# Strategies for efficient execution of Pipelined Conjugate Gradient method on GPU systems

Manasi Tiwari and Sathish Vadhiyar

Department of Computational and Data Sciences
Indian Institute of Science, Bangalore, India
{manasitiwari,vss}@iisc.ac.in

**Abstract.** The Preconditioned Conjugate Gradient (PCG) method is widely used for solving linear systems of equations with sparse matrices. A recent version of PCG, Pipelined PCG (PIPECG), eliminates the dependencies in the computations of the PCG algorithm so that the non-dependent computations can be overlapped with communication. In this paper, we develop three methods for efficient execution of the Pipelined PCG algorithm on GPU accelerated heterogeneous architectures. The first two methods achieve task-parallelism using asynchronous executions of different tasks on multi-core CPU and a GPU. The third method achieves data parallelism by decomposing the workload between multi-core CPU and GPU based on a performance model. We performed experiments on both the K40 and V100 GPU systems and our methods give up to 8x speedup and on average 3x speedup over PCG CPU implementation of Paralution and PETSc libraries. They also give up to 5x speedup and on average 1.45x speedup over PCG GPU implementation of Paralution and PETSc libraries. The third method also provides an efficient solution for solving problems that cannot be fit into the GPU memory and gives up to 6.8x speedup for such problems.

**Keywords:** Preconditioned Conjugate Gradient · Pipelined Methods · Heterogeneous Architectures · GPU · Asynchronous executions

## 1 Introduction

Conjugate Gradient (CG) [7] is one of the most widely used iterative methods for finding the solution of linear systems $Ax = b$ with symmetric positive definite sparse matrices. A preconditioner can be applied to the system to condition the input system and to improve convergence.

Today's HPC systems have accelerators like GPUs along with traditional multi-core CPUs. The programming models for these accelerators are different from that of the multi-core processors as well. In order to use all the resources available within a compute node efficiently, we must interleave the features in the programming models in such a way that we achieve the best possible performance from the platform.

The main computational kernels in the PCG method are Sparse Matrix Vector Product (SPMV), Preconditioner Application (PC), Vector-Multiply-Adds

(VMAs) and Dot Products. For distributed memory systems, the bottleneck in PCG is the synchronization that occurs on all cores due to the allreduce in the dot products of the algorithm. **Pipelined PCG (PIPECG)** proposed by Ghysels et al [6], on which this work is based, has one allreduce per iteration. By introducing extra VMAs, they eliminate the dependencies between the dot products and PC+SPMV of PCG. The aim of doing this is to overlap the communication introduced by dot products with PC and SPMV. The resulting algorithm offers another advantage which makes it a perfect candidate for our hybrid executions on a single node-single GPU system. As the PC and SPMV in PIPECG do not depend on results of the previous dot products, we can execute them simultaneously on multi-core CPU and GPU. This would require communicating data between CPU and GPU, thus introducing additional costs. We show that by using asynchronous streams efficiently, we can hide the complete time for data movement between CPU and GPU.

We develop three methods for efficient execution of PIPECG on a single node-single GPU system. The first two methods, **Hybrid-PIPECG-1** and **Hybrid-PIPECG-2**, achieve task-parallelism by simultaneous execution of the dot products on multi-core CPU and PC and SPMV on the GPU. They are different in the amount of data that needs to be moved between the CPU and GPU in every iteration of PIPECG. The third method, **Hybrid-PIPECG-3**, achieves data parallelism by decomposing the workload between multi-core CPU and GPU based on a performance model and then using asynchronous data transfers for PIPECG iterations. We use CUDA streams for asynchronous data transfers between CPU and GPU. We performed experiments on both the K40 and V100 GPU systems and our methods give up to 8x speedup and on average 3x speedup over PCG CPU implementation of Paralution and PETSc libraries. Our methods give up to 5x speedup and on average 1.45x speedup over PCG GPU implementation of Paralution and PETSc libraries. Hybrid-PIPECG-3 method also provides an efficient solution for solving problems that cannot be fit into the GPU memory and gives up to 6.8x speedup for such problems.

## 2 Related Work

To achieve optimum performance of the PCG method on GPU systems, many works have concentrated on efficient GPU implementation of the PC kernel. Algebraic Multigrid GPU implementations are presented in [3]. Incomplete LU and Cholesky factorizations on GPUs are presented in [9]. Research works also concentrate on optimizing the most time consuming kernel in PCG, the SPMV kernel [5]. Different sparse matrix formats have been proposed in [4] to improve SPMV performance on GPUs. All the works mentioned above concentrate on kernel executions only on the GPUs. They do not utilize the multi-core CPU present in the system. Our work is different from all the works described above since we aim to utilize all the available resources of the system and accelerate the performance of the PCG method as a whole. Furthermore, our work can be used in conjunction with the enhanced kernels on the GPUs mentioned above.

## 3 Background

**PCG:** PCG introduced by Hestenes [7] is given in Algorithm 1. The computational kernels in PCG are SPMV in line 10, PC in line 15, VMAs in lines 9, 13 and 14 and dot products in lines 11, 16 and 17. In PCG, we can see that the operation in every line depends on the operation in the previous line. There are no independent computations in each iteration which can be executed simultaneously.

**PIPECG:** PIPECG was proposed by Ghysels et al. [6]. As shown in Algorithm 2, PIPECG introduces extra VMAs (on lines 11,12,13,17,18) to remove the dependencies between the dot products (lines 19,20,21) and PC (line 22) and SPMV (line 23) so that PC and SPMV can be computed while dot products are being computed. We can use PIPECG for our hybrid executions as the dot products can be executed on the CPU while PC+SPMV can be executed simultaneously on GPU as they are not dependent on each other. This strategy helps us utilize all the resources in the GPU accelerated node and achieve optimum performance.

**Algorithm 1** PCG Method

1: $r_0 = b - Ax_0$; $u_0 = M^{-1}r_0$;
2: $\gamma_0 = (u_0, r_0)$; $norm_0 = \sqrt{(u_0, u_0)}$
3: **for** i=0,1... **do**
4: **if** $i > 0$ **then**
5: $\beta_i = \gamma_i/\gamma_{i-1}$
6: **else**
7: $\beta_i = 0$
8: **end if**
9: $p_i = u_i + \beta_i p_{i-1}$
10: $s = Ap_i$
11: $\delta = (s, p_i)$
12: $\alpha = \gamma_i/\delta$
13: $x_{i+1} = x_i + \alpha p_i$
14: $r_{i+1} = r_i - \alpha s$
15: $u_{i+1} = M^{-1}r_{i+1}$
16: $\gamma_{i+1} = (u_{i+1}, r_{i+1})$;
17: $norm_{i+1} = \sqrt{(u_{i+1}, u_{i+1})}$
18: **end for**

**Algorithm 2** PIPECG Method

1: $r_0 = b - Ax_0$;$u_0 = M^{-1}r_0$;$w_0 = Au_0$;
2: $\gamma_0 = (r_0, u_0)$;$\delta = (w_0, u_0)$;$norm_0 = \sqrt{(u_0, u_0)}$
3: $m_0 = M^{-1}w_0$; $n_0 = Am_0$
4: **for** i=0,1... **do**
5: **if** $i > 0$ **then**
6: $\beta_i = \gamma_i/\gamma_{i-1}$;
7: $\alpha_i = \gamma_i/(\delta - \beta_i\gamma_i/\alpha_{i-1})$;
8: **else**
9: $\beta_i = 0; \alpha_i = \gamma_i/\delta$
10: **end if**
11: $z_i = n_i + \beta_i z_{i-1}$
12: $q_i = m_i + \beta_i q_{i-1}$
13: $s_i = w_i + \beta_i s_{i-1}$
14: $p_i = u_i + \beta_i p_{i-1}$
15: $x_{i+1} = x_i + \alpha_i p_i$
16: $r_{i+1} = r_i - \alpha_i s_i$
17: $u_{i+1} = u_i - \alpha_i q_i$
18: $w_{i+1} = w_i - \alpha_i z_i$
19: $\gamma_{i+1} = (r_{i+1}, u_{i+1})$
20: $\delta = (w_{i+1}, u_{i+1})$
21: $norm_{i+1} = \sqrt{(u_{i+1}, u_{i+1})}$
22: $m_{i+1} = M^{-1}w_{i+1}$
23: $n_{i+1} = Am_{i+1}$
24: **end for**

# 4 Methodology

## 4.1 Hybrid-PIPECG-1 Method

In the standard GPU implementation of PCG, the CPU launches CUDA kernels for VMAs, dot products, PC and SPMV on the GPU and then remains idle. In PIPECG, we have independent kernels and thus, we can make use of the idle CPU cores. We show the execution flow of Hybrid-PIPECG-1 in Figure 1(a).

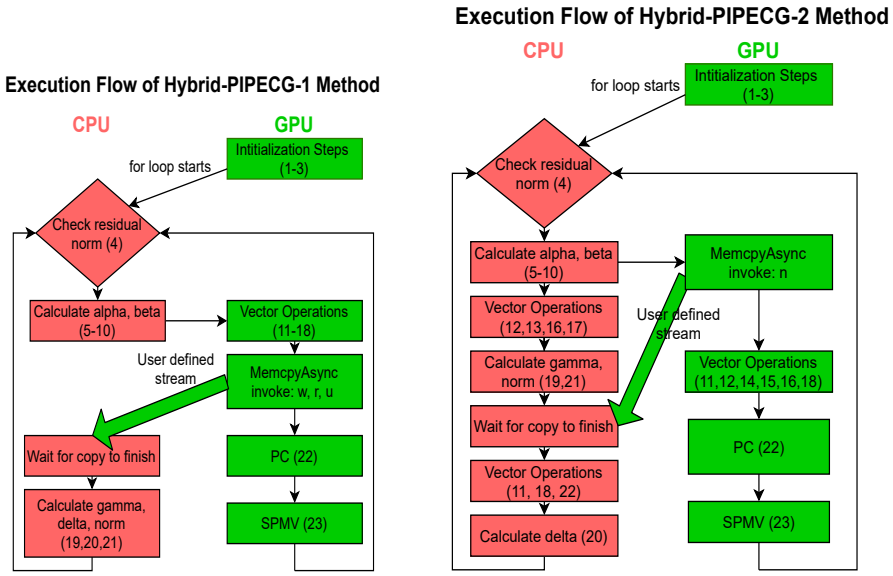
The rectangular boxes show the operation performed and the number within the bracket is the line number of Algorithm 2 that the box executes. The solid thick arrow represents data movement and its direction shows the source and destination of the data movement. The matrix $A$, the vectors $b$ and $x$ have been moved to the GPU prior to this execution flow.

The implementation starts with executing the initialization steps on the GPU. After this, the *for* loop starts which iterates until the preconditioned residual norm becomes smaller than the user defined tolerance. In each iteration, $\alpha$ and $\beta$ are calculated on the CPU. Then the Vector Operations are executed on the GPU which update the vectors $w$, $r$ and $u$ among others. We know that dot products $\gamma$, $\delta$ and $norm$ can be executed simultaneously with PC and SPMV. For executing these dot products on the CPU cores, the CPU needs to have the vectors $w$, $u$ and $r$ but as the updated vectors are on the GPU, we have to copy them to the CPU at every iteration. So here, we define a stream which asynchronously copies $w$, $r$ and $u$ while GPU carries on with its kernel executions. The CPU waits on this stream till the copy is completed and then proceeds to calculate $\gamma$, $\delta$ and $norm$ by using all available cores. Thus, in Hybrid-PIPECG-1, PC and SPMV computations on the GPU are overlapped with the data movement from GPU to CPU and the dot product calculation on the CPU cores.

## 4.2 Hybrid-PIPECG-2 Method

Hybrid-PIPECG-1 requires copy of 3N elements from GPU to CPU in every iteration which can become costly for linear systems with vectors with large N, as the time for copying will exceed the PC + SPMV times thus degrading the overall performance. Therefore, we develop Hybrid-PIPECG-2 shown in Figure 1(b) to reduce the number of vectors to be copied from GPU to CPU in every iteration.

If we want to compute the dot products $\gamma$, $\delta$ and $norm$ on the CPU, we need to have $w$, $u$ and $r$ vectors on the CPU. Instead of copying the updated vectors from the GPU at every iteration, we can update them on the CPU itself. In PIPECG method in algorithm 2, we see that we can update $w$, $u$ and $r$ on the CPU using the vectors $z$, $q$ and $s$. In turn, we would need $n$ and $m$ for updating $z$ and $q$. This means the CPU should have a copy of $z$, $q$, $s$, $n$, $m$, $w$, $u$ and $r$. For updating these vectors on the CPU, we can copy only $n$ from the GPU to CPU. As shown in Figure 1(b), in the for loop, after calculating $\alpha$ and $\beta$, the vector $n$ is copied from the GPU to the CPU on the user defined stream.




(a) Hybrid-PIPECG-1 method (b) Hybrid-PIPECG-2 method

**Fig. 1.** Task Parallel Algorithms

While the copy is progressing, both CPU and GPU perform their operations. GPU proceeds with its Vector Operations, PC and SPMV kernels. On the CPU, we observe that for updating the vectors $z$, $w$ and $m$, CPU needs the vector $n$. While $n$ is being copied, CPU can proceed with the update of vectors $q$, $s$, $r$ and $u$ as they don't need $n$. After vector updates, $\gamma$ and $norm$ can be calculated. Then the CPU waits on the user defined stream until the copy is copied. After $n$ is successfully received, CPU can proceed to update $z$, $w$ and $m$ vectors and compute $\delta$.

Thus, with Hybrid-PIPECG-2, we are able to reduce the number of vector copies to one per iteration. Moreover, the data movement is hidden by computations on the CPU so the CPU doesn't have to be idle while the copy proceeds.

### 4.3 Hybrid-PIPECG-3 Method

Hybrid-PIPECG-1 and Hybrid-PIPECG-2 achieve task parallelism by executing independent kernels on CPU and GPU simultaneously. But for linear systems with even larger N, executing redundant computations for complete vectors of length N on both CPU and GPU proves to be counter-productive. Thus, we propose a data parallel version of PIPECG, Hybrid-PIPECG-3, where the matrix and vectors are decomposed between the CPU and GPU and both these entities carry out PIPECG on their data with communication between each other for important data. This method can also be used for problems that cannot fit in

the GPU memory. Hybrid-PIPECG-3 consists of 3 parts: Performance modelling, Data decomposition, and the actual PIPECG iterations.

**1. Performance Modelling:** We want to calculate the relative performances of the CPU and GPU so that we can decompose data between them according to these relative performances. For this, we execute the SPMV kernel for the complete matrix $A$ (nnz elements) on CPU and GPU separately. We select the SPMV kernel because that is the most time dominating kernel in the PIPECG iteration. If we decompose the data in a way such that the time taken by CPU for SPMV kernel on its data is equal to the time taken by GPU for the SPMV kernel on its data, then complete overlap of the most time consuming kernel is achieved. Hence, we perform five executions of SPMV on both CPU and the GPU for nnz elements. We perform five executions so that effects of cache locality that become prevalent in the later iterations are also be taken into consideration.

Once we have the time taken by CPU cores, $t_{cpu}$ and the time taken by GPU, $t_{gpu}$, we calculate the performance of CPU cores, $s_{cpu}$ and the performance of GPU, $s_{gpu}$ as follows:
$s_{cpu} = nnz/t_{cpu}$
$s_{gpu} = nnz/t_{gpu}$
Then, we calculate the relative performance $r_{cpu}$ and $r_{gpu}$ as follows:
$r_{cpu} = s_{cpu}/(s_{cpu} + s_{gpu})$
$r_{gpu} = s_{gpu}/(s_{cpu} + s_{gpu})$
After we obtain $r_{cpu}$ and $r_{gpu}$, we now divide the $nnz$ into two parts, $nnz_{cpu}$ and $nnz_{gpu}$ as follows:
$nnz_{cpu} = nnz * r_{cpu}$
$nnz_{gpu} = nnz - nnz_{cpu}$

For ease of implementation, we do not assign exact $nnz_{cpu}$ elements to CPU and $nnz_{gpu}$ elements to GPU. Instead, we find out the number of rows to be assigned to the CPU, $N_{cpu}$, which would contain either equal to or slightly less number of non-zeroes than $nnz_{cpu}$. This gives a 1-D decomposition of the $A$ matrix. $N_{gpu}$ is then obtained by $N - N_{cpu}$.

**2. Data Decomposition:** Now that we have $N_{cpu}$ and $N_{gpu}$, we assign $N_{cpu}$ rows to the CPU and $N_{gpu}$ rows to the GPU. We also divide the vectors between the CPU and GPU using same parameters. The division of vectors ensures that there are no redundant computations as both CPU and GPU will be acting on just their local elements. But in every iteration, the SPMV kernels of both CPU and GPU will require the full $m$ vector. After 1-D decomposition, the CPU has $N_{cpu}$ elements of the $m$ vector and the GPU has the other $N_{gpu}$ elements. It is clear that we need to copy these partial vectors from their home device to the other device.

In order to hide the time taken for this copy, we perform a further decomposition of the $nnz_{cpu}$ into $nnz1_{cpu}$ and $nnz2_{cpu}$ in such a way that all the nnz's in $nnz1_{cpu}$ need only the local $N_{cpu}$ elements of $m$ for the SPMV. When SPMV kernel acts on just $nnz1_{cpu}$ elements, we call it SPMV part 1. After the copy of $N_{gpu}$ elements of $m$ is complete, we will then commence SPMV part 2 on $nnz2_{cpu}$ elements which will complete the entire SPMV. We perform the same

for $nnz_{gpu}$. So, through this further local decomposition, we are able to achieve better overlap of computations with communication. In effect, we have achieved the 2-D decomposition of the matrix $A$. This is illustrated in figure 2.

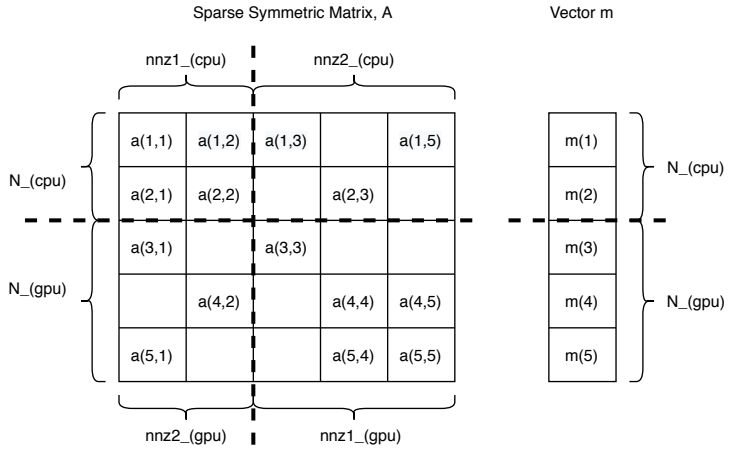



**Fig. 2.** 2-D decomposition of Matrix A

**3. Execution Flow of Hybrid-PIPECG-3 method:** Figure 3 shows the execution flow of the Hybrid-PIPECG-3 method.

For Performance Modelling, we execute the SPMV kernel on CPU and GPU simultaneously. After we get $N_{cpu}$ and $N_{gpu}$, we perform 2-D decomposition of the matrix $A$ and also decompose the vectors. After the decomposition step, the PIPECG method starts. Both CPU and GPU perform the initialization steps on their data except the computation of $n$ vector. Then the for loop starts. After checking the residual norm, CPU calculates $\alpha$ and $\beta$. Then asynchronous copy of $m$ vector is started from CPU to GPU as well as GPU to CPU. These two Copy's are executed simultaneously using two user defined streams, Stream 1 and Stream 2. Similar to Hybrid-PIPECG-2 method, while CPU and GPU wait for $m$ vector to be copied so that they can calculate vector $n$, the vectors that do not depend on $n$ can be updated. This results in vector operations for $q$, $s$, $p$, $x$ and $r$. After these vector updates, $\gamma$ and $norm$ can be computed. To further use the waiting time, CPU and GPU can compute SPMV part 1 as described in section 4.3. Both CPU and GPU then wait for the Copy's to finish. With proper data decomposition, this wait is negligible as the data movement time is completely overlapped with useful computations. Then, CPU and GPU execute SPMV part 2 and obtain the vector $n$. They update the vectors that depend on $n$ and apply PC. Finally, they compute $\delta$ and follow the same steps iteratively.

Thus, with Hybrid-PIPECG-3, we achieve data parallelism by decomposing data between CPU and GPU and the simultaneous operations on CPU and GPU are overlapped with the asynchronous data movement.

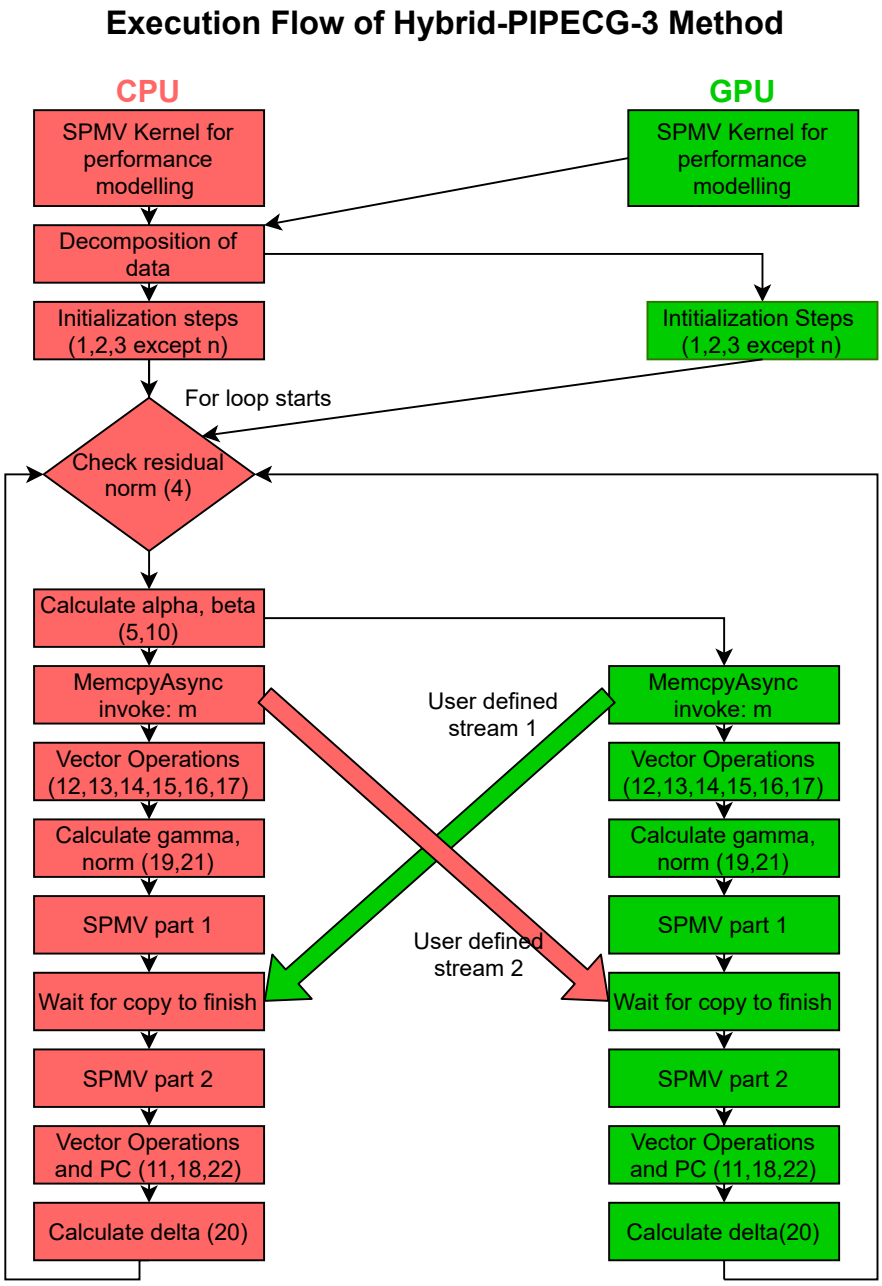



**Fig. 3.** Execution flow of Hybrid-PIPECG-3 method

## 5 Experiments and Results

**Experimental Setup**: We run our tests on two systems: first, a Tesla K40 GPU with 15 Streaming Multiprocessors (SMX), 5GB memory, 16 core Intel CPU and second, a Volta V100 GPU with 80 SMX, 32GB memory and 32 core Intel CPU. We employ OpenMP for using all CPU cores and we employ CUDA kernels, cublas and cusparse libraries for GPU. We run experiments on matrices from the SuiteSparse Matrix Collection[1] as well our own generated Poisson matrices shown in table 1. N is the number of rows and nnz is the number of non-zeroes in the matrix. We solve a linear system of equations $Ax = b$ with the exact solution $x_0 = 1/\sqrt{N}$, where N is the number of rows of A and $b = Ax_0$. We set the absolute tolerance to $10^{-5}$, maximum number of iterations to 10000 and use Jacobi preconditioner. We run all tests to convergence and compare the total execution times of Hybrid-PIPECG-1, Hybrid-PIPECG-2 and Hybrid-PIPECG-3 with the PCG CPU and GPU implementations in the widely used Paralution[8] and PETSc[2] libraries. We also compare our methods with the CPU and GPU implementations of PIPECG method. Here, we note that the total execution time for the Hybrid-PIPECG-3 method always includes the time consumed for performance modelling and data decomposition.

**Table 1.** Matrices used for Experiments

| System | Matrix | N | nnz |
|---|---|---|---|
| K40 | bcsstk15 | 3,948 | 117,816 |
| | gyro | 17,361 | 1,021,159 |
| | boneS01 | 127,224 | 6,715,152 |
| | hood | 220,542 | 10,768,436 |
| | offshore | 259,789 | 4,242,673 |
| | Serena | 1,391,349 | 64,531,701 |
| | Queen_4147 | 4,147,110 | 329,499,284 |
| K40 | 4.5M Poisson | 4,492,125 | 549,353,259 |
| | 5M Poisson | 4,913,000 | 601,211,584 |
| | 6M Poisson | 5,929,741 | 726,572,699 |
| V100 | 17.5M Poisson | 17,576,000 | 2,166,720,184 |
| | 20M Poisson | 19,902,511 | 2,454,911,549 |
| | 25M Poisson | 24,897,088 | 3,073,924,664 |

**Figure 4** compares the performance of our hybrid methods with CPU implementations of PCG in Paralution and PETSc, and with our CPU implementation of PIPECG method on a single node with 16 CPU cores and a K40 GPU. We present the speedups obtained by each method wrt to our PIPECG-OpenMP implementation. We observe that PIPECG-OpenMP performs the worst for every matrix. This is because the PIPECG method introduces extra VMAs to remove the dependencies. This VMA overhead is less pronounced for distributed memory systems but more pronounced for multi-core CPU in a single node. We see that PETSc-PCG-MPI always performs worse than Paralution-PCG-OpenMP.

Finally, we observe that our hybrid methods perform better than all the CPU versions for all matrices because we use GPU cores as well.

For bcsstk15 and gyro, Hybrid-PIPECG-1 performs the best. The same behaviour is observed for matrices with N from 100 to 36000. The other hybrid methods don't perform well for these matrices with small N as Hybrid-PIPECG-2 has redundant computations on the CPU cores and Hybrid-PIPECG-3 has extra overhead of performance modelling and data decomposition. For boneS01, hood and offshore, Hybrid-PIPECG-2 performs the best. The same behaviour is observed for matrices with N from 36000 to 260,000. Hybrid-PIPECG-1 doesn't perform well for larger matrices because copying 3N elements becomes costly for large N. Hybrid-PIPECG-2 copies only N elements. Hybrid-PIPECG-3 performs worse than Hybrid-PIPECG-2 because in Hybrid-PIPECG-2, the vector copy is overlapped by the full SPMV kernel, whereas in Hybrid-PIPECG-3 method, it is overlapped by only SPMV part 1 kernel. For Serena and Queen_4147, Hybrid-PIPECG-3 performs the best. Similar behavior is observed for matrices with N from 260,000 to 4M. Hybrid-PIPECG-1 copies 3N elements in every iteration and hence performs poorly for matrices with very large N. Hybrid-PIPECG-2 copies N elements but performs redundant computations on CPU and GPU which provide great overhead for very large N. So, for very large N (and consequently large nnz), Hybrid-PIPECG-3 provides almost perfect overlap of operations on the CPU and GPU and is the best suited. Thus, we find that different hybrid methods give the best performance for different matrix size ranges.

**Figure 5** compares the performance of our hybrid methods with GPU implementations of PCG in Paralution, PCG in PETSc and PIPECG method in PETSc. We present the speedups obtained by each method wrt to PETSc-PIPECG-GPU implementation. Similar trends as the CPU comparison are observed here as well and we observe that different hybrid methods give the best performance for different matrix size ranges.

Until now, we have presented results on matrices from the SuiteSparse collection that can be fit in K40's memory. Queen_4147 is the largest matrix size that we are able to run on a single K40 GPU. We now analyse Poisson matrices that cannot be fit in K40 and V100 memory (shown in the last 6 rows of table 1). Hybrid-PIPECG-1 and Hybrid-PIPECG-2 launch the SPMV kernel on only the GPU and thus require the complete matrix to be on the GPU. Hence, these methods cannot be used for these cases. We can use Hybrid-PIPECG-3 method because it decomposes data between multi-core CPU and GPU. We compare Hybrid-PIPECG-3 with CPU-only implementations of our PIPECG, PETSC PCG and Paralution PCG methods. We show the performance comparisons on various Poisson Matrices in **Figure 6**. We find that our Hybrid-PIPECG-3 method gives 2-2.5 times speedup over the other methods on K40 and 2.5-6.8 times speedup on V100.

## 6 Conclusion and Future Work

In this work, we proposed three methods for efficient execution of PIPECG method on GPU accelerated systems. Hybrid-PIPECG-1 and Hybrid-PIPECG-

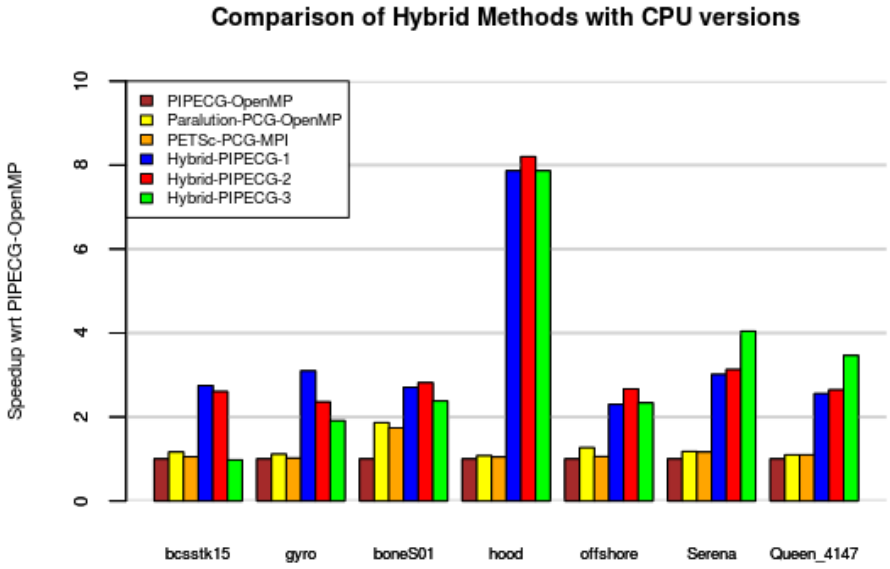



**Fig. 4.** Comparison of Hybrid methods with various CPU versions on a single node with 16 CPU cores and K40 GPU. Speedup presented wrt PIPECG-OpenMP.

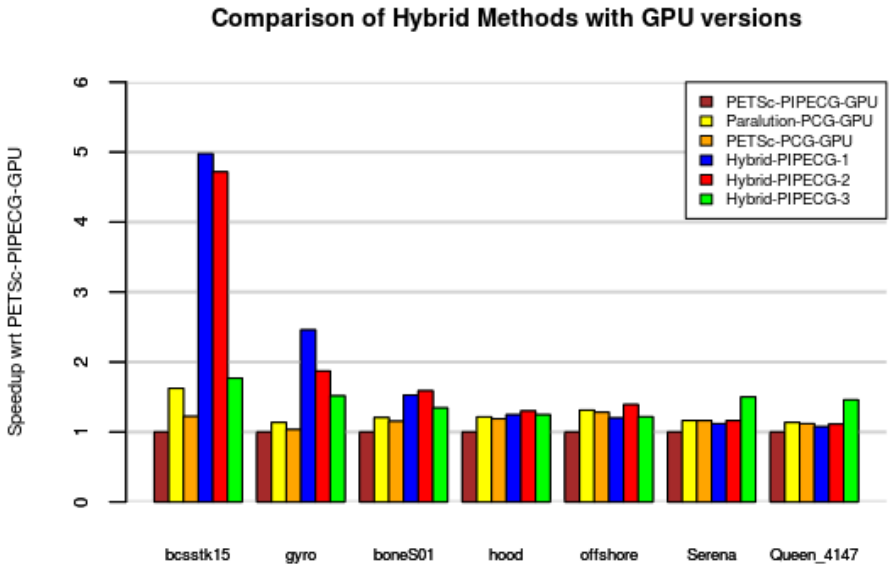
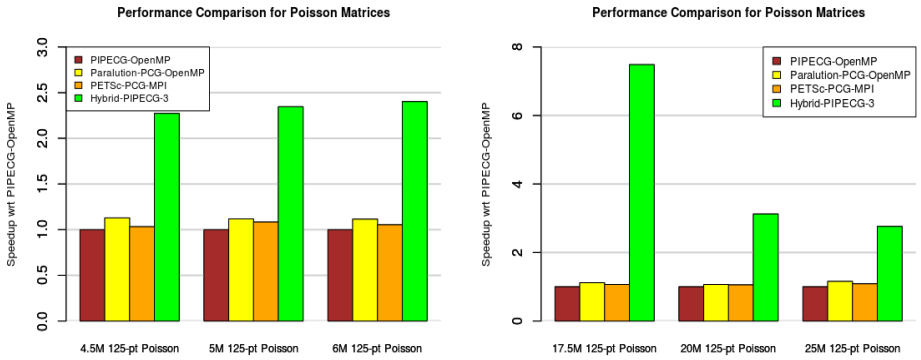



**Fig. 5.** omparison of Hybrid methods with various GPU versions on a single node with 16 CPU cores and K40 GPU. Speedup presented wrt PETSc-PIPECG-GPU.




(a) K40 GPU (b) Volta GPU

**Fig. 6.** Comparison of Hybrid-PIPECG-3 with CPU versions for various Poisson problems on K40 and V100. Speedup presented wrt PIPECG-OpenMP.

2 methods achieve task parallelism by executing dot products on the CPU while GPU executes PC and SPMV kernels. Hybrid-PIPECG-3 method achieves data parallelism by decomposing the workload between multi-core CPU and GPU based on a performance model. Our methods give up to 8x speedup and on average 3x speedup over PCG CPU implementation of Paralution and PETSc libraries. Our methods give up to 5x speedup and on average 1.45x speedup over PCG GPU implementation of Paralution and PETSc libraries. Hybrid-PIPECG-3 method also provides an efficient solution for solving problems that cannot be fit into the GPU memory and gives up to 6.8x speedup for such problems. In the future, we plan to extend this single node single GPU work to multiple nodes with multiple GPUs.

## References

1. Suitesparse matrix collection (2020), https://sparse.tamu.edu/
2. Balay, S., Gropp, W.D., McInnes, L.C., Smith, B.F.: Efficient management of parallelism in object oriented numerical software libraries. In: Modern Software Tools in Scientific Computing. pp. 163–202. Birkhäuser Press (1997)
3. Bell, N., Dalton, S., Olson, L.N.: Exposing fine-grained parallelism in algebraic multigrid methods. SIAM Journal on Scientific Computing **34**(4)
4. Bell, N., Garland, M.: Efficient sparse matrix-vector multiplication on cuda
5. Bell, N., Garland, M.: Implementing sparse matrix-vector multiplication on throughput-oriented processors. In: SC 2009 (2009)
6. Ghysels, P., Vanroose, W.: Hiding global synchronization latency in the preconditioned conjugate gradient algorithm. Parallel Comput. (2014)
7. Hestenes, M.R., Stiefel, E.: Methods of conjugate gradients for solving linear systems. Journal of research of the National Bureau of Standards **49**, 409–436 (1952)
8. Labs, P.: Paralution v1.1.0 (2020), http://www.paralution.com/
9. Li, R., Saad, Y.: Gpu-accelerated preconditioned iterative linear solvers. J. Supercomput. **63**(2) (2013)