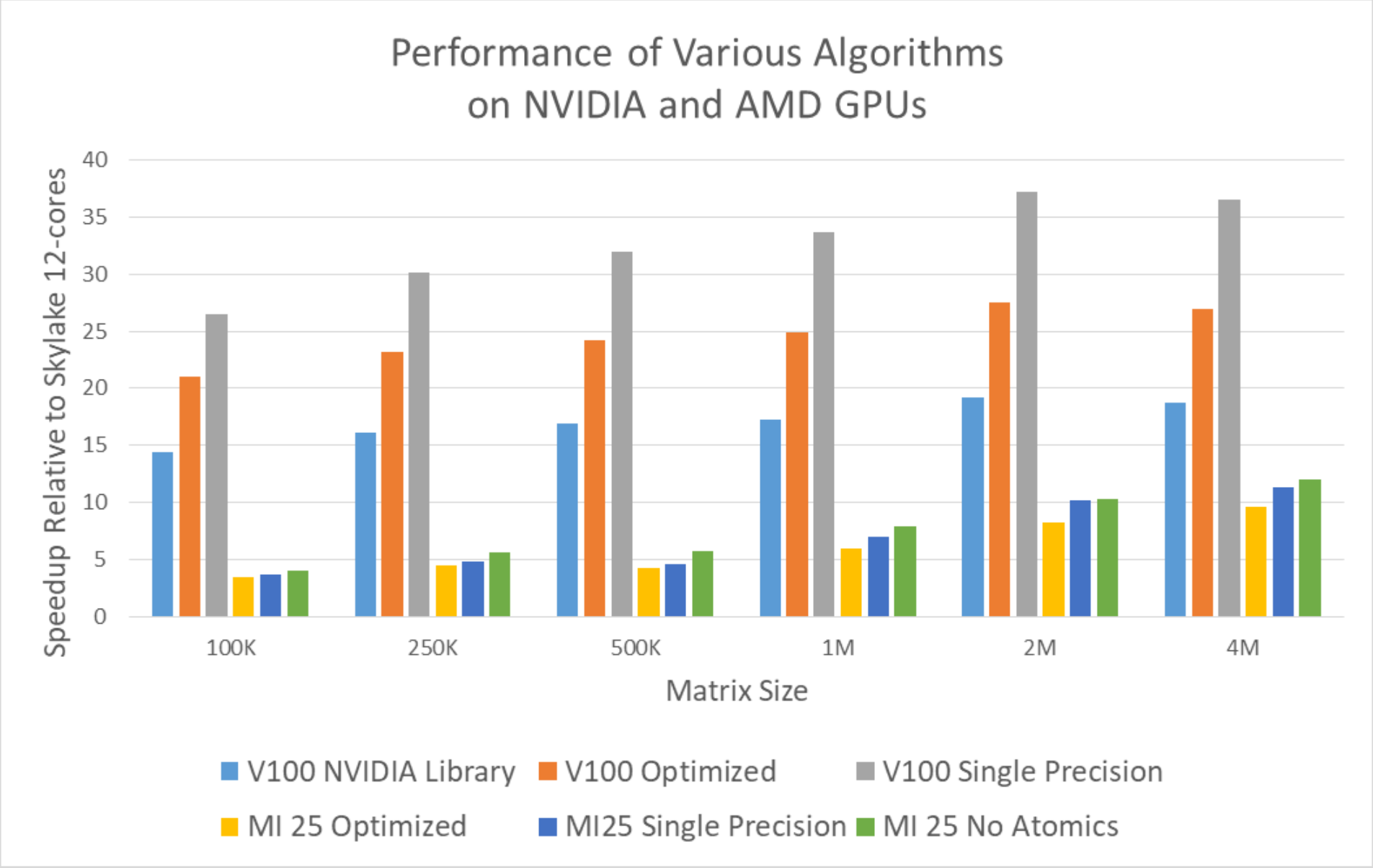
Structure of Test Matrices for Performance Results

Performance Results



Baseline performance of solver uses PETSc library on a 12-cores Intel Skylake 3104 CPU.

Performance Results



Performance Results

Matrix	V100 Double Precision Bandwidth (GB/s)	Percentage of Theoretical Peak Bandwidth (V100)	MI 25 Double Precision Bandwidth (GB/s)	Percentage of Theoretical Peak Bandwidth (MI 25)
100K	605	67%	208	43%
250K	663	74%	249	51%
500K	688	76%	224	46%
1M	698	78%	236	49%
2M	712	79%	265	55%
4M	713	79%	284	59%

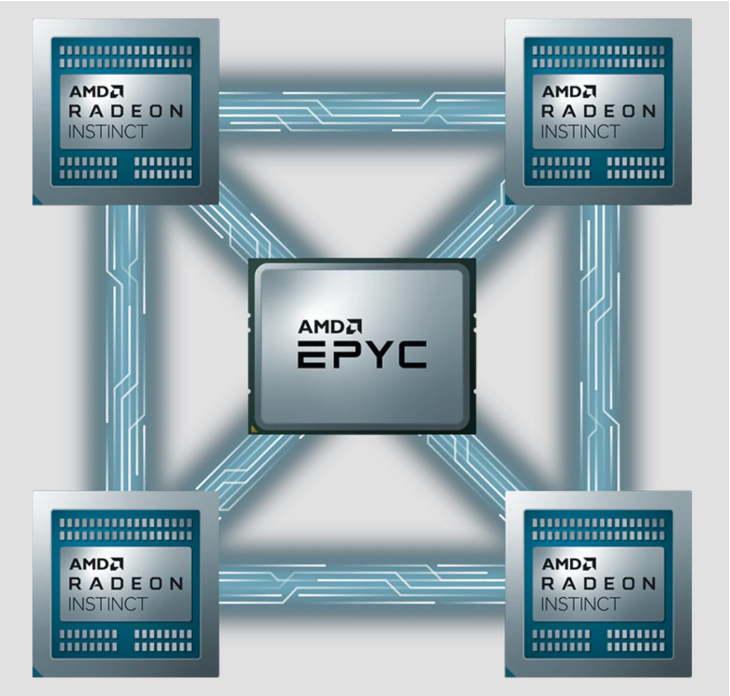
Conclusion and Future Work

- Scientific kernels can greatly benefit from emerging high-performance architectures such as NVIDIA and AMD GPUs.
- For achieving performance on these architectures it is necessary to put effort in careful planning and optimization of computation intensive kernels.
  - GPU issues: Fine grain data parallelism, Memory and cache utilization, SIMT architecture (Warp/Subgroup), Accelerator Issue (Heterogeneous computing)
- Port and optimize linear system construction on GPUs

Backup Slides

AMD GPU

- The new Frontier system at Oak Ridge will be based on AMD EPYC CPU with Radeon Instinct Accelerator. Peak Performance > 1.5 exaflops
- Four GPU per CPU.
- Radeon Instinct GPU models: MI25, MI50, MI60 (all three models are based on Vegas architecture)



RadeonOpenCompute (ROCm) - Open Source Platform for HPC and GPU Computing

- LLVM compiler foundation
- HCC C++ and HIP for application portability
  - HIP (Heterogeneous-compute Interface) is a C++ runtime API and kernel language that allows developers to create portable applications that can run on AMD and other GPU's
- GCN assembler and disassembler

CUDA Code Conversion for AMD GPU

Use hipify tool

CUDA Code (solver.cu)

```
__global__ void cuda_point5_kernel
(int solve_sign, int  start, int end,
 const int* __restrict__  iam, const int* __restrict__  jam,
 const float* __restrict__  a_off, const double* __restrict__  a_diag_lu,
 float *dq, const double* __restrict__  res)
{

    __shared__ float
    fs[5][BLOCK_DIM_Y];
    __shared__ float
    sm_f[5][5][BLOCK_DIM_Y];
    __shared__ double
    a_diag_lu_shared[5][5][BLOCK_DIM_Y];

    int const k = threadIdx.x % 5;
    int const l = threadIdx.x / 5;
    int n = start + blockIdx.x * blockDim.y + threadIdx.y - 1;
```

```
for ( ; j < iend; j++) {
    jam0 = jam[j];
    fk += A_OFF(k, l, j) *  DQ(l, jam0 - 1);
}
sm_f[k][l][threadIdx.y] = fk;
__syncthreads();
```

hipify-perl solver.cu > solver.cpp

NOTE: For this code segment the two codes are identical.

HIP Compatible Code (solver.cpp)

```
__global__ void cuda_point5_kernel
(int solve_sign, int  start, int end,
 const int* __restrict__  iam, const int* __restrict__  jam,
 const float* __restrict__  a_off, const double* __restrict__  a_diag_lu,
 float *dq, const double* __restrict__  res)
{

    __shared__ float
    fs[5][BLOCK_DIM_Y];
    __shared__ float
    sm_f[5][5][BLOCK_DIM_Y];
    __shared__ double
    a_diag_lu_shared[5][5][BLOCK_DIM_Y];

    int const k = threadIdx.x % 5;
    int const l = threadIdx.x / 5;
    int n = start + blockIdx.x * blockDim.y + threadIdx.y - 1;
```

```
for ( ; j < iend; j++) {
    jam0 = jam[j];
    fk += A_OFF(k, l, j) *  DQ(l, jam0 - 1);
}
sm_f[k][l][threadIdx.y] = fk;
__syncthreads();
```