Porting a state-of-the-art communication avoiding Krylov subspace solver to P100 GPUs

Yussuf Ali[1,2], Takuya Ina[1,2], Naoyuki Onodera[1], Yasuhiro Idomura[1]

[1] Japan Atomic Energy Agency

[2] Visible Information Center

1. Introduction

- ◆ Krylov solvers for the pressure Poisson equation occupy ~90% of the computing cost
- ◆ MPI collective functions become critical point in the computation
- ◆ We ported a Communication-Avoiding (CA) Krylov solver algorithm to the GPU
- ◆ Much faster than the CPU version
- ◆ Tested on the TSUBAME and Reedbush GPU clusters

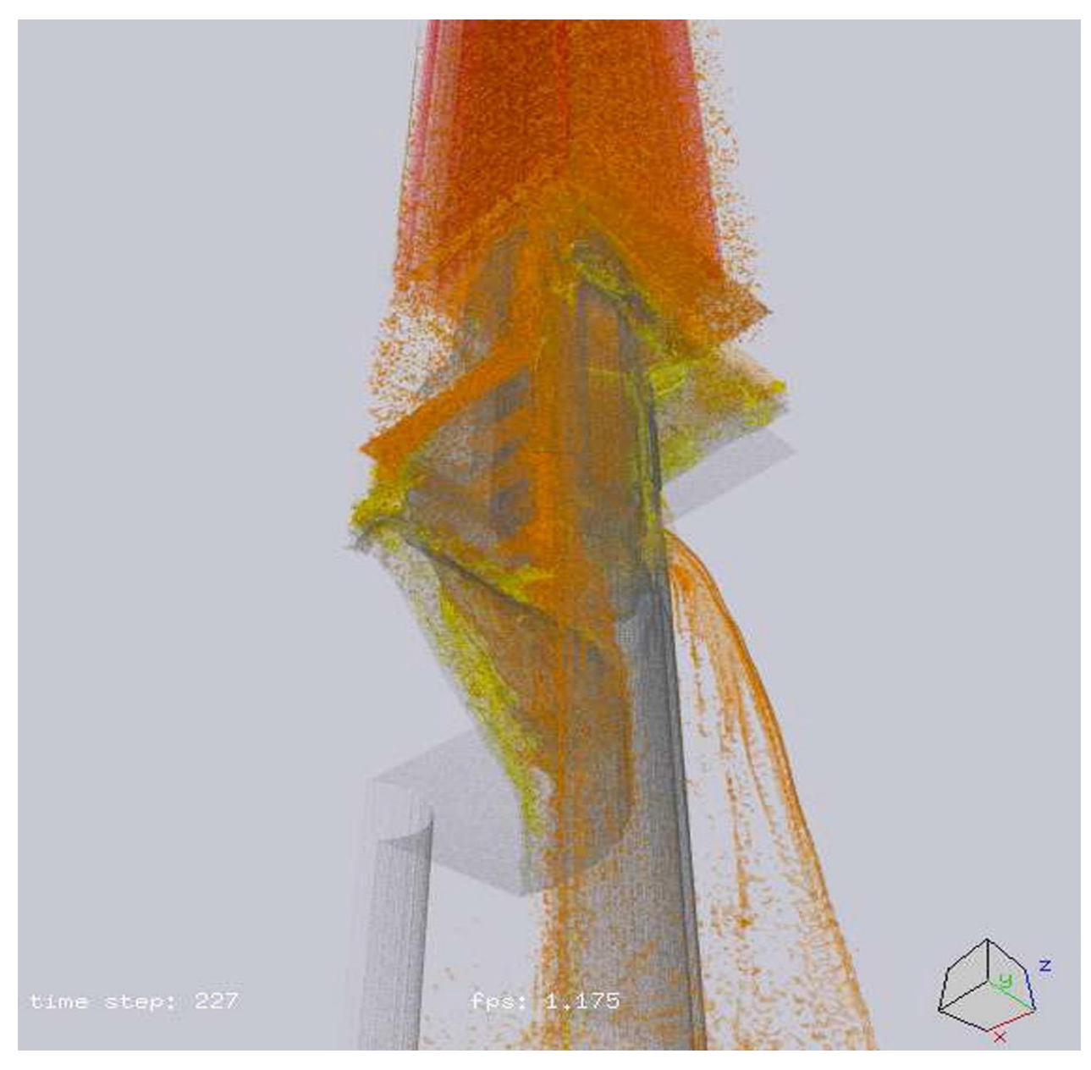


Figure 1: Visualized output of the solver showing molten parts inside a reactor

1. What is "communication-avoiding (CA) " ?

- **Communication** = communication between different nodes which take part in the same computation over the network (typically MPI function calls) [1]
- **Communication** = **very expensive** compared to floating point operations
- **Communication-avoiding** = avoid communication by clustering together several steps of computation before sending data
- **Krylov s-step method** = calculate **s-steps** at once before sending the data → Communication reduced by a factor of **s**

2. The Preconditioned Chebyshev Basis communication-avoiding CG algorithm (P-CBCG)

- P-CBCG calculates **s** vectors at once (blue box in *Fig. 2*) in our case s = 12
- **s** vectors together form a matrix with **n rows** and **s columns** with **n >> s**, a so called **Tall and Skinny matrix**
- P-CBCG contains many *Tall and Skinny matrix* operations (red boxes in *Fig. 2*)

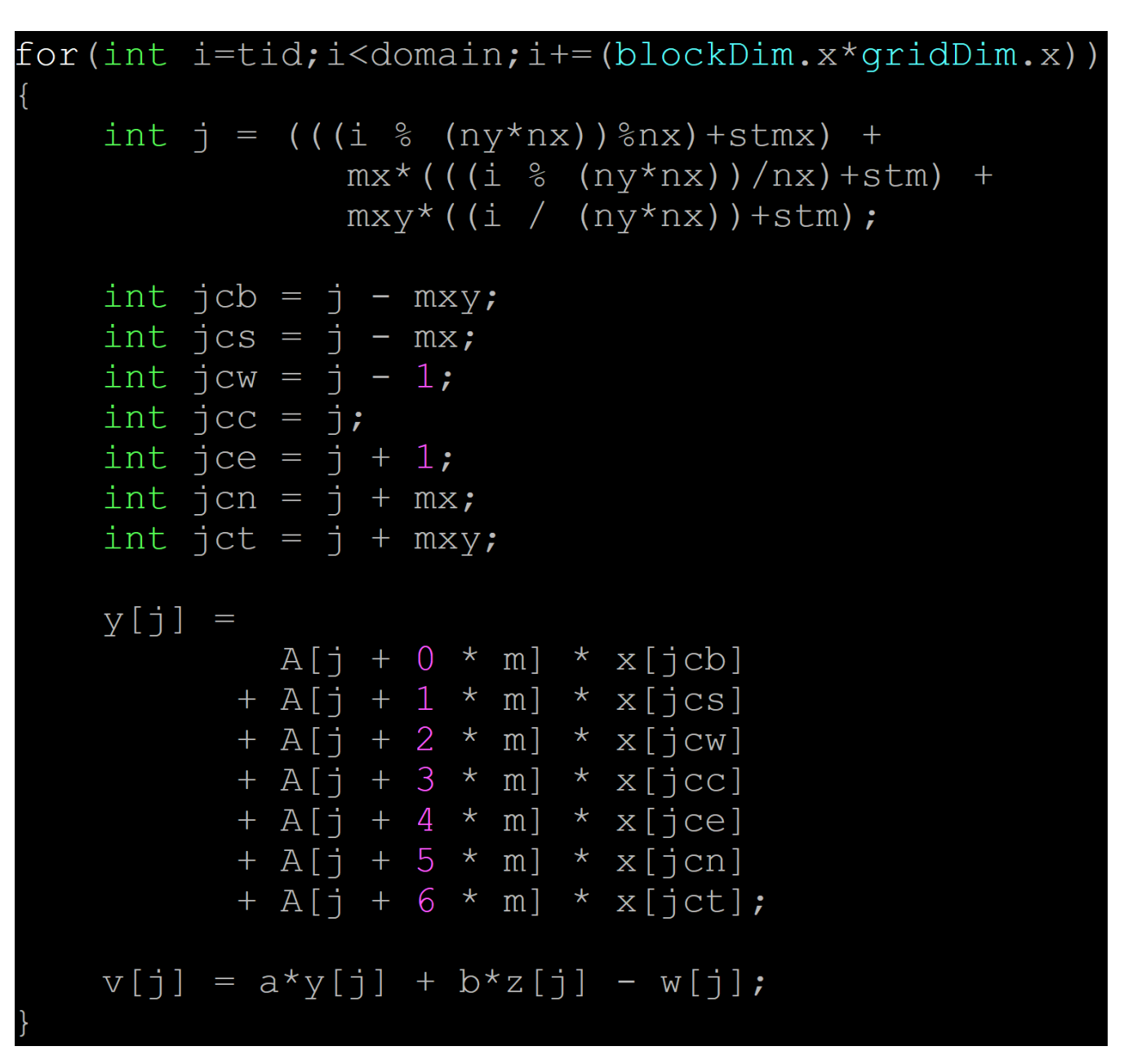
```
Input: $Ax = b$, Initial guess $x_0$
Output: Approximate solution $x_i$
 1: $r_0 := b - Ax_0$
 2: Compute $S_0$ $(T_0(AM^{-1})r_0, T_1(AM^{-1})r_0, ..., T_{s-1}(AM^{-1})r_0)$
 3: $Q_0 = S_0$
 4: for $k = 0, 1, 2, ...$ until convergence do
 5:     Compute $Q_k^* AQ_k$                        Tall and Skinny matrix operations
 6:     Compute $Q_k^* r_{sk}$
 7:     $a_k := (Q_k^* AQ_k)^{-1} Q_k^* r_{sk}$
 8:     $x_{s(k+1)} := x_{sk} + Q_k a_k$            s-steps of SpMV and preconditioning
 9:     $r_{s(k+1)} := r_{sk} - AQ_k a_k$
10:     Compute $S_{k+1}$ $(T_0(AM^{-1})r_{s(k+1)}, T_1(AM^{-1})r_{s(k+1)}, ..., T_{s-1}(AM^{-1})r_{s(k+1)})$
11:     Compute $Q_k^* AS_{k+1}$
12:     $B_k := (Q_k^* AQ_k)^{-1} Q_k^* AS_{k+1}$     Tall and Skinny matrix operations
13:     $Q_{k+1} := S_{k+1} - Q_k B_k$
14:     $AQ_{k+1} := AS_{k+1} + AQ_k B_k$
15: end for
```

Figure 2: The P-CBCG algorithm calculates s-steps in one iteration (approx. of largest Eigenvalue before the main loop not shown in the figure)

2. Implementation

1. Sparse Matrix Vector product (SpMV)

- Compressed row storage (CRS) sparse matrix format does not provided optimal memory access pattern for threads within a warp → no coalesced memory access
- The **Diagonal** (**DIA**) format is used inside the SpMV kernel
- **DIA** format maps very well to the GPU hardware because coalesced memory access can be guaranteed
- Triple nested loop of the CPU version was replaced by a grid strided loop as seen in *Fig. 3*



```
for(int i=tid;i<domain;i+=(blockDim.x*gridDim.x))
{
    int j = (((i % (ny*nx))%nx)+stmx) +
                mx*(((i % (ny*nx))/nx)+stm) +
                mxy*((i / (ny*nx))+stm);

    int jcb = j - mxy;
    int jcs = j - mx;
    int jcw = j - 1;
    int jcc = j;
    int jce = j + 1;
    int jcn = j + mx;
    int jct = j + mxy;

    y[j] =
            A[j + 0 * m] * x[jcb]
          + A[j + 1 * m] * x[jcs]
          + A[j + 2 * m] * x[jcw]
          + A[j + 3 * m] * x[jcc]
          + A[j + 4 * m] * x[jce]
          + A[j + 5 * m] * x[jcn]
          + A[j + 6 * m] * x[jct];

    v[j] = a*y[j] + b*z[j] - w[j];
}
```

Figure 3: SpMV kernel code in CUDA

2. Tall and Skinny (TS) matrix operations

- **TS** matrix = matrix with **rows** >> **columns**
- All BLAS **gemm** implementations perform very poorly on **TS** matrices as seen in *Fig. 4* and [3]
- In addition our data is **non-continuous** in memory because halo regions for data exchange are contained
- Solution = **CUBLAS gemmBatched** [3]
- **gemmBatched** = high performance on TS matrices + skipping of halo regions *Fig. 5*

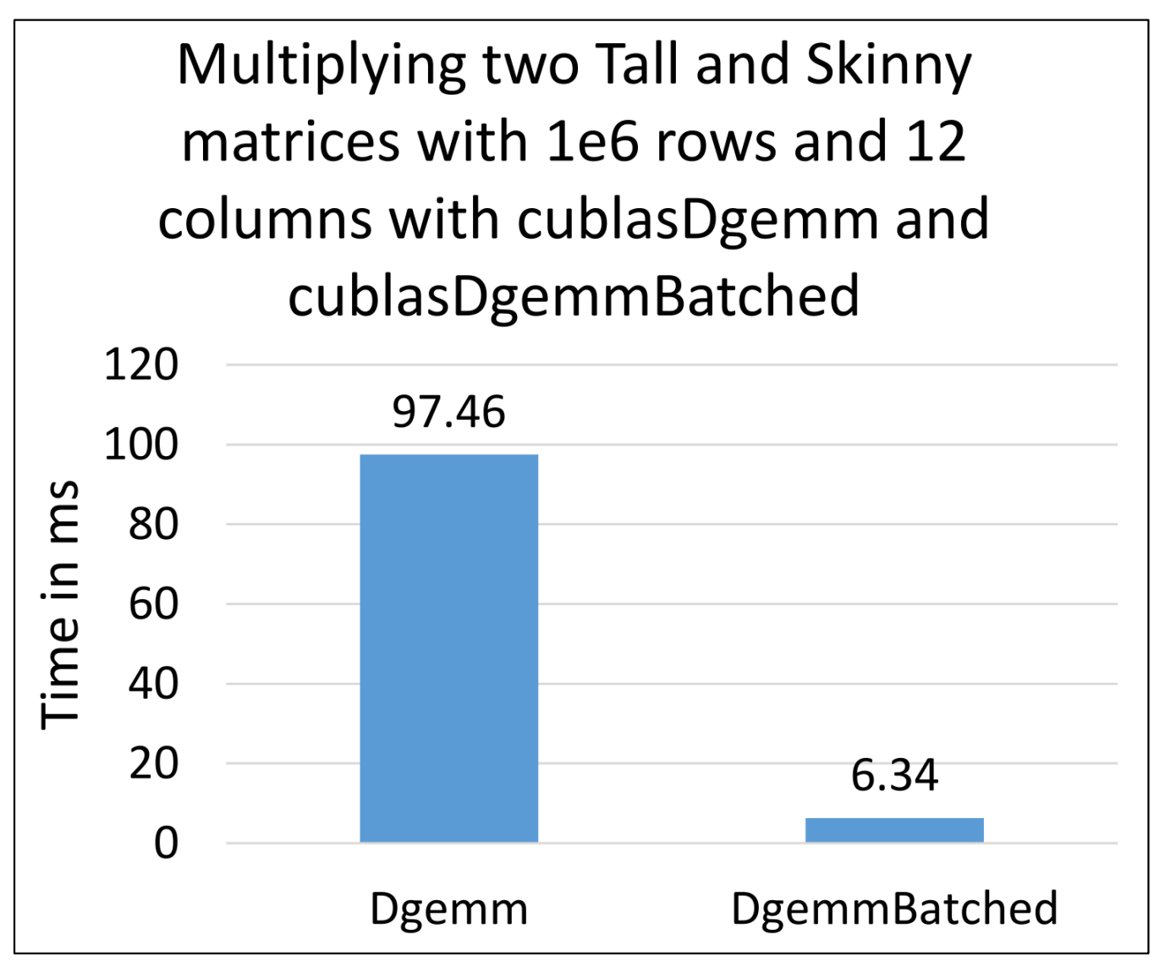


Figure 4: dgemmBacthed performance inc. reduction

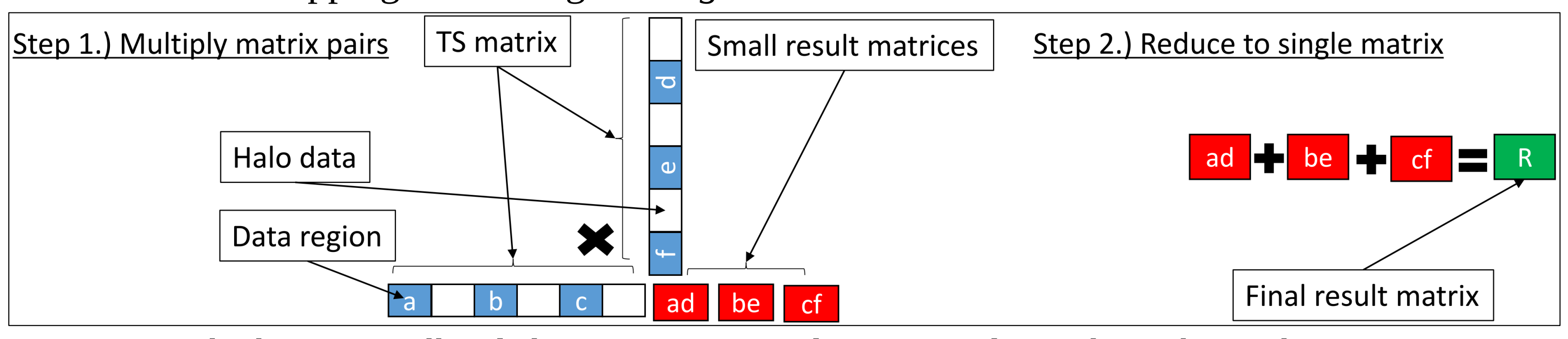


Figure 5: Multiplying two Tall and Skinny matrices + reduction in order to obtain the result matrix R

3. Block-Jacobi preconditioning

- On the CPU the domain is divided among OpenMP threads along the z-axis *Fig. 6*
- CPU approach does not map to the GPU hardware because of the memory access pattern
- **Solution**: divide into more smaller areas *Fig. 6*
- Smaller areas will result in higher performance on the GPU
- Each area is processed by one thread
- Side-effect of smaller areas = Convergence property changes
- GPU version needs much more iterations until convergence

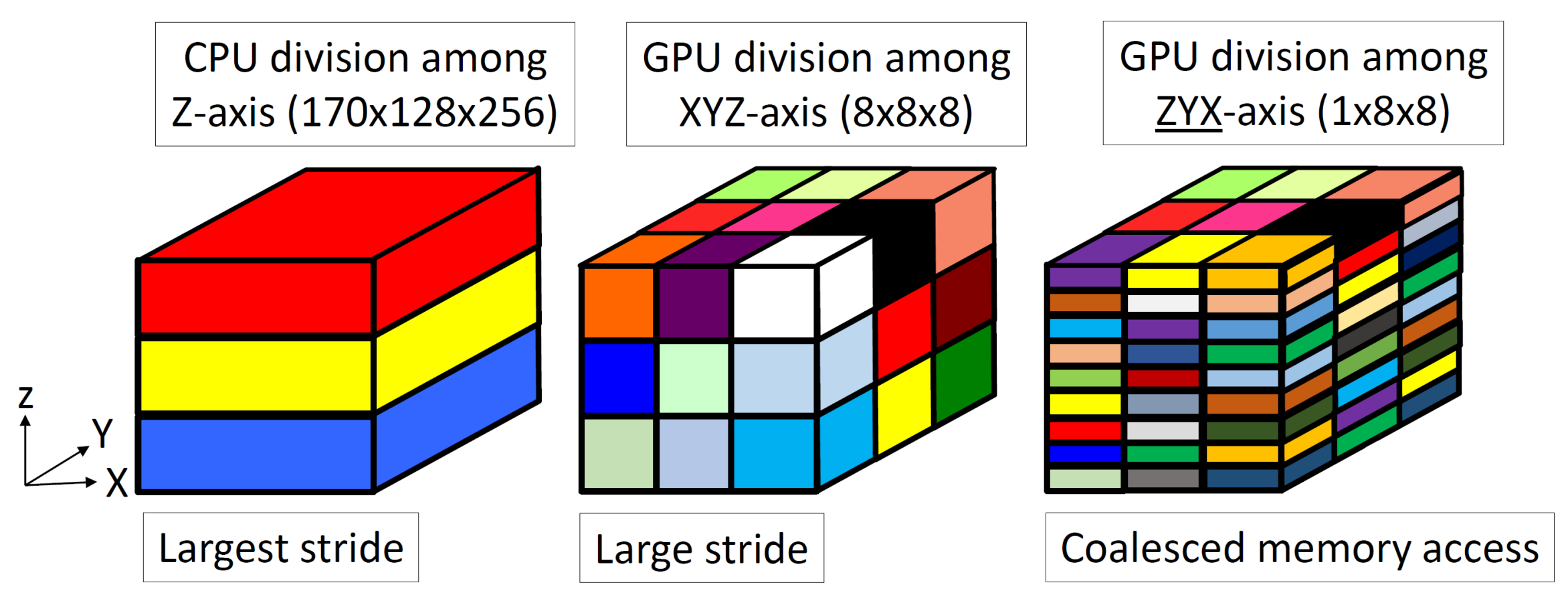

CPU division among Z-axis (170x128x256)
GPU division among XYZ-axis (8x8x8)
GPU division among ZYX-axis (1x8x8)
Z
Y
X
Largest stride
Large stride
Coalesced memory access

Figure 6: Different area settings for the preconditioning

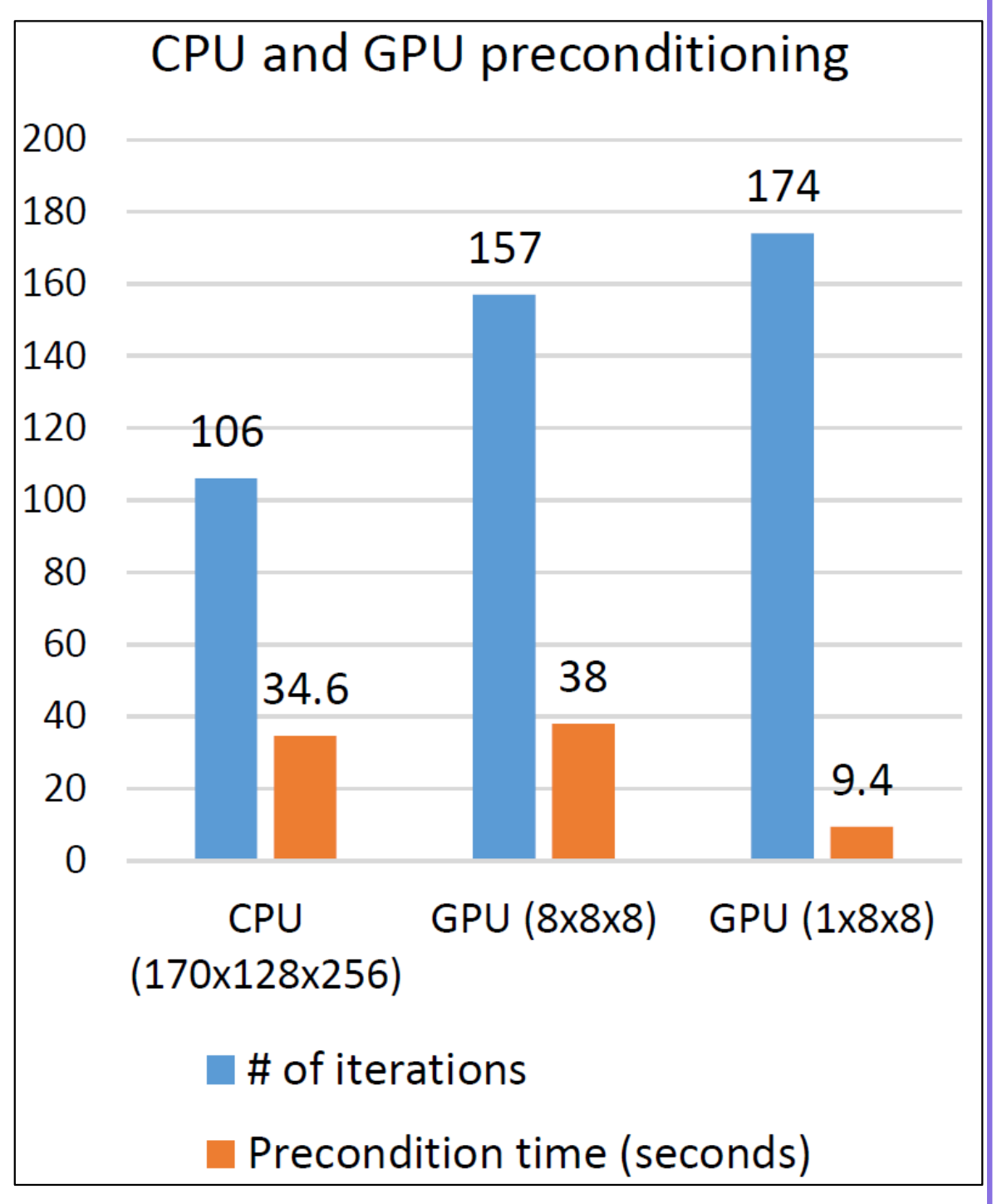


Figure 7: CPU and GPU preconditioning performance

- The area setting *1x8x8* results in 70% more iterations until convergence but has the highest performance compared to the CPU and to the *8x8x8* GPU area setting *Fig. 7*

3. Performance results

- GPU version is compared to a highly optimized CPU version of the same algorithm [2]
- *Table 1* shows the arithmetic intensity and the roofline ratio of the CPU and GPU version

Table 1: Roofline evaluation

Kernel	Arith. Intensity		Roofline ratio	
	CPU	GPU	CPU	GPU
SpMV+Pre.	0.13	0.16	0.81	0.89
TS Matrix	1.12	0.79	0.91	0.88

Table 2: Specifications of the JAEA ICEX, Reedbush GPU and Tsubame GPU cluster

	ICEX (CPU)	Reedbush (GPU)	Tsubame (GPU)
Intel compiler and CUDA version	17	17 and CUDA 9	16 and CUDA 8
MPI	Intel MPI	MVAPICH-GDR 2.3a	OpenMPI 1.10.7
Hardware	Xeon (Haswell)	NVIDIA P100	NVIDIA P100
Peak performance flops [Gflops]	480	5300	5300
STREAM bandwidth [GB/sec]	58	550	550
Interconnect	InfiniBand (4x FDR)	InfiniBand (4x EDR 2 link)	Omi-Path HFI 100Gps x 4

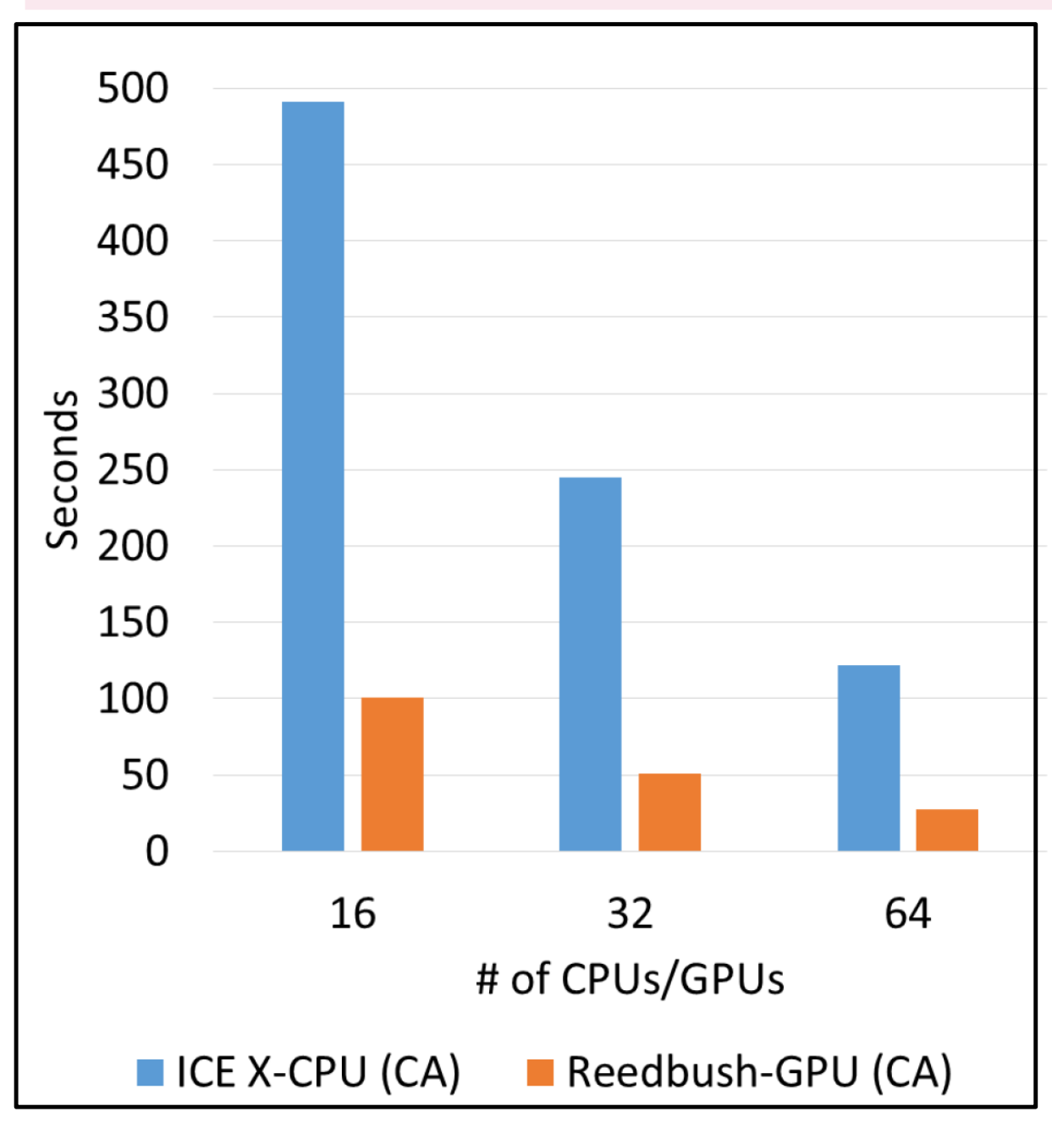


Figure 8: CPU comp. to GPU Domain: 512x320x2048

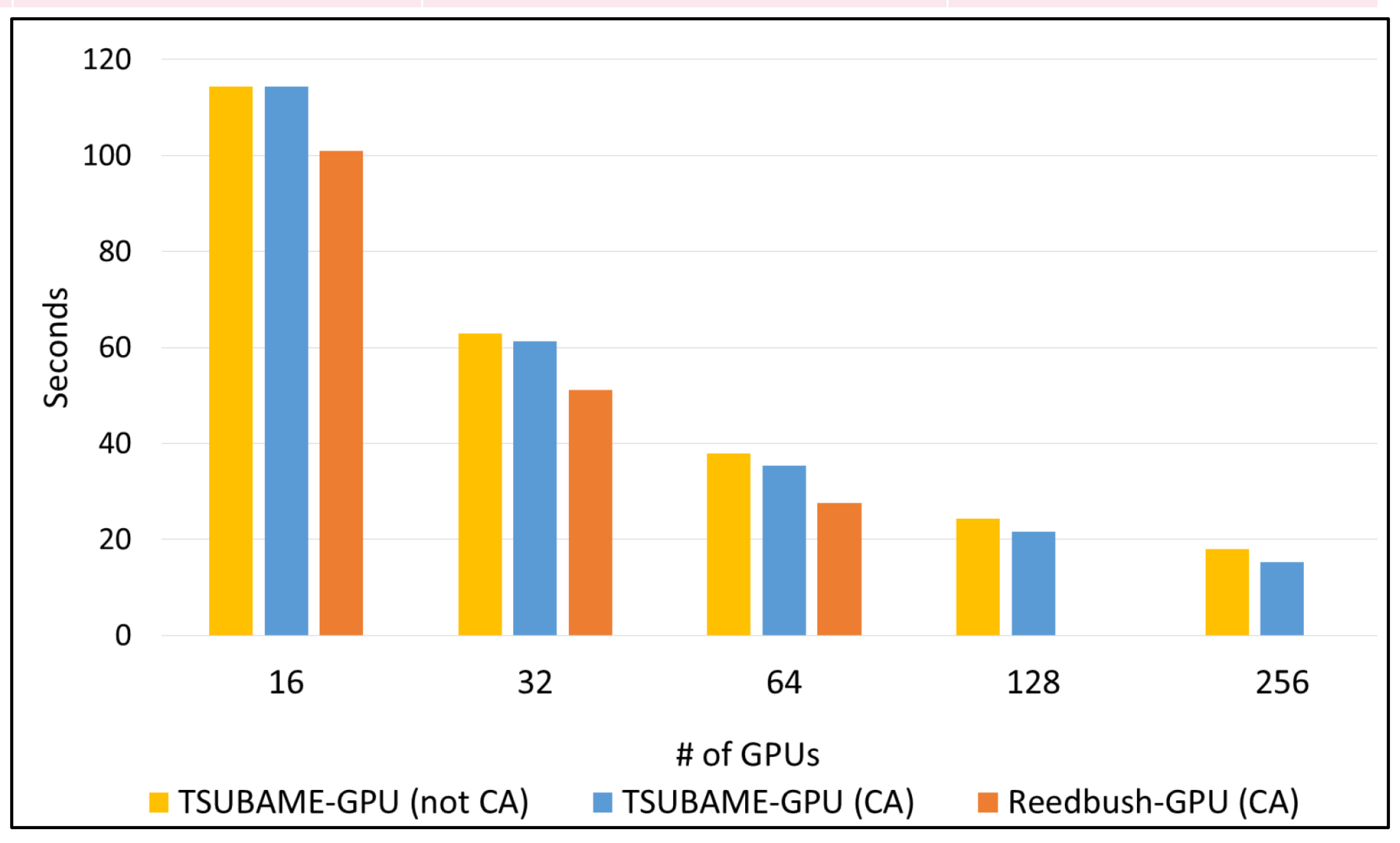


Figure 9: Strong scaling on TSUBAME and Reedbush Domain: 512x320x2048

- **Observation 1** = The GPU versions is much faster then the CPU version as seen in *Fig. 8*
- **Observation 2** = As seen in *Fig. 9* on the Reedbush GPU cluster the algorithm executes the fastest
- **Observation 3** = With the CA version of the algorithm the MPI_Allreduce communication cost could be significantly reduced as seen in *Fig. 10*
- This year the algorithm will also be tested on the new Summit supercomputer with up to 27,000 GPUs

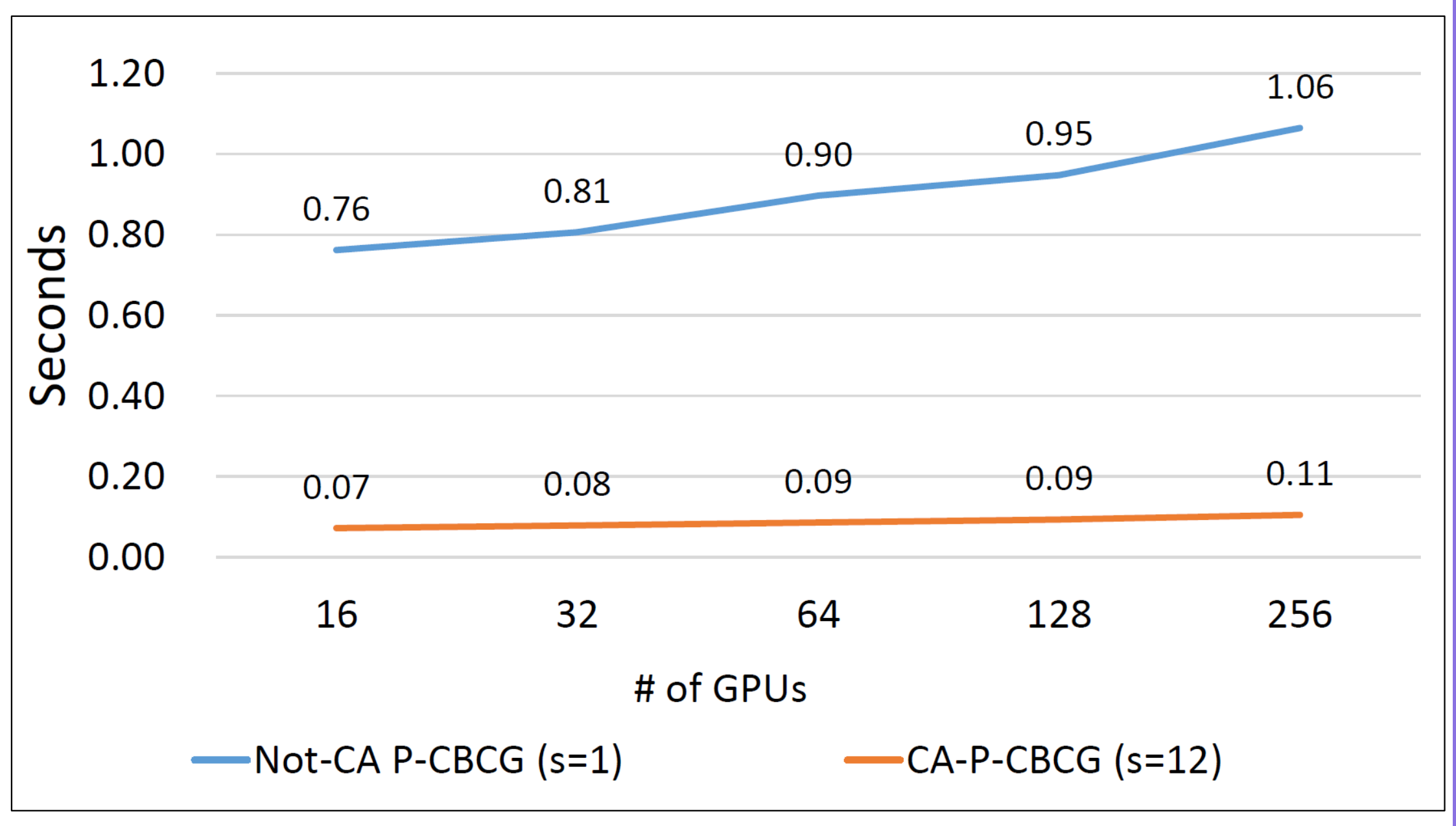


Figure 10: MPI_Allreduce comm. cost comparison on TSUBAME

4. Summary

- ◆ Ported the complete P-CBCG to the GPU
- ◆ Block-Jacobi preconditioner achieves high performance on the GPU
- ◆ Reduced the MPI_Allreduce communication cost

[1] Mark Hoemmen. 2010. Communication-Avoiding Krylov Subspace Methods. Ph.D. Dissertation
[2] Idomura Y., et al, 2018, SCFA 2018 , pp. 257 – 273,
[3] I. Yamazaki, H., et al, 2014 IEEE 28th International Parallel and Distributed Processing Symposium, 2014, pp. 382-391.

This research was supported in part by "Joint Usage/Research Center for Interdisciplinary Large-scale Information Infrastructures" in Japan (Project ID: jh170031-NAH)