UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Tecnura

http://revistas.udistrital.edu.co/ojs/index.php/Tecnura/issue/view/687
DOI: http://doi.org/10.14483/udistrital.jour.tecnura.2014.DSE1.a14

REFLEXIÓN

Parallel programming languages on heterogeneous architectures using openmpc, ompss, openacc and openmp

Lenguajes para programación paralela en arquitecturas heterogéneas utilizando openmpc, ompss, openacc y openmp

Esteban Hernández B.*, Gerardo de Jesús Montoya Gaviria**, Carlos Enrique Montenegro***

Fecha de recepción: June 10tyh, 2014 **Fecha de aceptación:** November 4t, 2014

Citation / Para citar este artículo: Hernández, E., Gaviria, G. de J. M., & Montenegro, C. E. (2014). Parallel programming languages on heterogeneous architectures using OPENMPC, OMPSS, OPENACC and OPENMP. *Revista Tecnura, 18* (Edición especial doctorado), 160–170. doi: 10.14483/udistrital.jour.tecnura.2014.DSE1.a14

ABSTRACT

On the field of parallel programing has emerged a new big player in the last 10 years. The GPU's have taken a relevant importance on scientific computing because they offer a high performance computing, low cost and simplicity of implementation. However, one of the most important challenges is the program languages used for this devices. The effort for recoding algorithms designed for CPUs is a critical problem. In this paper we review three of principal frameworks for programming CUDA devices compared with the new directives introduced on the OpenMP 4 standard resolving the Jacobi iterative method.

Keywords: *CUDA*, Jacobbi method, OmpSS, OpenACC, OpenMP, OpenMP, Parallel Programming.

RESUMEN

En el campo de la programación paralela, ha arribado un nuevo gran jugador en los últimos 10 años. Las GPU han tomado una importancia relevante en la computación científica debido a que ofrecen alto rendimiento computacional, bajo costos y simplicidad de implementación; sin embargo, uno de los desafíos más grandes que poseen son los lenguajes utilizados para la programación de los dispositivos. El esfuerzo de reescribir algoritmos diseñados originalmente para CPU es uno de los mayores problemas. En este artículo se revisan tres frameworks de programación para la tecnología CUDA y se realiza una comparación con el reciente estándar OpenMP versión 4, resolviendo el método iterativo de Jacobi.

Palabras clave: Método de Jacobi, OmpSS, OpenACC, OpenMP, Programación paralela.

* Network Engineering with Master degree on software engineering and Free Software construction, minor degree on applied mathematics and network software construction. Now running Doctorate studies on Engineering at Universidad Distrital and works as principal architect on RUNT. Now works in focus on Parallel Programming, high performance computing, computational numerical simulation and numerical weather forecast. E-mail: ejhernandezb@udistrital.edu.co

** Engineer in meteorology from the University of Leningrad, with a doctorate in physical-mathematical sciences of State Moscow University, pioneer in the area of meteorology in Colombia, in charge of meteorology graduate at Universidad Nacional de Colombia, researcher and director of more than 12 graduate theses in meteorology dynamic area and numerical forecast, air quality, efficient use of climate models and weather. He is currently a full professor of the Faculty of Geosciences at Universidad Nacional de Colombia. E-mail: gdmontoyag@unal.edu.co

*** System Engineering, PhD and Master degree on Informatics, director of research group GIIRA with focus on Social Network Analyzing, eLearning and data visualization. He is currently associate professor of Engineering Faculty at Universidad Distrital. E-mail: cemontenegrom@udistrital.edu.co

INTRODUCTION

Since 10 years ago, the massively parallel processors have used the GPUs as principal element on the new approach in parallel programming; it's evolved from a graphics-specific accelerator to a general-purpose computing device and at this time is considered to be in the era of GPUs. (Nickolls & Dally, 2010). However, the main obstacle for large adoption on the programmer community has been the lack of standards that allow programming on unified form different existing hardware solutions (Nickolls & Dally, 2010). The most important player on GPU solutions is Nvidia ® with the CUDA® language programming and his own compiler (nvcc) (Hill & Marty, 2008), with thousands of installed solutions and reward on top500 supercomputer list, while the portability is the main problem. Some community project and some hardware alliance have proposed solutions for resolve this issue. OmpSS, OpenACC and OpenMPC have emerged as the most promising solutions (Vetter, 2012) using the OpenMP base model. In the last year, OpenMP board released the version 4 (OpenMP, 2013) application program interface with support for external devices (including GPUs and Vector Processors). In this paper we compare the four implementation of Jacobi's factorization, to show the advantages and disadvantages of each framework.

METHODOLOGY

The frameworks used working as extensions of #pragmas of the C languages offering the simplest way to programming without development complicate and external elements. In the next section we describe the frameworks and give some implementations examples. In the last part, we show the pure CUDA kernels implementations.

Ompss

Ompss (a programming model form Barcelona Supercomputer center based on OpenMP and StarSs) is framework focusses on task decomposition paradigm for developing parallel applications on cluster environments with heterogeneous architectures. It provides a set of compiler directives that can be used to annotate a sequential code. Additional features have been added to support the use of accelerators like GPUs. OmpSS is based on StartsS a task based programming model. It is based on annotating a serial application with directives that are translated by the compiler. With it, the same program that runs sequentially in a node with a single GPU can run in parallel in multiple GPUs either local (single node) or remote (cluster of GPUs). Besides performing a task-based parallelization, the runtime system moves the data as needed between the different nodes and GPUs minimizing the impact of communication by using affinity scheduling, caching, and by overlapping communication with the computational task.

OmpSs is based on the OpenMP programming model with modifications to its execution and memory model. It also provides some extensions for synchronization, data motion and heterogeneity support.

1) Execution model: OmpSs uses a thread-pool execution model instead of the traditional OpenMP fork-join model. The master thread starts the execution and all other threads cooperate executing the work it creates (whether it is from work sharing or task constructs). Therefore, there is no need for a parallel region. Nesting of constructs allows other threads to generate work as well (Figure 1).

2) Memory model: OmpSs assumes that multiple address spaces may exist. As such shared data may reside in memory locations that are not directly accessible from some of the computational resources. Therefore, all parallel code can only safely access private data and shared data which has been marked explicitly with our extended syntax. This assumption is true even for SMP machines as the implementation may reallocate shared data to improve memory accesses (e.g., NUMA).

3) Extensions:

Function tasks: OmpSs allows to annotate function declarations or definitions Cilk (Durán, Pérez, Ayguadé, Badia & Labarta, 2008), with a task directive. In this case, any call to the function creates a new task that will execute the function body. The data environment of the task is captured from the function arguments.

Dependency synchronization: OmpSs integrates the StarSs dependence support (Durán *et al.,* 2008). It allows annotating tasks with three clauses: input, output, in/out. They allow expressing, respectively, that a given task depends on some data produced before, which will produce some data, or both. The syntax in the clause allows specifying scalars, arrays, pointers and pointed data.

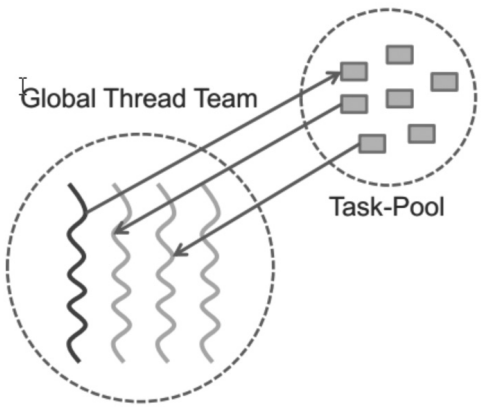

Figure 1. OmpSS execution model

Source: Barcelona supercomputing Center, p. 11. http://www.training.prace-ri.eu/uploads/tx_pracetmo/OmpSsQuickOverviewXT.pdf

OpenACC

OpenACC is an industry standard proposed for heterogeneous computing on SuperComputer Conference 2011. OpenACC follows the OpenMP approach, with annotation on Sequential code with compiler directives (pragmas), indicating those regions of code susceptible to be executed in the GPU.

The execution model targeted by OpenACC API-enabled implementations is host-directed execution with an attached accelerator device, such as a GPU. Much of a user application executes on the host. Compute intensive regions are offloaded to the accelerator device under control of the host. The device executes parallel regions, which typically contain work- sharing loops, or kernels regions, which typically contain one or more loops which are executed as kernels on the accelerator. Even in accelerator-targeted regions, the host may orchestrate the execution by allocating memory on the accelerator device, initiating data transfer, sending the code to the accelerator, passing arguments to the compute region, queuing the device code, waiting for completion, transferring results back to the host, and de-allocating memory (Figure 2). In most cases, the host can queue a sequence of operations to be executed on the device, one after the other (Wolfe, 2013).

The actual problems with OpenACC are relationship with the only for-join model support and support for only commercial compilers can support his directives (PGI, Cray and CAPS) (Wolfe, 2013; Reyes, López, fumero & Sande, 2012). In the last year, only one open source implementations has support (accULL) (Reyes & López-Rodríguez, 2012).

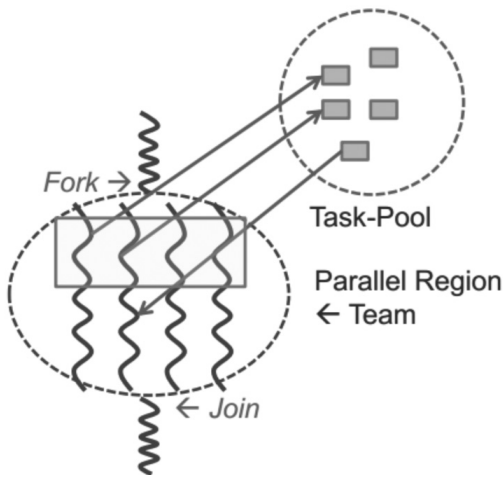

Figure 2. OpenACC execution model

Source: Barcelona supercomputing Center p. 11. http://www.training.prace-ri.eu/uploads/tx_pracetmo/OmpSsQuickOverviewXT.pdf

OpenMPC

The OpenMPC (OpenMP extendent for CUDA) is a framework to hide the complexity of programming model and memory model to user (Lee & Eigenmann, 2010). OpenMPC consists of a standard OpenMP API plus a new set of directives and environment variables to control important CUDA-related parameters and optimizations.

OpenMPC addresses two important issues on GPGPU programming: programmability and tunability. OpenMPC as a front-end programming model provides programmers with abstractions of the complex CUDA programming model and high-level controls over various optimizations and CUDA-related parameters. OpenMPC included fully automatic compilation and user-assisted tuning system supporting OpenMPC. In addition to a range of compiler transformations and optimizations, the system includes tuning capabilities for generating, pruning, and navigating the search space of compilation variants.

OpenMPC use the compiler cetus (Dave, Bae, Min & Lee, 2009) for automatic parallelization source to source. The Source code on C has 3 level of analyzing

- Privatization
- Reduction Variable Recognition
- Induction Variable substitution

OpenMPC adding a numbers of pragmas for annotate OpenMP parallel regions and select optimization regions. The pragmas added has the following form: #pragma cuda <<function>>.

OpenMP release 4

OpenMP is the most used framework for programming parallel software with shared memory and support on most of the existing compilers. With the explosion of multicore and manycore system, OpenMP gains acceptance on parallel programming community and hardware vendors. From his creation to version 3 the focus of API was the CPUs environments, but with the introduction of GPUs and vector accelerators, the new 4 release includes support for external devices (OpenMP, 2013). Historically, OpenMP has support Simple Instruction Multiple Data (SIMD) model only focusses on fork-join model (Figure 3), but in this new release the task-base model (Duran *et al.,* 2008; Podobas, Brorsson & Faxén, 2010) has been introduced to gain performance with more parallelism on external devices. The most important directives introduced were target, teams and distributed. This directives permit that a group of threads was distributed on a special devices and the result was copied to host memory (Figure 3).

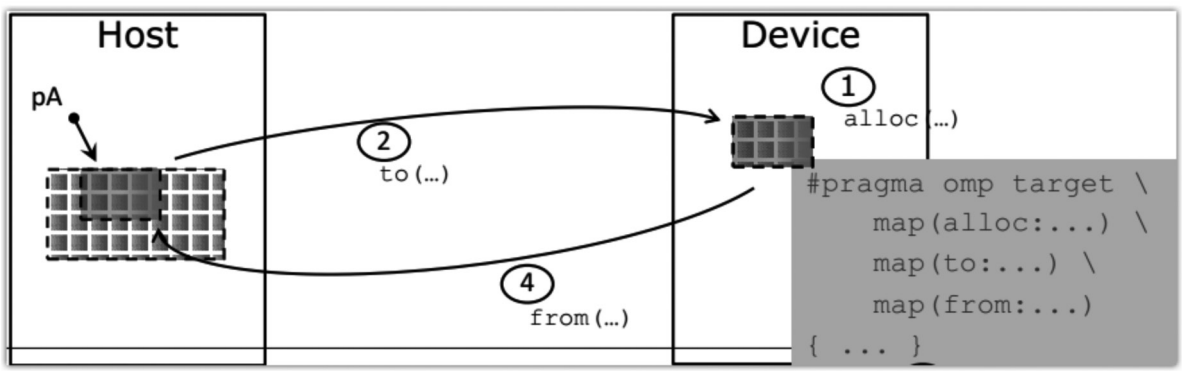

Figure 3. OpenMP 4 execution model

Source: Intel Parallel OpenMP. http://www.theclassifiedsplus.com/video/video/axnA3kcLHK4/intel-parallel-openmp.html

JACOBI ITERATIVE METHOD

Iterative methods are suitable for large scale linear equations. There are three commonly used iterative methods: Jacobi's method, Gauss method and SOR iterative methods (Gravvanis, Filelis-Papadopoulos & Lipitakis, 2013; Huang, Teng, Wahid & Ko, 2009).

The last two iterative methods convergence speed is faster than Jacobi's iterative method but lack of parallelism. They have advantages to Jacobi method only when implemented in sequential fashion and executed on traditional CPUs. On the other hand, Jacobi's iterative method has inherent parallelism. It's suitable to be implemented on CUDA or vector accelerators to run concurrently on many cores. The basic idea of Jacobi method is convert the system into equivalent system then we solved Equation (1) and Equation (2):

$$x = B \begin{array}{l} a_{11}x_{11} + a_{12}x_2 + a_{13}x_3 = b_1 \\ a_{21}x_{22} + a_{22}x_2 + a_{23}x_3 = b_2 \\ a_{31}x_{33} + a_{32}x_2 + a_{33}x_3 = b_3 \end{array} \quad (1)$$

$$x_1 = -\frac{a_{12}}{a_{11}}x_2 - \frac{a_{13}}{a_{11}}x_3 + \frac{b_1}{a_{11}}$$

$$x_2 = -\frac{a_{21}}{a_{22}}x_1 - \frac{a_{23}}{a_{22}}x_3 + \frac{b_2}{a_{22}}$$

$$x_3 = -\frac{a_{31}}{a_{33}}x_1 - \frac{a_{32}}{a_{33}}x_2 + \frac{b_3}{a_{33}}$$

On each iteration we solve :

$$x^k{}_i = \frac{1}{a_{i,i}}\left[b_i - \sum_{j \neq i} a_{i,j}x^{k-1}{}_j\right] \quad (2)$$

Where the values from the (*k-1*) iteration are used to compute the values for the *k*th iteration. The pseudo code for Jacobi method (Dongarra et al., 2008):

```
Choose an initial guest    to the solution x.
for k=1,2,…
                for i=1,2,…n
                    x_i=0
                    for j=1,2,…,i-1,i+1,…n
                            x_i = x_i + a_{i,j}x_j^(k-1)
                    end
                    x_i = (b_i + x_i)/ a_{i,j}
                end
                x^(k)=x
                check convergence; continue
if necessary
end
```

This iterative method can be implemented on a parallel form, using shared or distributed memory (Margaris, Souravlas & Roumeliotis, 2014) (Figure 4). For distributed memory, it needs some explicit synchronization and data out-process data copy. In share memory, it needs distribution and data merge in memory. It uses the following method:

$$x^{k+1} = D^{-1}(b - (L+U)x^k) \quad (3)$$

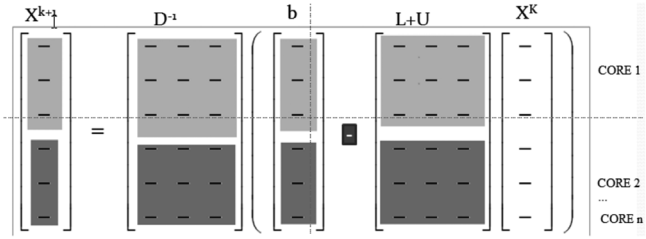

Figure 4. Parallel form of Jacobi method on shared memory (Alsemmeri, n.d.)

Source: Parallel Jacobi Algorithm https://www.cs.wmich.edu/~elise/courses/cs626/s12/PARALLEL-JACOBI-ALGORITHM11.pptx

OPENMP IMPLEMENTATION

```
int n=LIMIT_N;
int m=LIMIT_M;
A[n][m]; // the D^-1 matrix
```

```
    Anew [n][m];
    y_vector[n]; //b vector
    //fill the matriz with initial conditions
    …
    #pragma omp parallel for shared (m, n, Anew,
A)
        for (int j = 1; j < n-1; j++)
        {
          for (int i = 1; i < m-1; i++ )
          {
            Anew[j][i] = 0.25f * ( A[j][i+1] + A[j][i-1]
                    + A[j-1][i] + A[j+1][i]);
            error = fmaxf (error, fabsf(Anew[j][i]-A[j]
[i]));
          }
        }
    #pragma omp parallel for shared (m, n,
Anew, A)
      for (int j = 1; j < n-1; j++)
        {
          for (int i = 1; i < m-1; i++ )
          {
            A[j][i] = Anew[j][i];
          }
        }
        if (iter % 100 == 0) printf("%5d, %0.6f\n",
iter, error);
        iter++;
      }
    …//print the result
```

In this section, the for-joint model appears on section annotate with *#pragma omp parallel for shared (m, n, Anew, A)* where every threads (normally equals to cores) on system running a copy of code with different data section shared all variables named on *shared()* section. When the size of *m* and *n* is minor or equals to number of cores, the performance is similar on GPUs and CPUs, but when the size if much higher that number of cores available, the performance of GPUs increases because the parallelism level is higher (Fowers, Brown, Cooke & Stitt, 2012; Zhang, Miao, & Wang, 2009).

OPENACC IMPLEMENTATION

```
    int n=LIMIT_N;
    int m=LIMIT_M;
    A[n][m]; // the D-1 matrix
    Anew [n][m];
    y_vector[n]; //b vector
    //fill the matriz with initial conditions
    …
    #pragma omp parallel for shared (m, n, Anew,
A)
    #pragma acc kernels
        for( int j = 1; j < n-1; j++)
        {
          for( int i = 1; i < m-1; i++ )
          {
            Anew[j][i] = 0.25f * ( A[j][i+1] + A[j][i-1]
                    + A[j-1][i] + A[j+1][i]);
            error = fmaxf( error, fabsf(Anew[j][i]-A[j]
[i]));
          }
        }
    #pragma omp parallel for shared (m, n, Anew,
A)
    #pragma acc kernels
      for( int j = 1; j < n-1; j++)
        {
          for( int i = 1; i < m-1; i++ )
          {
            A[j][i] = Anew[j][i];
          }
        }
        if(iter % 100 == 0) printf("%5d, %0.6f\n",
iter, error);
        iter++;
      }
    …//print the result
```

In this implementation appears a new annotation #pragma acc kernels, where it indicates that the code will be executed. This simple annotation hides a complex implementation of CUDA kernel, the copy of data from host to devices and devices to GPU and definition of grid of threads and the

data manipulation (Amorim & Haase, 2009; Sanders & Kandrot, 2011; Zhang *et al.,* 2009).

PURE CUDA IMPLEMENTATION (WANG, N.D.)

```
…
int dimB, dimT;
 dimT = 256;
 dimB = (dim / dimT) + 1;

 float err = 1.0;

 // set up the memory for GPU
 float * LU_d;
 float * B_d;
 float * diag_d;
 float *X_d, *X_old_d;
 float * tmp;

  cudaMalloc( (void **) &B_d, sizeof(float) *
dim );
  cudaMalloc( (void **) &diag_d, sizeof(float)
* dim );
  cudaMalloc( (void **) &LU_d, sizeof(float) *
dim * dim);

 cudaMemcpy( LU_d, LU, sizeof(float) * dim *
dim, cudaMemcpyHostToDevice);
  cudaMemcpy( B_d, B, sizeof(float) * dim,
cudaMemcpyHostToDevice);
  cudaMemcpy( diag_d, diag, sizeof(float) *
dim, cudaMemcpyHostToDevice);

  cudaMalloc( (void **) &X_d, sizeof(float) *
dim);
 cudaMalloc( (void **) &X_old_d, sizeof(float)
* dim);
  cudaMalloc( (void **) &tmp, sizeof(float) *
dim);
…
//call to cuda kernels

 // 2. Compute X by A x_old
  cudaMemcpy( X_old_d, x_old, sizeof(float)
* dim, cudaMemcpyHostToDevice);
 matMultVec<<<dimB, dimT>>>(LU_d, X_
old_d, tmp, dim, dim); // use x_old to compute
LU X_old and store the result in tmp
      substract<<<dimB, dimT>>>(B_d, tmp,
X_d, dim);   // get the (B - LU X_old), which is
stored in X_d
      diaMultVec<<<dimB, dimT>>>(diag_d,
X_d, dim);   // get the new X

      // 3. copy the new X back to the Host
Memory
  cudaMemcpy( X, X_d, sizeof(float) * dim,
cudaMemcpyDeviceToHost);

    // 4. calculate the norm of X_new - X_old
    substract<<<dimB, dimT>>>(X_old_d, X_d,
tmp, dim);
    VecAbs<<<dimB, dimT>>>(tmp, dim);
    VecMax<<<dimB, dimT>>>(tmp, dim);
    // copy the max value from Device to Host
 cudaMemcpy(max, tmp, sizeof(float),
cudaMemcpyDeviceToHost);
 // cuda kernel for vector multiplication
 __global__ void matMultVec(float * mat_A,
          float * vec,
          float * rst,
          int dim_row,
          int dim_col)
 {
   int rowIdx = threadIdx.x + blockIdx.x *
blockDim.x; // Get the row Index
  int aIdx;
  while(rowIdx < dim_row)
  {
    rst[rowIdx] = 0; // clean the value at first
    for (int i = 0; i < dim_col; i++)
    {
      aIdx = rowIdx * dim_col + i; // Get the
index for the element a_{rowIdx, i}
     rst[rowIdx] += (mat_A[aIdx] * vec[i] ); // do
the multiplication
    }
```

```
      rowIdx += gridDim.x * blockDim.x;
    }
    __syncthreads();
  }

  // cuda kernel for vector subtraction
  __global__ void substract(float *a_d,
             float *b_d,
             float *c_d,
             int dim)
  {
   int tid = threadIdx.x + blockIdx.x * blockDim.x;
    while ( tid < dim )
    {
      c_d[tid] = a_d[tid] - b_d[tid];
      tid += gridDim.x * blockDim.x;
    }
  }

  __global__ void VecMax(float * vec, int dim)
  {
   int tid = threadIdx.x + blockIdx.x * blockDim.x;
    while (dim > 1)
    {
      int mid = dim / 2; // get the half size
      if (tid < mid)  // filter the active thread
      {
       if (vec[tid] < vec[tid+mid] ) // get the larger
one between vec[tid] and vec[tid+mid]
          vec[tid] = vec[tid+mid]; // and store the
larger one in vec[tid]
      }

      //deal with the odd case
      if (dim % 2 )   // if dim is odd...we need care
about the last element
      {
       if (tid == 0 ) // only use the vec[0] to com-
pare with vec[dim-1]
        {
          if (vec[tid] < vec[dim-1] )
            vec[tid] = vec[dim-1];
        }
      }

      __syncthreads(); // sync all threads
      dim /= 2;    // make the vector half size short.
    }
  }
```

The effort for writing three kernels, management the logic of grids dimensions, copy data from hosts to GPU and GPU to host, the aspect of synchronization thread on the groups of Threads on GPU and some aspects as ThreadID calculation required high computation on hardware devices and code programming.

TEST TECHNIQUE

We take the three implementations of Jacobi method (Pure OpenMP, OpenACC, OpenMPC) and running it on SUT (System Under Test) of Table 1, and make multiples running with square matrices of incremental sizes (Table 2), take processing time for analyzing the performance (Kim, n.d.; Sun & Gustafson, 1991) against the code number lines needed on the algorithm.

Table 1. Characteristics of System under Test

System	Supermicro SYS-1027GR-TRF
CPU	Intel® Xeon® 10-core E5-2680 V2 CPUs @ 2.80 GHz
Memory	32GB DDR3 1600MHz
GPU	NVIDIA Kepler K40 GPUs, 2880 Cuda Cores, Memory 12GB
OS	Red Hat Enterprise Linux Server release 6.4
Compiler	PGI Compiler Accelerator Fortran/C/C++ 14 release 9 for Linux
L2 Cache	256K
L3 Cache	25MB

Table 2. Matrix size performance comparison.

Matrix size (N)	Implementation	Mean Running Time (Seconds)
2048	OpenMP with 20 Threads	26.578
	OpenACC	74,970
	OpenMPC	69,890
4096	OpenMP with 20 Threads	74,280
	OpenACC	92,320
	OpenMPC	75,120
8096	OpenMP with 20 Threads	116,23
	OpenACC	98,00
	OpenMPC	101,420
16192	OpenMP with 20 Threads	522.320
	OpenACC	190,230
	OpenMPC	222,320

RESULT

The performance on the three frameworks (OpenMP v. 4, OpenACC, OpenMPC) presents a similar result on square matrices with size of 2048; however, if the size ingresses to 4096, the performance is little high on OpenACC implementation. With huge matrices greater than 12288, another factor as cache L2 and L3 has impact on process of data copy from host to device (Bader & Weidendorfer, 2009; Barragan & Steves, 2011; Gupta, Xiang, & Zhou, 2013). The performance of using a framework against using direct GPU CUDA languages, was just a bit (<5%), but the number of code lines was 70% minor. This shows that the framework offers useful programming tools with computational performance advantage; however, the performance gains on GPUs is on region of algorithms with high level of parallelism, and low data coupling with the host code (Dannert, Marek, & Rampp, 2013).

CONCLUSION

The new devices for high performance computing need standardized methods and languages that permits interoperability, easy programming and well defined interfaces for integrating the data interchange between memory segments of the processors (CPUs) and devices. Besides, it is necessary that languages have support for working with two or more devices on parallel using the same code but running segments of high parallelism in automatically form. OpenMP is the *de facto* standard for shared memory programming model, but the support for heterogeneous devices (Gpus, accelerators, fpga, etc.) is in very early stage, the new frameworks and industrial API need help for a growing and maturating standard.

FINANCING

This research was developed with own resources, in the Doctoral thesis process at Universidad Distrital Francisco José de Caldas.

FUTURE WORKS

It is necessary to analyze the impact of frameworks for multiple devices on the same host for management problems of memory locality, unified memory between CPU and devices and I/O operations from devices using the most recent version of its standard framework. With the support of GPUs device on the following version of GCC5, it is possible to obtain a compiler performance comparison between commercial and standards open source solutions. Furthermore, it's necessary to review the impact of using cluster with multi-devices, messaging pass and MPI integration for high performance computing using the language extensions (Schaa & Kaeli, 2009).

ACKNOWLEDGMENT

We express our acknowledgment to director of high performance center of Universidad Industrial de Santander, the director of advanced computing center of Universidad Distrital and Pak Liu Technical responsible for application characterization, profiling and testing of HPC Advisor Council www.hpcadvisorycouncil.com

REFERENCES

Alsemmeri, M. (n.d.). CS 6260. Advanced Parallel Computations Course. Retrieved October 3, 2014, from https://cs.wmich.edu/elise/courses/cs626/home.htm

Amorim, R. & Haase, G. (2009). Comparing CUDA and OpenGL implementations for a Jacobi iteration. *… & Simulation, 2009. …*, 22–32. doi:10.1109/HPCSIM.2009.5192847

Bader, M. & Weidendorfer, J. (2009). Exploiting memory hierarchies in scientific computing. *2009 International Conference on High Performance Computing & Simulation*, 33–35. doi:10.1109/HPCSIM.2009.5192891

Barragan, E. H. & Steves, J. J. (2011). Performance analysis on multicore system using PAPI. In *2011 6th Colombian Computing Congress (CCC)* (pp. 1–5). IEEE. doi:10.1109/COLOMCC.2011.5936277

Dannert, T., Marek, A. & Rampp, M. (2013). Porting Large HPC Applications to GPU Clusters: The Codes GENE and VERTEX. Retrieved from http://arxiv.org/abs/1310.1485v1

Dave, C., Bae, H., Min, S. & Lee, S. (2009). Cetus: A source-to-source compiler infrastructure for multicores. *Computer* (0429535). Retrieved from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5353460

Dongarra, J., Golub, G. H., Grosse, E., Moler, C. & Moore, K. (2008). Netlib and NA-Net: Building a Scientific Computing Community. *Annals of the History of Computing, IEEE*, *30*(2), 30–41. doi:10.1109/MAHC.2008.29

Durán, A. et al. (2008). Extending the OpenMP tasking model to allow dependent tasks. *Lecture Notes in Computer Science*, *5004*, 111–122. doi:10.1007/978-3-540-79561-2_10

Fowers, J., Brown, G., Cooke, P., & Stitt, G. (2012). A performance and energy comparison of FPGAs, GPUs, and multicores for sliding-window applications. *Proceedings of the ACM/SIGDA …*, 47. doi:10.1145/2145694.2145704

Gravvanis, G. a., Filelis-Papadopoulos, C. K. & Lipitakis, E. a. (2013). On numerical modeling performance of generalized preconditioned methods. *Proceedings of the 6th Balkan Conference in Informatics on - BCI '13*, 23. doi:10.1145/2490257.2490266

Gupta, S., Xiang, P. & Zhou, H. (2013). Analyzing locality of memory references in GPU architectures. *… of the ACM SIGPLAN Workshop on Memory …*, 1–2. doi:10.1145/2492408.2492423

Hill, M. & Marty, M. (2008). Amdahl's law in the multicore era. *Computer*, *41*(7), 33–38. doi:10.1109/MC.2008.209

Huang, P., Teng, D., Wahid, K. & Ko, S.-B. (2009). Performance evaluation of hardware/software codesign of iterative methods of linear systems. *2009 3rd International Conference on Signals, Circuits and Systems (SCS)*, 1–5. doi:10.1109/ICSCS.2009.5414164

Kim, Y. (2013). Performance tuning techniques for GPU and MIC. *2013 Programming weather, climate, and earth-system models on heterogeneous multi-core platforms*. Boulder, CO, National Oceanic and Atmospheric Administration.

Lee, S. & Eigenmann, R. (2010). OpenMPC: Extended OpenMP programming and tuning for GPUs. *Proceedings of the 2010 ACM/IEEE International …*, (November), 1–11. doi:10.1109/SC.2010.36

Margaris, A., Souravlas, S. & Roumeliotis, M. (2014). Parallel Implementations of the Jacobi Linear Algebraic Systems Solve. *arXiv Preprint arXiv:1403.5805*, 161–172. Retrieved from http://arxiv.org/abs/1403.5805

Meijerink, J. & Vorst, H. van der. (1977). An iterative solution method for linear systems of which the coefficient matrix is a symmetric -matrix. *Mathematics of Computation*, *31*(137), 148–162. Retrieved from http://www.ams.org/mcom/1977-31-137/S0025-5718-1977-0438681-4/

Nickolls, J. & Dally, W. (2010). The GPU computing era. *Micro, IEEE*, *30*(2), 56–69. doi:10.1109/MM.2010.41

OpenMP, A. (2013). *OpenMP application program interface, v. 4.0*. *OpenMP Architecture Review Board*. Retrieved from http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:OpenMP+Application+Program+Interface#1

Podobas, A., Brorsson, M. & Faxén, K. (2010). A comparison of some recent task-based parallel programming models, 1–14. Retrieved from http://soda.swedish-ict.se/3869/

Reyes, R., López, I., Fumero, J. & Sande, F. De. (2012). A Comparative Study of OpenACC Implementations. *Jornadas Sarteco*. Retrieved from http://www.jornadassarteco.org/js2012/papers/paper_150.pdf

Reyes, R. & López-Rodríguez, I. (2012). accULL: An OpenACC implementation with CUDA and OpenCL support. *Euro-Par 2012 Parallel …*, *2*(228398), 871–882. Retrieved from http://link.springer.com/chapter/10.1007/978-3-642-32820-6_86

Sanders, J. & Kandrot, E. (2011). CUDA by Example. *An Introduction to General-Purpose GPU Programming* …. Retrieved from http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:CUDA+by+Example#1

Schaa, D. & Kaeli, D. (2009). Exploring the multiple-GPU design space. *2009 IEEE International Symposium on Parallel & Distributed Processing*, 1–12. doi:10.1109/IPDPS.2009.5161068

Sun, X. & Gustafson, J. (1991). Toward a better parallel performance metric. *Parallel Computing*, 1093–1109. Retrieved from http://www.sciencedirect.com/science/article/pii/S0167819105800286

Vetter, J. (2012). On the road to Exascale: lessons from contemporary scalable GPU systems. … */A* CRC Workshop on Accelerator Technologies for* …. Retrieved from http://www.acrc.a-star.edu.sg/astaratipreg_2012/Proceedings/Presentation - Jeffrey Vetter.pdf

Wang, Y. (n.d.). Jacobi iteration on GPU. Retrieved September 25, 2014, from http://doubletony-pblog.azurewebsites.net/jacboi_gpu.html

Wolfe, M. (2013). The OpenACC Application Programming Interface, *2*.

Zhang, Z., Miao, Q. & Wang, Y. (2009). CUDA-Based Jacobi's Iterative Method. *Computer Science-Technology and …*, 259–262. doi:10.1109/IFCSTA.2009.68