PERFORMANCE EVALUATION OF JACOBI ITERATIVE METHOD IN SOLVING DIAGONALLY DOMINANT LINEAR SYSTEM USING OPENACC

IAEME Publication

IAEME PUBLICAITON

International Journal of Advanced Research in Engineering and Technology (IJARET)
Volume 11, Issue 12, December 2020, pp. 1248-1260, Article ID: IJARET_11_12_120
Available online at http://www.iaeme.com/ijaret/issues.asp?JType=IJARET&VType=11&IType=12
ISSN Print: 0976-6480 and ISSN Online: 0976-6499
DOI: 10.34218/IJARET.11.12.2020.120

PERFORMANCE EVALUATION OF JACOBI ITERATIVE METHOD IN SOLVING DIAGONALLY DOMINANT LINEAR SYSTEM USING OPENACC

Muhammad Hafizul Hazmi Wahab
Faculty of Computer Science and Information Technology, University Putra Malaysia,

Nor Asilah Wati Abdul Hamid
Faculty of Computer Science and Information Technology, University Putra Malaysia,
Institute for Mathematical Research, Faculty of Science, University Putra Malaysia

ABSTRACT

Solving linear system with a magnitude of thousand to ten thousand of unknowns takes a very long time in serial fashion. Furthermore, linear system that is discretised from Partial Differentiation Equations (PDE) is also typically solved by a class of iterative method. Therefore, by parallelising Jacobi Iterative Method using OpenACC, we are able to review and compare OpenACC's capabilities in accelerating Jacobi Iterative Method using compiler directives approach as opposed to CUDA's approach. Moreover, we implemented OpenACC in two distinctive domains, where the first domain is on manycore environment with a testbed hardware of Nvidia GeForce GTX 980 and the second domain is on multicore environment where 4 CPU(s) of AMD Opteron 6272 chips are clustered on a single machine with a total of 64 cores. This research project has shown great potentials of the implemented Jacobi Iterative Method using OpenACC as we managed to obtain verily rewarding results. Where, the highest speedup gain is up to 82x faster on GPU with Unified Memory (UM) enabled and 55x times faster on CPU where all 64 cores are fully utilized, and this is when the number of unknowns to be solved is 25,000 and 2,500 respectively.

Keywords: OpenACC, Jacobi, Iterative Method, GPU Programming, Linear System Solver.

Cite this Article: Muhammad Hafizul Hazmi Wahab and Nor Asilah Wati Abdul Hamid, Performance Evaluation of Jacobi Iterative Method in Solving Diagonally Dominant Linear System using OpenACC. *International Journal of Advanced Research in Engineering and Technology,* 11(12), 2020, pp. 1248-1260.
http://www.iaeme.com/IJARET/issues.asp?JType=IJARET&VType=11&IType=12

1. INTRODUCTION

In the realm of numerical linear algebra, Jacobi Iterative Method is used to solve linear system which could be derived from various discretisation processes of partial differentiation equations (PDE); which is used to model natures' physics and many more. In addition, Jacobi Iterative Method is known to its primitiveness in term of its scheme, for example Gauss – Seidel Iterative Method requires the uses of up – to – date result of each variables at each iteration, resulting faster convergence to the approximated solutions of the linear system, this however is not the case for Jacobi Iterative Method. Moreover, linear system of algebraic equations with colossal size commonly exists in atmospheric science, where up to thousands of unknowns are required to be solved simultaneously to find the approximated solutions. In this research project, the Jacobi Iterative Method has been chosen, specifically in solving diagonally dominant linear system as a baseline problem to showcase OpenACC's capabilities in both manycore (GPU) and multicore (CPU) environments.

The introduction of GPU as an accelerator device into the HPC (High Performing Computing) world has induced more and more industry giants to resolve their high performing programs to be ported into an executable kernel on GPU, especially in the field of engineering and sciences. With the growth of GPU adoption into HPC systems in research institutions, not only it helps to accelerate the pace of crucial researches, it also increases the yield of these researches. Needless to say, GPU plays a crucial role in supporting current growth of HPC systems and with that the need of simple yet adoptable API (application programming interfaces) to utilise the GPU's raw power has to be introduced.

Since the introduction of CUDA, GPU programming using CUDA is not an easy learning curve, especially for engineers and scientists with mediocre programming skills [18]. The GPU parallel programming introduced by CUDA has always been using library routines embedded into C/C++ usual code which requires clear crystal understanding of each library routines used in the program. CUDA has becomes the main standard of GPGPU programming for scientific and engineering problems especially in the focus field of deep learning where numerous modern deep learning frameworks has been exploiting the potential of manycore in GPU as well as producing great speedups [7].

This, however, has not stopped OpenACC (Open Accelerator) to make an appearance in the industry with the effort of making GPU programming a little brisker in term of learning curve as well as preserving understanding of the original code. OpenACC approach is different from CUDA, this broad statement can be elaborated further, however, the most prominent difference is that it requires a little less understanding of the underlying hardware, be it tightly coupled or loosely coupled hardware. Moreover, OpenACC is using high - level compiler directives, much like OpenMP. By embedding these compiler directives into serial C/C++ code, OpenACC's compiler recognises the code's portion that must be parallelised accordingly to the various physical target architectures according the command in Subsection 3.4.

This paper was disintegrated into a few sections to facilitate the understanding of our readers, starting from brief introductions to OpenACC, PGI Compiler, & CUDA Unified Memory in Section 2, followed by the implementation of Jacobi's Iterative Method using OpenACC in Section 3, Experiment Result & Discussion in Section 4, and lastly Conclusion & Future Work in Section 5.

2. OPENACC, PGI COMPILER, & CUDA UNIFIED MEMORY

In this Section, we introduce OpenACC, PGI Compiler, and the key features of CUDA Unified Memory. Where the choice of the OpenACC compiler as well as its programming model is briefly introduced. Moreover, the choice compiler is justified in this section to highlight the compiler of choice core features in Subsection 2.1. Lastly, we discussed the key elements of CUDA Unified Memory in Subsection 2.2, where this technology has been implemented in Section 3 and specifically in Subsection 3.3.

2.1. Introduction to OpenACC & PGI Compiler

OpenACC is a programming standard developed by Cray, Nvidia, CAPS, and PGI; the parallel computing standard has been developed with a main goal of simplifying parallel programming on heterogenous underlying hardware of CPU/GPU systems. Much like other parallel programming standards, they are a lot of compilers out there recognise OpenACC's directives by enabling command flags. Keep in mind that OpenACC's Accelerator Model goal is to make certain that OpenACC is not limited to particular target architecture at the moment, but also ready to adapt evolution in the future in regards of target architectures [12]. Furthermore, with OpenACC, an extremely important feature is its code portability, with code portability a single source code can be compiled and executed on various target architectures, it could also be compiled to detect the availability of any compatible accelerator devices before runtime.

In general, this feature alone welcomes scientists and engineers to port their high performing program into using OpenACC's solely or cooperatively with other standards. In addition, in this research project, PGI Compiler has been chosen due to its full support of OpenACC's directives standard. Besides, PGI Compiler also provides some others important feature like compiler output analysis, support implicit approach of parallelism, command line profiler (pgprof), as well as compilation options and flags. Evidently, some OpenACC implementations performed better than CUDA implementation, in a work presented by [14], they managed to achieved a speedup of 85.06x using OpenACC over 83.72x using CUDA; this shows that even with simpler approach of compiler directive – based programming model, OpenACC still manage to outshine a well – matured programming model like CUDA in some cases, thus intrigued our interests in exploring OpenACC over CUDA.

2.2. CUDA Unified Memory Technology

With recent rise of CUDA Unified Memory researchers from various background has reviewed its earlier iteration, which resulted varied performance of CUDA Unified Memory, in some cases it performs well and some it just not. Consequently, CUDA Unified Memory is deemed as limited utility due to its overly high overhead and insignificant improvement in code complexity and performance [8]. Since the date of the work presented, CUDA has release many iterations and improvements to its CUDA Unified Memory algorithm, which not only helps to reduce the overheads but also helps to accelerate some scientific problems by miles. According to [6], selected tests and heat distribution obtained by them shows slightly better result using CUDA Unified Memory which is between 1 – 3 %. With the integration of OpenACC and CUDA Unified Memory (sometimes termed as Managed Memory) it is no doubt that the development of scientific high performing programs can be fast forwarded with lesser performance loss especially with the newer version of CUDA Unified Memory.

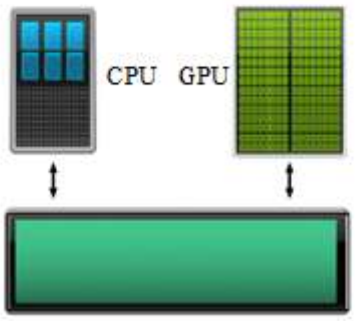


Figure 1 OpenACC/CUDA Unified Memory single global address space, adapted from Brueckner (2018)

With CUDA Unified Memory, programmers' efforts in term of data locality is greatly lifted, refer Figure 1 for an illustration of CUDA Unified Memory. Based on Figure 1, we can see that GPU's and CPU's memory is logically treated as single global address space, which ease the hurdles of address locality management in GPU programming, especially CUDA without Unified Memory. In OpenACC, data management is managed by its data clauses, for example clauses like "*acc data copyin", "copyout", "copy", "create"*, and "*present"*. In this research project, we implemented Jacobi Iterative Method using both Unified and non – Unified Memory using OpenACC as well as using the data clauses with the help of PGI Compiler's analysis.

3. JACOBI'S ITERATIVE METHOD ON OPENACC

Jacobi Iterative Method promises convergence of solutions when the criteria of linear system meet the following characteristic, whereby strictly following criteria (1). Let a linear system be $A_{i,j}$, and i, j are the integers used to traverse through the linear system, A.

$$|A_{ij}| \geq \sum_{j \neq i} |A_{ij}| \#(1)$$

Based on (1), a linear system is strictly diagonally dominant linear system if and only if it follows strictly the condition above. By which applied for all linear equations in the linear system, moreover, a linear system is said to be weakly diagonally dominant if only some of the linear equations in the linear system managed to adjoin criteria (1). In Subsection 3.1, we explained the generation and validation of our datasets to meet the mentioned criteria (1).

Moreover, commonly we denote a linear system with n unknowns as (2) where A represents a linear system vector, y represents unknowns to be solved vector, c is the right - hand side vector of the linear system (RHSV).

$$Ay = c\#(2)$$

Based from (2) we can see that the linear system is organized into its matrix form, and with that A could be further disintegrated into $A = D + R$, where D is the diagonal element of A, and R is the remainder elements of A. The solutions y can be approximated iteratively using (3.1 – 3.2). In general, we used Jacobi Iterative Method general equations (3.1 – 3.2) to solve (2).

$$y^{(k+1)} = D^{-1}\left(c - Ry^{(k)}\right)\#(3.1)$$

$$y_i^{(k+1)} = \frac{1}{a_{ii}}\left(c_i - \sum_{j \neq i} a_{ij} y_j^{(k)}\right) \#(3.2)$$

Observing (3.1 – 3.2) adapted from [17], we can see that (3.2) is an expanded form of (3.1), from the equations we can see that there are three control variables which are $i, j, \& \, k$. The former two are used to traverse the 2-dimensional array of the linear system, A. Meanwhile, the latter k is used to control the iteration convergence as well as representing the current iteration number of the iterative method. Jacobi's Iterative Method used the previous value of $y_j^{(k)}$ to approximate the next value of the approximation, $y_j^{(k+1)}$. In short, to find a solution of a linear system at iteration $y_j^{(k+1)}$, the inverse of the diagonal element needed to be timed with the product of C_i minus the summation of non – diagonal element of linear system times with previous result of the iteration $y_j^{(k)}$.

3.1. Dataset Generation & Validation

Based from the criteria (1), we generate a few bounded random data sets and validated them before performing Jacobi's Iterative Method, this is to avoid prolonged convergence of the iterative method, as well as controlling the tests in a smaller scale in term of its execution times. Next, the sizes of the dataset we generated is adapted from [11], where in his tests the matrix used was between 2,000 – 20,000. However, in this research project, we scaled the matrix size from 1500, 2500, 5000, 10000, 15000, 20000, 25000, and 30000. The smaller matrix sizes are used to determine whether OpenACC implementation is worthwhile or not on manycore environment (GPU). Whilst the bigger size linear systems are used to test the performance loss or gain as we increase the size of the matrices (linear system). The simulation of the Jacobi Iterative Method implemented in this research project is done by first validating the dataset uses at the beginning, to make certain that the dataset meets criteria (1), consequently allowing convergence of the Jacobi Iterative Method.

3.2. Convergence Criterion

Convergence criterion is an important aspect of any iterative method, without a proper convergence criterion, a set of approximated solutions may not even come close to the real solutions and might deviated further from the real solutions. In this research project, a control variable has been deployed, Δy, as a sentry on convergence behaviour. The implementation of the sentry can be simplified as (4), where $max\{x, y\} \Rightarrow x \vee y$ and I is the number of unknowns to be solved.

$$\Delta y \leftarrow max\{|y_i^k - y_i^{k-1}|, |y_{i+1}^k - y_{i+1}^{k-1}|\} \leq \mathcal{E} \, \forall i \in \{1, \dots, I\} \# \quad (4)$$

Based on (4), we have collimated that the approximated set of solutions of this research project is with the relative accuracy of $\mathcal{E} = 1.0 \times 10^{-16}$ from the real solutions. Furthermore, once the biggest value in the solutions' vectors reached this accuracy at any point of the iteration process, the iteration is then halted, and Jacobi Iterative Method is now converged to a set of approximated set of solutions with the accuracy of $\mathcal{E}$. Although commonly converging at larger number of iterations compared to Gauss – Seidel Iterative Method, Jacobi's scheme allows better parallelism, thus consequently it iterated faster on GPU manycore execution, in another words, better converging point does not necessarily mean faster iteration rate. It is also worth noting here that in work presented, they evaluated both single precision floating point and double precision floating point and the latter proved to be producing lower speedup gain compared to single precision floating point [18]. Factors like memory address length and the vector's portion that resides in GPU's memory could be the main factors that caused such result; however, this research has not scoped to such discussion.

3.3. OpenACC's Implementation

From Figure 2 below we can observe that there are a few OpenACC's directives has been enacted in the base algorithm pseudocode, the first one is OpenACC data directive, whereby it marks the beginning and the end of the data region. Note that, data region used in this research project is structured - data approach by OpenACC standard due to its suitability with the code's logics as well as the code's structure; there is also an unstructured data approach provided by OpenACC's APIs, which is a more explicit approach of data management in OpenACC. Once, the logical data region has been deployed, vectors that are needed for computation are fetched and GPU memory is allocated before compute region is entered. Furthermore, the involved vectors will be maintained in GPU's memory until the computation is finished; consequently, reducing unnecessary data movements between host (CPU) and device (GPU) throughout the computation.

Algorithm 1: OpenACC Implementation of Jacobi's Iterative Method in Solving Strictly Diagonally Dominant Linear System

```
$I \leftarrow NumberOfUnknown$
$\varepsilon \leftarrow 1.0 \times 10^{-16}$
Begin OpenACC Data Region
while $k \leftarrow 1 \leq Iteration_{max}$ or $\Delta y \geq \varepsilon$ do
    $\Delta y \leftarrow 0.0$
    Begin OpenACC Parallel Compute Region
    Begin OpenACC Gang - Level Parallelism
    for $i \leftarrow 1, I$ do
        $\sigma \leftarrow 0.0$
        Begin OpenACC Vector - Level Parallelism
        for $j \leftarrow 1, I$ do
            OpenACC $\sigma$ Plus Reduction Clause
            if $j \neg i \equiv true$ then
                $\sigma \leftarrow \sigma + A_{ij} \cdot y_j^{k-1}$
            end
        end
        $y_i^k \leftarrow \frac{C_i - \sigma}{A_{ii}}$
        OpenACC $\Delta y$ Max Reduction Clause
        $\Delta y \leftarrow max\{ \mid y_i^k - y_i^{k-1} \mid , \mid y_{i+1}^k - y_{i+1}^{k-1} \mid \}$
        if $i \mod j \equiv 10$ then
            OpenACC Update Host
        end
    end
    End OpenACC Compute Region
end
End OpenACC Data Region
```

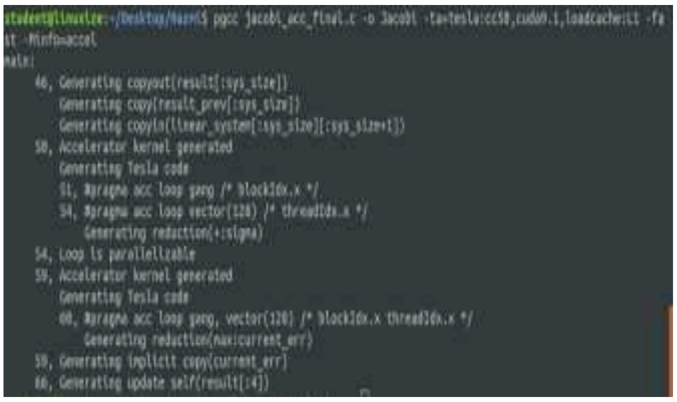
Figure 2 Jacobi's Iterative Method OpenACC Implementation derived from Equation (3.1 – 3.2) in Section 3

Although OpenACC allows implicit data region, where compiler will be deciding where and when data should be available on device before it is needed; this approach is however not recommended for complex problems, especially when there are multiple nested loops and complex logic operations which could affect the compiler's data dependencies analysis at a greater magnitude. For instance, a complex loop might be indisputably parallelisable in the eye of programmer, but not to the compiler's data dependencies analyser, where the compiler further inhibits code's ability to be parallelised as means to avoid possible logical errors or race condition; though this can be overwritten by programmer using loop construct rather than kernel construct.

3.4. OpenACC Program Compilation using PGI Compiler

PGI Compiler provides set of compilation tools as extension that can be used to ease the process of improving code's performance by the means of compiler analysis output as well as light – weight code profiling, by which able to profile both GPU and CPU activities. The following are the commands and the flags used to compile a single identical OpenACC's code to be executed on different target architectures.

In this research project we used the first command (i) in Figure 3 to compile the program to be executed as a serial program on the host. Then, we used the second command (ii) to compile Jacobi Iterative Method to be executed as multicore program on multiple CPU(s) chips; an additional flag is included in the command to yield a compiler analysis output on the parallelisation process, we are able to determine which line of the code is parallel inhibiting and which line is not with the analysis. This is an important aspect of parallel programming, where sometimes parallel inhibitor is not quite apparent in the code, thus making the program executing much slower; therefore, a *report* from the compilation process is crucial to help with the debugging process as well as confirming the code's implementation, refer Figure 4.



```
student@linuxlze:~/Desktop/Hazmi$ pgcc jacobi_acc_final.c -o Jacobi -ta=tesla:cc50,cuda9.1,loadcache:L1 -fa
st -Minfo=accel
main:
     46, Generating copyout(result[:sys_size])
         Generating copy(result_prev[:sys_size])
         Generating copyin(linear_system[:sys_size][:sys_size+1])
     50, Accelerator kernel generated
         Generating Tesla code
         51, #pragma acc loop gang /* blockIdx.x */
         54, #pragma acc loop vector(128) /* threadIdx.x */
             Generating reduction(+:sigma)
     54, Loop is parallelizable
     59, Accelerator kernel generated
         Generating Tesla code
         60, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
             Generating reduction(max:current_err)
     59, Generating implicit copy(current_err)
     66, Generating update self(result[:4])
```

Figure 3 PGI Compiler's output compiling program to be executed on Nvidia GTX980

In addition, the third command (iii) was used to compile the program to be executed on the device (GPU), where the target architecture is Nvidia GTX980, with compute capability of 5.0 (tesla:cc50). The additional flags *"-fast"* is used to instruct the compiler to use CUDA fast math library instead of standard C math library. Lastly, the last command (iv) is used to compile the program to make use CUDA Unified Memory technology; this is done by attaching *"managed"* option onto the target architecture flag during the compilation. When OpenACC program is compiled with *"managed"*, OpenACC's data directives will be partially or completely ignored or used as means to prefetch data.

```
i    $pgcc jacobi_acc.c -o Jacobi –ta=host
ii   $pgcc jacobi_acc.c -o Jacobi –ta=multicore -
     Minfo=accel
iii  $pgcc jacobi_acc.c -o Jacobi –ta=tesla:cc50,
     cuda9.1, loadcache:L1 -fast -Minfo=accel
iv   $pgcc jacobi_acc.c -o Jacobi –ta=tesla:cc50,
     cuda9.1, loadcache:L1, managed -fast -
     Minfo=accel
```

Figure 4 Compilation commands to target various architecture

In this architecture (Unified Memory), some of system memory is shared among the host and accelerator threads of the device; whilst each device may have their own local memories, it accesses a portion of memory that's shared; or to simplify it enables the possibility of using a single logical address space between host and device memories [2]. Since OpenACC relies on CUDA libraries to enable such technology, the memory management is fully handled by CUDA Managed Memory. Therefore, prefetching, transfer, and pages mapping algorithm(s) depend on the version of CUDA used during the compilation, some version might perform better in some cases and some older version performs at mediocrity like the one [8] presented in their work.

3.5. Test Platforms

In this research project, the implemented Jacobi Iterative Method program is carried out on two distinctives machines, representing CPU – GPU Manycore and CPU Multicore domain, respectively. The first machine houses Nvidia GeForce GTX980 with 4GB of GDDR5 memory, 2048 CUDA Cores, Compute Capability 5.0, and Intel Xeon® CPU E5 – 1620 v2 @ 3.70GHz (8 Logical Cores); the operating system that is running on this hardware is Ubuntu 18.04.01 x64. Moreover, the second machine houses no GPU, but consists of 4 AMD Opteron 6272 chips (16 cores per chip). In total, the second machine consists of 64 CPU Cores, with base frequency of 2.4GHz and running CentOS 6.9 x64. Both machines sport a total of 16GB DDR3 RAM each as well as using the same compiler version of PGI Community Edition Compiler (18.04). With this version of compiler, OpenACCv2.6 standard is fully supported along with newer version of CUDA 9.1. Refer Table 1 for simplification.

Table 1 Test platforms used to execute Jacobi Iterative Method program

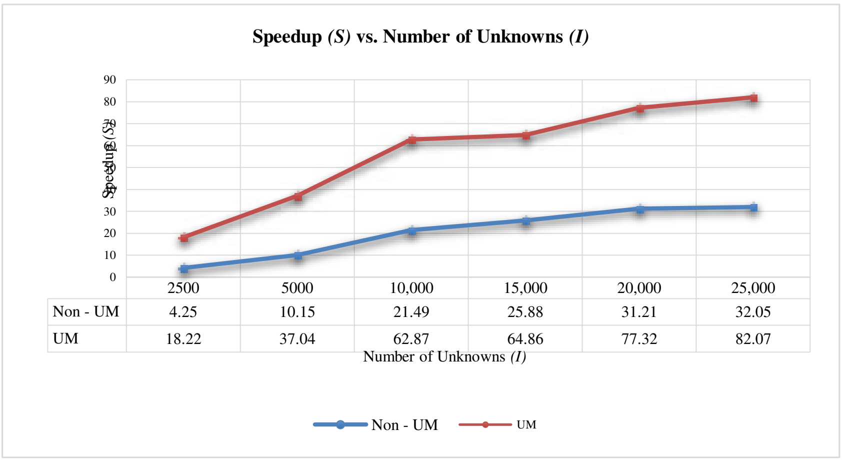
	Manycore Execution (GPU)	**Multicore Execution (CPU)**
CPU Clock Speed	Intel Xeon® CPU E5 – 1620 v2	4x AMD Opteron 6272
GPU Clock Speed	3.70GHz	2.4GHz
Memory (RAM)	16GB DDR3	16GB DDR3
CPU Cores No.	4 Physical	64 Physical
GPU CUDA Cores No.	2048	-
OS	Ubuntu 18.04.01	Ubuntu 18.04.01
OpenACC (v)	2.6	2.6
OpenACC Compiler	PGI Community Compiler	PGI Community Compiler

4. RESULT & DISCUSSION

In this section, we organised our research outputs by sectioning it into two distinctive domains, which are manycore (GPU) in Subsection 4.1 and multicore environments (4 CPU Chips) in Subsection 4.2. Lastly, in Subsection 4.3, the results of prior subsections are compiled and contrast, and finally showcased to elucidate our findings in total clarity.

4.1. Manycore Environment Result (CPU – GPU Discrete Memories & Unified Memory)

In this section, we ran the serial program on Intel Xeon® CPU E5 – 1620v2 as means to baseline the speedups tabled and plotted in Figure 5. The relative speedup is denoted as S and the number of unknowns is denoted as I, ranging from 2,500 – 25,000. Based on Figure 5, we can observe that the speedup gains using GTX980 with Unified Memory enabled is at least doubled the speedup gains without Unified Memory enabled. The speedup plot obtained is reasonably high, with the highest speedup gained is up to 82.07x when they are 25,000 unknowns to be solved in the diagonally dominant linear system.



	2500	5000	10,000	15,000	20,000	25,000
Non - UM	4.25	10.15	21.49	25.88	31.21	32.05
UM	18.22	37.04	62.87	64.86	77.32	82.07

Figure 5 Nvidia GTX980 Speedup

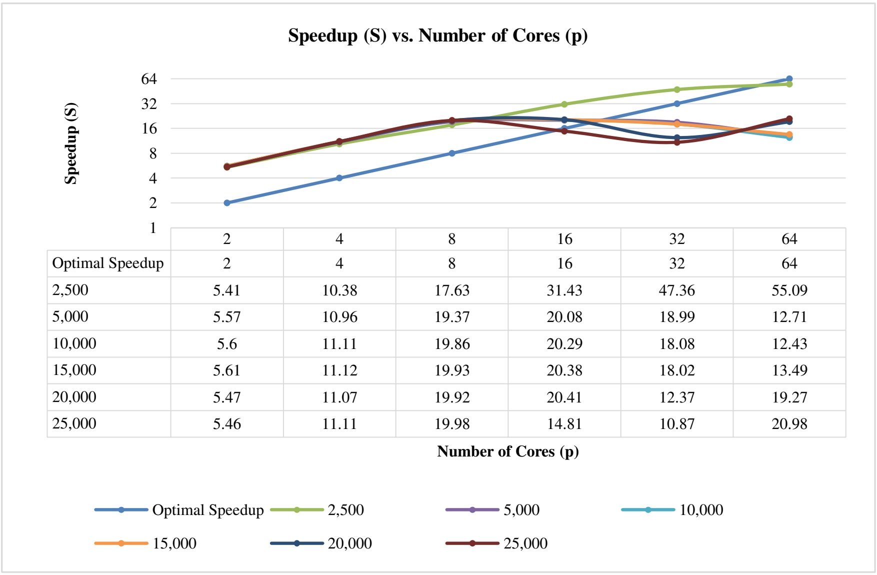
Furthermore, we can see that with Unified Memory (UM), a consistently higher speedup is obtained compared to plain Host – Device execution without Unified Memory (Non – UM). It is also worth noting here that when the differences between the speedup gain between these two lines are more significant when the number of unknowns to be solved are 10,000 – 25,000, this indicates Unified Memory execution managed data prefetching and locality better when larger dataset size is fed, particularly in the case of Jacobi Iterative Method execution. In this project, total freedom for the program runtime in managing the involved vectors to achieve such results has been given.

In CUDA, calls like *"cudaMemAdvise"* and *"cudaMemPrefetchAsync"* can be used to hint the compiler on where data vectors should be prefetched before it is needed. In addition, in OpenACC, compiler directives like *"cache"* can be used to cache certain variable that is accessed frequently to reduce variable migration between the memories. Ultimately, the Jacobi Iterative Method implemented on OpenACC has proven to show a significantly better speedup gains when Unified Memory is activated.

4.2. Manycore Environment Result (CPU – GPU Discrete Memories & Unified Memory)

The code is again compiled to run specifically on AMD Opteron 6272, which is housed by another machine, refer Table 1. The execution time is sampled and used as baseline to calculate the speedup we plotted in Figure 6. The symbols and the range of data set sampled is the same as prior section and the number of cores used is manipulated by *exporting* OpenACC's runtime environment variables *"ACC_NUM_CORES=p"* accordingly before execution.

Based from Figure 6, we can discern that almost all number of unknowns managed to be solved with super – linearity by using just a single chip of AMD Opteron 6272, $p = 16$. Except when $I = 2500$, where super – linear speedup is achieved even when 32 cores are used (2 chips of AMD Opteron 6272). However, this is not the case when it comes to larger problem, $I = 25000$, where super – linear speedup can only be achieved when we use only up to 8 cores. This proves that when the problem size grows, the intra – chip communication becomes very well – saturated, especially when the architecture of AMD Opteron 6272 (bulldozer architecture) shares the same floating – point unit between 2 of 16 cores on the same chip; therefore, the access frequency to the unit increases significantly, consequently yielding less favourable result.



	2	4	8	16	32	64
Optimal Speedup	2	4	8	16	32	64
2,500	5.41	10.38	17.63	31.43	47.36	55.09
5,000	5.57	10.96	19.37	20.08	18.99	12.71
10,000	5.6	11.11	19.86	20.29	18.08	12.43
15,000	5.61	11.12	19.93	20.38	18.02	13.49
20,000	5.47	11.07	19.92	20.41	12.37	19.27
25,000	5.46	11.11	19.98	14.81	10.87	20.98

Figure 6 AMD Opteron 6272 Speedup

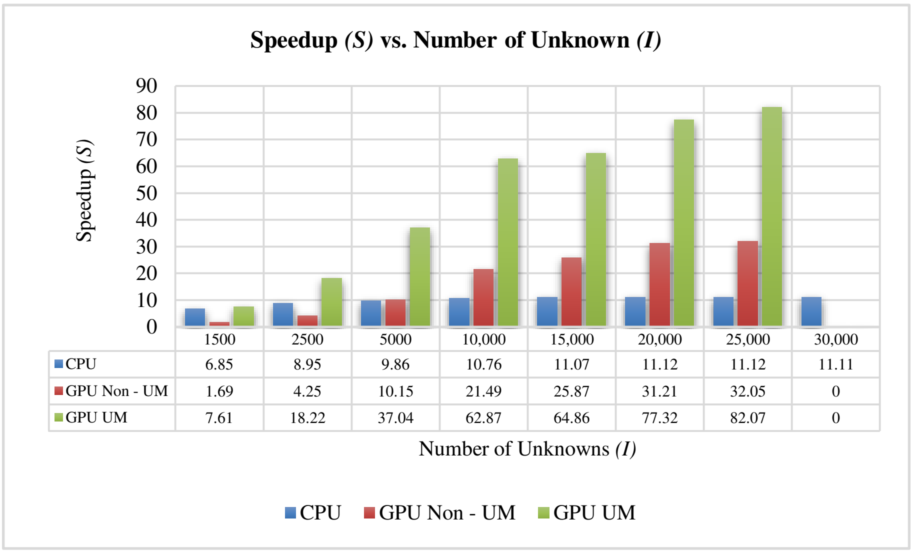
For smaller size of unknowns, a larger percentage of the involved vector's elements managed to reside on the chip's caches during the computation, thus yielding super – linear results. Moreover, when it comes to the usage of more processing unit, $p = 32$, it seems like almost all sizes of unknowns failed to maintain a scalable speedup, except when $I = 2500$. The coordination of using up to 3 chips seems to be hard task for AMD Opteron 6272 and is proven to be unworthy of the computing resources supplied. Lastly, when it comes to full utilisation of 4 chips, $p = 64$, the speedup gains that is scalable with the resources provided is only when $I = 2500, 20000, 25000$. Nevertheless, OpenACC has proven to be able to yield a reasonable speedup on multicore environment, specifically in solving system of linear algebraic equations using Jacobi Iterative Method, thus fortifying its position as one of the parallel programming models that a parallel programmer should consider when it comes to multicore environment.

4.3. OpenACC on Discrete Memory (Non – UM & UM) & Shared Memory

The higher transistors count that has been devoted for ALU (Arithmetic Logic Unit) in GPU has always been one of the contributing factors in yielding very high speedup gains for compute intensive problems. However, the costs of data movements between Host – Device do not come without sacrificing performance. For instance, a highly granular program that requires a lot of data movement between host and device will surely perform better solely without the uses of GPU (on shared memory). Thus, one of the practices in GPU parallel programming is to pipeline involved vectors and make use of the large GPU's memory bandwidth to overlap communication and computation or restructure the code to be as coarsely granular as possible. Therefore, to investigate the matter further a test on the machine in Subsection 4.1 has been executed again to compare and contrast the viability of using GPU (Non – UM & UM) over Multicore CPU execution on the machine, refer Figure 7.

Based solely from Figure 7, we can see that for smaller size of unknowns,$I = 1500, 2500$, full utilisation of CPU outperforms GPU (Non – UM), this is caused by the high costs of transferring involved vectors physically to device's memory before the computation begins. However, this is not the case when it comes to GPU (UM), where it managed to perform very well even when the size of problem is very small, this is due to the fact that with CUDA Managed Memory, the vectors that's not worth the physical migrations from host's memory to device's memory are recognised and thus CUDA Managed Memory employs the uses of the same address space; indirectly eliminating the cost of the physical data migration. Besides to further utilise the massively parallel nature of the GPU, an algorithm needs to be highly arithmetic intense, where, the ratio of compute operations needs to be higher than the communication (memory accesses) to benefit the available computational resources on the GPU [3]. Therefore, we can safely conclude that here for smaller sizes of unknowns, CPU Shared Memory outperforms Non – Unified Memory approach of OpenACC. It is also worth noting here that CPU managed to achieve super linearity when the unknown size increase; [2] mentioned that OpenACC's level of parallelism is up to vector level, where in CPU it is mapped to SIMD (Single Instruction Multiple Data) unit, with OpenACC, the compiled program can better prefetch and pipelined the required operands up to vector level, and thus yielding super linear results.

Moreover, by comparing Figure 6 & Figure 7, we can see that single manycore GPU outperforms the usage of clustered CPUs, where 4 CPU chips with total of 64 physical cores still could not come near GPU speedup gains. Therefore from the perspective of performance gains over the cost of compute cores added, we could say that GPU implementation is more cost – efficient, where, the cost of deploying single GPU over the cost of clustering 4 CPU chips cost less in the case of this research project. Though, this could be very subjective due to few factors, such as, the cost of the GPU used, the cost of the CPU used, and the cost of maintaining the whole system. It is also worth mentioning here that power usage of both systems is different, and thus to properly evaluated cost efficiency ratio, this study has to be extended with additionally set of performance matrices to be evaluated. As conclusion, OpenACC has proven to be a tough contender to OpenMP and CUDA in both multicore and manycore departments respectively where it evidently managed to deliver high speedup gains with high level of code portability simultaneously.



	1500	2500	5000	10,000	15,000	20,000	25,000	30,000
CPU	6.85	8.95	9.86	10.76	11.07	11.12	11.12	11.11
GPU Non - UM	1.69	4.25	10.15	21.49	25.87	31.21	32.05	0
GPU UM	7.61	18.22	37.04	62.87	64.86	77.32	82.07	0

Figure 7 CPU & GPU (Non – UM & UM) Speedup

5. CONCLUSION & FUTURE WORK

In the nutshell, OpenACC's performance on both multicore and manycore environments has been presented and evaluated, the results elucidated in Section 4 has proven and showcased OpenACC's capabilities in parallelising program with its universally unprejudiced accelerator model. Additionally, the compiler directives approach of OpenACC has proven to be up to par with CUDA implementation of Jacobi Iterative Method in works presented by [11] as well as [18]. Whereby, when the size of matrix to be solved is 20000, [11] managed to solve linear system using Jacobi Iterative Method with the speedup of x72.8. With the introduction of CUDA Unified Memory into OpenACC (termed as OpenACC Managed Memory), we believe that researchers from various fields could benefit the massive reduction of laborious works in the process of parallelising their compute intensive programs, especially when it comes to harnessing the power of GPU's manycore nature. Furthermore, OpenACC also supports the implementation of multi – GPUs besides its inordinate code's portability feature, which could haste and catalyse the development process without sacrificing exorbitant code's performance. Work presented by [14] has explored the possibility of decomposing Koch – Baker – Alcouffe (KBA) parallel – Wavefront sweep miniapplication using MPI with OpenACC. The implementation allows the work to run on numerous nodes and accelerators. Besides that, [4] has also implemented multi GPUs approach of simulating lattice QCD simulations using OpenACC and MPI, the results achieved shows promising speedup gain in implementing multi GPUs. Besides, OpenACC cache directive was not explored in this work, where cache directive could efficiently move certain vectors that were frequently accessed to device's memory, consequently allowing higher speedup yield, like the work presented by [9]. Where they achieved 20.8x speedup for Jacobi preconditioned conjugate gradient finite – element solver. Therefore, we believe that the basal Jacobi Iterative Method & OpenACC can further be explored from the perspective of multi GPUs approach, where more than single standard is used simultaneously to achieve even greater performance, much like the mentioned works that used OpenACC along with MPI to properly utilise distributed multi GPUs architecture.

ACKNOWLEDGEMENT

This research is supported by the University Putra Malaysia under Research Grant Scheme (GP/2017/9569600).

REFERENCES

[1] Brueckner, R. (2018). Accelerating HPC Applications on NVIDIA GPUs with OpenACC. Retrieved from https://insidehpc.com/2018/03/accelerating-hpc-applications-nvidia-gpus-openacc/ on 20 February 2019.

[2] Chandrasekaran, S., & Juckeland, G. (2017). OpenACC for programmers: Concepts and strategies. Boston: Addison-Wesley.

[3] Eichstädt, J., Vymazal, M., Moxey, D., & Peiró, J. (2020). A comparison of the shared-memory parallel programming models OpenMP, OpenACC and Kokkos in the context of implicit solvers for high-order FEM. Computer Physics Communications, 107245.

[4] Gupta, S., & Majumdar, P. (2018). Accelerating lattice QCD simulations with 2 flavors of staggered fermions on multiple GPUs using OpenACC-A first attempt. Computer Physics Communications, 228, 44-53.

[5] Hamid, N. A. W. A., & Coddington, P. (2010). Comparison of MPI benchmark programs on shared memory and distributed memory machines (point-to-point communication). *The International Journal of High Performance Computing Applications*, *24*(4), 469-483.

[6] Jarząbek, Ł., & Czarnul, P. (2017). Performance evaluation of unified memory and dynamic parallelism for selected parallel CUDA applications. Journal of Supercomputing, 73(12), 5378–5401. https://doi.org/10.1007/s11227-017-2091-x.

[7] Kim, H., Nam, H., Jung, W., & Lee, J. (2017). Performance analysis of CNN frameworks for GPUs. ISPASS 2017 - IEEE International Symposium on Performance Analysis of Systems and Software, 55–64. https://doi.org/10.1109/ISPASS.2017.7975270.

[8] Landaverde, R., Zhang, T., Coskun, A. K., & Herbordt, M. (2014). An investigation of Unified Memory Access performance in CUDA. 2014 IEEE High Performance Extreme Computing Conference, HPEC 2014. https://doi.org/10.1109/HPEC.2014.7040988.

[9] Lashgar, A., & Baniasadi, A. (2019). Efficient implementation of OpenACC cache directive on NVIDIA GPUs. International Journal of High Performance Computing and Networking, 13(1), 35-53.

[10] NVIDIA. (2020). CUDA Technology; http://www.nvidia.com/CUDA.

[11] Oancea, B., Andrei, T., & Dragoescu, R. M. (2012). Improving the performance of the linear systems solvers using cuda. Challenges for the Knowledge Society, 2036–2045.

[12] OpenACC Organization. (2015). OpenACC Programming and Best Practices Guide, (June).

[13] OpenMP, http://www.openmp.org

[14] Searles, R., Chandrasekaran, S., Joubert, W., & Hernandez, O. (2019). MPI+ OpenACC: Accelerating radiation transport mini-application, minisweep, on heterogeneous systems. Computer Physics Communications, 236, 176-187.

[15] Snyder, L. (1988). A taxonomy of synchronous parallel machines. WASHINGTON UNIV SEATTLE DEPT OF COMPUTER SCIENCE.

[16] The OpenACC standard, http://www.openacc-standard.org, Accessed on 1/8/2020.

[17] Tian, Z., Tian, M., Liu, Z., & Xu, T. (2017). The Jacobi and Gauss–Seidel-type iteration methods for the matrix equation AXB = C. *Applied Mathematics and Computation*, *292*, 63-75.

[18] Zhang, Z., Miao, Q., & Wang, Y. (2009). CUDA-based Jacobi's iterative method. IFCSTA 2009 Proceedings - 2009 International Forum on Computer Science-Technology and Applications, 1, 259–262. https://doi.org/10.1109/IFCSTA.2009.68.