ANN. MULT. GPU PROG.

Study of basic vector operations on Intel Xeon Phi and NVIDIA Tesla using OpenCL

Edoardo Coronado-Barrientos*, Guillermo Indalecio and Antonio Garcia-Loureiro

*Correspondence:
edoardoemilio.coronado@usc.es
Centro de Investigacion en Tecnoloxias da Informacion (CiTIUS), Universidad de Santiago de Compostela, Rua de Jenaro de la Fuente Dominguez, 15782 Santiago de Compostela, Spain
Full list of author information is available at the end of the article

Abstract

The present work is an analysis of the performance of the basic vector operations AXPY, DOT and SpMV using OpenCL. The code was tested on the NVIDIA Tesla S2050 GPU and Intel Xeon Phi 3120A coprocessor. Due to the nature of the AXPY function, only two versions were implemented, the routine to be executed by the CPU and the kernel to be executed on the previously mentioned devices. It was studied how they perform for different vector's sizes. Their results show the NVIDIA architecture better suited for the smaller vectors sizes and the Intel architecture for the larger vector's sizes. For the DOT and SpMV functions, there are three versions implemented. The first is the CPU routine, the second one is an OpenCL kernel that uses local memory and the third one is an OpenCL kernel that only uses global memory. The kernels that use local memory are tested by varying the size of the work-group; the kernels that only uses global memory are tested by varying the arrays size. In the case of the first ones, the results show the optimum work-group size and that the NVIDIA architecture benefits from the use of local memory. For the latter kernels, the results show that larger computational loads benefits the Intel architecture.

Keywords: Nvidia Tesla; Intel Xeon Phi; OpenCL; kernel; vector operations; AXPY; DOT; SPMV

Introduction

Nowadays, the top supercomputer systems are heterogeneous systems that use more than one kind of processors to gain performance. One of the leading vendors to provide accelerators for high performance computing is NVIDIA. As of November 2014, 61.3 percent of the heterogeneous systems at Top500 use NVIDIA to provide acceleration to their systems [1]. More concretely, the number 2 and 6 of the list run with NVIDIA. This situation differs greatly from the state at June 2012, where 91.5 percent of the heterogeneous systems used NVIDIA. The difference between then and now is the incursion of the Intel Xeon Phi coprocessor for acceleration. In June 2012 only one system was found using this card at Top500, and now in November 2014, 28 percent of the heterogeneous systems use Xeon Phi, including the first in the list. These two platforms have different programming APIs and memory models, which complicates the comparison of the algorithm's performance between them. In order to simplify the development in heterogeneous systems, the OpenCL project provides a single interface that enables the execution of the same code in any compliant device. This is a wide available and lively developed framework maintained by a non-profit consortium, the Khronos Group [2], and adopted by many vendors like Apple, Intel, AMD, and some more, by providing support on their devices. Both the NVIDIA Tesla S2050 and the Intel Xeon Phi 3120A are compatible, so it is possible to write code to both and make a direct comparison of the performance.

With heterogeneous systems positioned on the first places in the Top500, having the same code that can be executed in several platforms is becoming a necessity instead of an option. The present work is inspired in this reality, so the objective is to analyse the kernels' performance to improve executions'

time of algorithms of linear systems solvers [3]. The focus was on solvers that could manage to find the solution for systems whose matrix is not symmetric, which are commonly generated by numerical simulations, including semiconductors devices [4] among other research fields. The kernels tested in this work were included in a three-dimensional Finite Element simulator tool developed by our group. The applications of this simulation tool include the study of several semiconductor devices like: Bipolar Junction Transistors [5], High-Electron Mobility Transistors [6], Metal-Oxide-Semiconductor Field-Effect Transistors [7] and Solar Cells [8].

This work has the following organization: on section **Justification** there is a brief justification of the present analysis; on section **Specification of devices** the technical specifications of the NVIDIA Tesla and Intel Xeon Phi devices are given; on section **OpenCL** a brief introduction to the terminology used by OpenCL is done; on section **Kernels** an analysis of the implemented kernels is done; on section **Results** the results of the tests are registered; and the conclusions are summarised on the last section.

Justification

The main classification for sparse linear systems solver are: direct solutions methods and iterative methods. An example of the first class is the LU factorization. The iterative methods can be classified in two classes: basic iterative methods and projection methods. The latter class has the most important iterative methods for solving large sparse linear systems [9]. Examples of these algorithms are: Full Orthogonalization Method (FOM), Incomplete Orthogonalisation Method (IOM), Generalized Minimal Residual Method (GMRES).

The most used operations in iterative methods for solving sparse linear systems, whose matrix is unsymmetric [9], are: the AXPY function ($a\mathbf{x}+\mathbf{y}$), the DOT function ($\mathbf{x}\cdot\mathbf{y}$) and the SpMV function ($\mathbf{Ax}$), where a is a scalar, $\mathbf{x},\mathbf{y}$ are vectors and $\mathbf{A}$ is a square sparse matrix. For example, having a look on one iterative method for solving sparse linear systems, the Flexible variant of the Generalized Minimal Residual method, shows that the algorithm depends highly on these operations (see Fig. 1). Note that also the norm-2 has a dependency on the DOT function, as its calculation requires a dot product of a vector with itself.

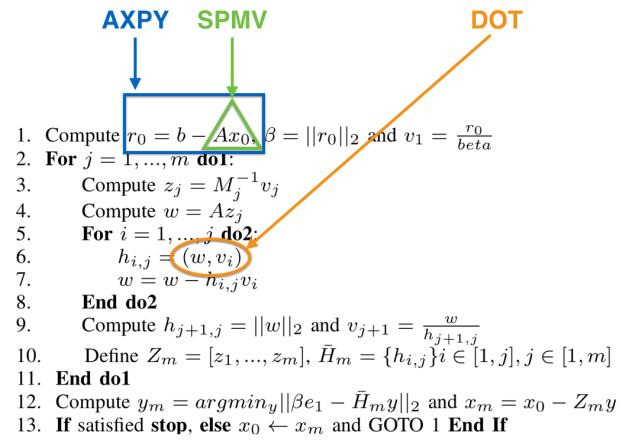

Figure 1 Functions of the FGMRES algorithm. For clarity, it has been located one instance of each function in the code, but several others can be found on other steps of the code.

The performance of these functions have a considerable dependency on the specific implementation. Considering the differences in architecture and capabilities of the available coprocessors and General

Purpose Graphics Processing Units (GPGPUs), changing the implementation of the kernels will have an extra impact on the performance of the functions, hence boosting or decreasing the running time of the linear system solver.

Specifications of devices

In this work, two devices are used in order to comprehend and analyse the differences between kernel implementations and performance. The first is a High Performance Computing (HPC) oriented GPU from NVIDIA, the Tesla S2050, which represents a well tested and established product. Also, it is used the novel Xeon Phi Coprocessor from Intel, arising with a different paradigm of more complex cores and logic. Both devices have support for OpenCL, which makes them perfect to be compared using a common application program interface (API). In TABLE 1 is a summary of the main specifications of the NVIDIA Tesla S2050 and the Intel Xeon Phi 3120A.

GPUs were originally developed to be used in graphics manipulation, so they have very simple processing cores, however the GPGPUs have been designed to be more useful in the field of HPC, starting with the Fermi line. The NVIDIA Tesla S2050 is a Fermi based GPU which has two slots, each having two M2050 cards of 14 CUDA cores (Streaming Multiprocessors) for a total of 448 cores and 3,072 MB of memory. Each GPU core operates at a frequency of 1.1 GHz. Its theoretical peak performance is 1028 GFLOPS with single precision [10].

The Intel Xeon Phi coprocessor contains 57 cores, with a 512-bit vector arithmetic unit, for SIMD vector instructions. The coprocessor has an L1 cache in each core of 64 KB and L2 cache in each core of 512 KB (both are combined Data and Instructions). Each core can execute up to four hardware threads simultaneously and operates at a frequency of 1.1 GHz. The theoretical peak performance of the Intel Xeon Phi coprocessor is 1011 GFLOPS [11].

The NVIDIA architecture is benefited by parallelism of fine grain due to larger quantity of lighter cores that are not capable of supporting SIMD instructions. On the other hand the Intel Xeon Phi coprocessor has heavier cores that supports SIMD instruction and with its own cache hierarchy implying a necessity for a complex mechanism for cache coherency. Because of this, the latter is benefited by parallelism of coarse grain. Another important difference to consider between these two devices, is where each has allocated the local memory. For the NVIDIA Tesla architecture, local memory is allocated on each CUDA core and its available to each of its 32 cores; on the other hand, for the Intel Xeon Phi architecture the local memory is allocated on the regular GDDR memory, so it introduces additional overhead in terms of redundant data copy and management if used [12].

Table 1 Specifications for the NVIDIA Tesla S2050 and Intel Xeon Phi 3120A devices.

	Tesla S2050	Xeon Phi 3120A
Cores	448	57
Core clock (GHz)	1.10	1.55
Total memory (GB)	3	6
Memory max bandwidth (GB/s)	148	240
TDP (W)	225	300
Maximum work-group size	1024	8192
Double arithmetic support	yes	yes
Cache L2 (MB)		28.5
Cache line size (B)	128	64
Device type	GPU	Accelerator card

OpenCL

OpenCL is a programming framework that enables the utilisation of the same code on multiplatform systems [13], [14], [15]. In this work, OpenCL provided a common ground to test functions on two

devices with very different architectures. As a new programming paradigm it has its own terminology, much as CUDA does the same for the NVIDIA's GPUs.

The description of the elements of the OpenCL frame is needed to explain the behaviour of the kernels along the article. Please note that these elements will have a different counterpart in the physical device depending on the card. Especially the last three (compute unit, work-item and work-group), and the memory model, as explained in section **Specification of devices**. The key concepts list is as follows:

- **Host**: the Central Processing Unit (CPU) were the code is being executed.
- **Host application**: the code that contains the OpenCL instructions and manages the kernels.
- **Kernel**: is a special coded function to be executed by one or more devices.
- **Device**: any processing unit, such as CPUs, GPUs or accelerator cards (coprocessor), that is OpenCL complaint.
- **Context**: a container used by the host to manage all the connected devices.
- **Program**: the container where the kernels are searched by the host to send into a device.
- **Queue**: is a mechanism used by the host to dispatch kernels from a program into a device.
- **Compute unit**: a processing core contained within a device.
- **Work-item**: is an individual kernel execution with a specific set of data.
- **Work-group**: is a set of work-items that access the same processing resources (e.g. local memory).

The host is running the application that contains all the code, namely *source.c*. Once the host finds the instructions that set the frame for OpenCL, it looks into the program container, namely *kernels_source.cl*, for kernels. Once the host have access to the kernels, it can send them to a specific device included into the context by its queue, as shown on Fig. 2. Note that each device needs its own queue, so if three devices are available on the platform, three queues will be required to communicate with each one of them.

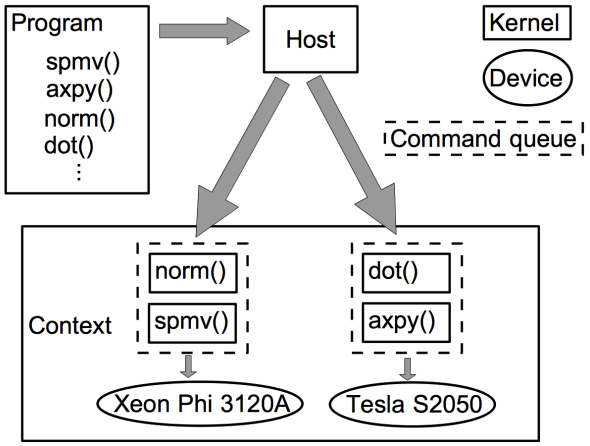

Figure 2 OpenCL frame.

The last three terms of the key terms list are more related to the OpenCL's device model. It is important to know this model because explains how memory address spaces are accessed. The OpenCL device model identifies four address spaces:

- **Global memory**: write/read memory that stores data for the entire device.
- **Constant memory**: read memory that stores data for the entire device.

- **Local memory**: stores data for work-items in a work-group.
- **Private memory**: stores data for a single work-item.

In Fig. 3 the relations between the different types of memory and the work-items, work-groups and compute units are shown.

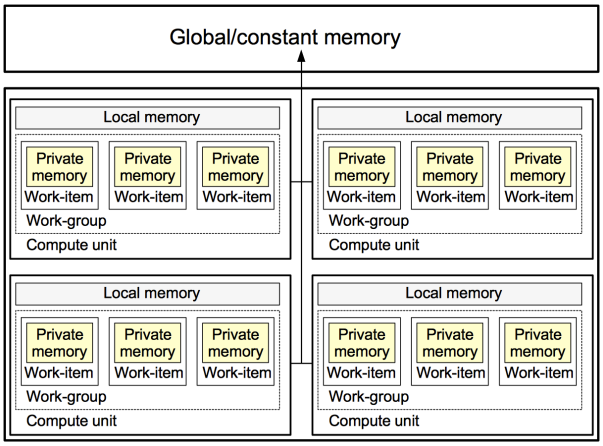

Figure 3 Schematic representation of the OpenCL device model.

The global memory is the largest and also the slowest memory region on the device. When the host sends data to the device, it writes directly into global memory, and when the host receives data from the device it reads from global memory. The constant memory has the same characteristics of the global memory, except it has read-only access. The local memory is faster and commonly much smaller than global memory, the NVIDIA Tesla architecture benefits from using it, as later results will show. Finally, the private memory is the fastest and smallest memory region, it can be only accessed by individuals work-items.

Kernels

Several implementations of the functions are done in order to analyse them, using OpenCL, so kernels can run in both devices without changes. Besides the length of the arrays to process, there is another parameter that influence the performance of some of these kernels, the work-group size. In section **Results** is showed the impact of this parameter.

The AXPY function.

There are two versions of the AXPY function, the CPU version and one OpenCL version. There is only one OpenCL version of this function because is almost a simple read-write operation giving no space to program a different version. This operation was included to see which device performed better for a highly parallel operation like this.

The CPU version of the AXPY function is:

```
for (i = 0; i < LENGTH; i++)
    r[i] = a*x[i] + y[i];
```

and the kernel for the AXPY function is:

```
__kernel void axpy(          float a,
                   __global float *x,
                   __global float *y,
```

```
                    __global float *r)
{
    unsigned int gid = get_global_id(0);
    r[gid] = a*x[gid] + y[gid];
}
```

In this kernel, one work-item computes one component of the vector r, this is, the work-item whose $gid = 0$ will compute $r[0] = a * x[0] + y[0]$, the work-item with $gid = 1$ will compute $r[1] = a * x[1] + y[1]$ and so on, until all elements of vector r are calculated (see Fig. 4). This is the perfect example of a highly parallel routine as all rows operations are independent of each other and each work-item access a single memory position per array consecutively .

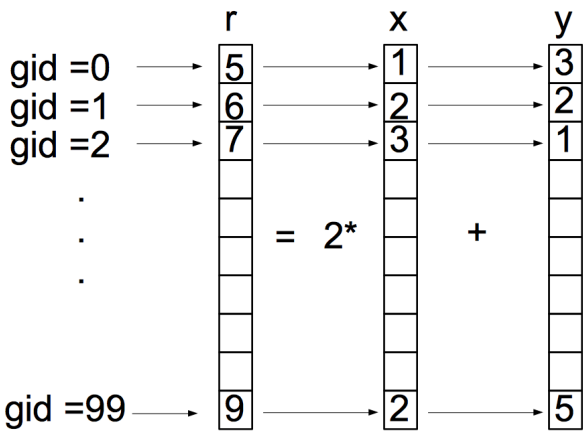

Figure 4 Array accesses by work-items in kernel $axpy$. This figure contains the example of $\mathbf{r} = 2\mathbf{x} + \mathbf{y}$ with $\mathbf{r}, \mathbf{x}, \mathbf{y}$ arrays of 100 elements.

The DOT function

There are three versions of the DOT function. The first version is the CPU implementation, the second version is an OpenCL kernel, optimized for the NVIDIA architecture as it uses local memory (see section **Specification of devices**), and the third version is an OpenCL kernel optimized for the Intel Xeon Phi architecture, as it does not use local memory (see section **Specification of devices**).

The CPU version is:

```
r = 0.0f;
for (i = 0; i < LENGTH; i++)
    r = r + v1[i]*v2[i];
```

the first kernel of the DOT function is:

```
__kernel void dot0(__global float *v1,
                   __global float *v2,
                   __global float *vrp)
{
  unsigned int lid = get_local_id(0);
  unsigned int ls = get_local_size(0);
  unsigned int grpid = get_group_id(0);
  unsigned int gid = grpid*(ls*2) + lid;
  unsigned int s;
  __local float LB[WORK_GROUP_SIZE];
  LB[lid] = v1[gid]*v2[gid] + v1[gid+ls]*v2[gid+ls];
```

```
  barrier(CLK_LOCAL_MEM_FENCE);
  if (lid < 8) LB[lid] = LB[lid] + LB[lid+8];
  if (lid < 4) LB[lid] = LB[lid] + LB[lid+4];
  if (lid < 2) LB[lid] = LB[lid] + LB[lid+2];
  if (lid < 1) LB[lid] = LB[lid] + LB[lid+1];
  if (lid==0) vrp[grpid] = LB[0];
}
```

In Fig. 5 there is an example of the memory access by work-items in this kernel with a work-groups' size of 8. This kernel is based on [16]. In this kernel, each work-item loads two elements of each vector and computes their product and a first reduction, then it stores the result in LB (local memory). Then the reduction is performed by completely unrolling the loop. The first half of the work-items in the work-group performs a first reduction within LB in the first if line, then the first quarter do the same in the second if line and so on, until the position 0 of LB has the final reduction of the block. For the sake of space, only the unrolling of a work-group of size 8 is shown and some barriers were omitted, however the code tested also includes the unrolling of work-groups of size: 32, 64, 128, 256, 512 and 1024.

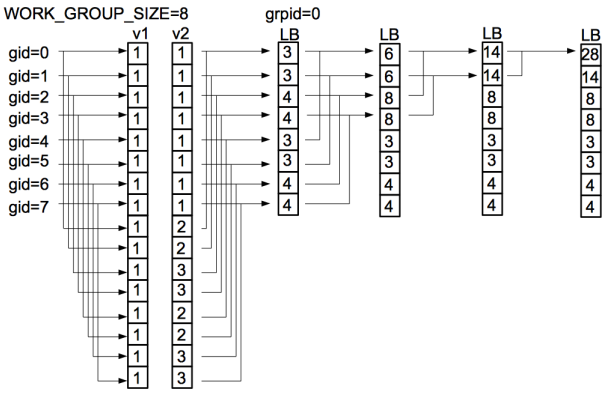

Figure 5 Array accesses by work-items in kernel $dot0$. This figure contains the example of the reduction step of a local block (LB) which work-group size is set to 8.

The second kernel of the DOT function is:

```
__kernel void dot1(__global float *v1,
                   __global float *v2,
                   __global float *vrp)
{
  unsigned int gid = get_global_id(0);
  unsigned int lid = get_local_id(0);
  unsigned int i;
  float aux;
  vrp[gid] = v1[gid]*v2[gid];
  barrier(CLK_GLOBAL_MEM_FENCE);
  if (lid=0)
  {
      aux = 0.0f;
      for (i=0; i<CHK; i++)
          aux = aux + vrp[gid+i];
      vrp[gid] = aux;
  }
}
```

In this kernel, each work-item computes a chunk of elements of the array vrp, if $CHK = 64$ and the length of the arrays is 256, then the work-item with $gid = 0$ will compute elements from positions 0 to 63, the work-item with $gid = 1$ will compute elements from positions 64 to 127, and so on, until all the partial results are stored in array vrp (see Fig. 6). Note that this kernel is not using local memory, each work-item reads the value from global memory and stores the result on private memory (aux) until its done with its chunk, then stores the partial sum on global memory.

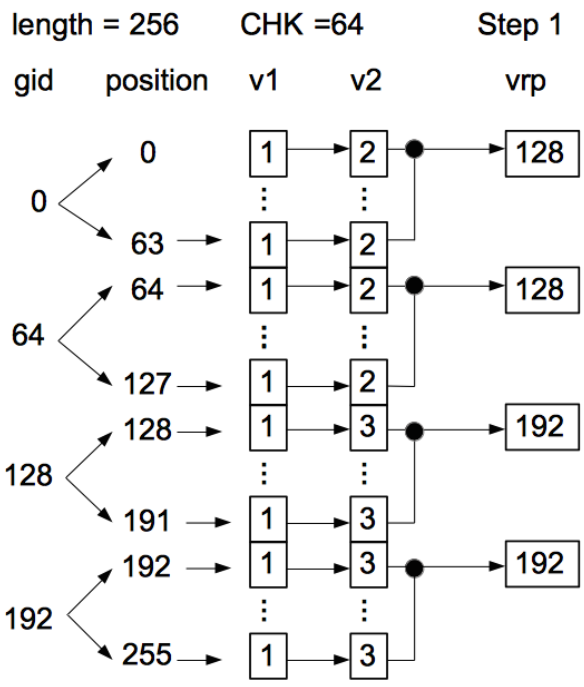

Figure 6 Array accesses by work-items in kernel $dot1$. This figure contains the example of vectors of length 256 and work-group size set to $CHK = 64$.

In both kernels, the total number of work-items (global size) is equal to the size of elements of the vectors, or equal to the nearest multiple of CHK that covers the size of the elements of the vectors, if they are not divisible by the value CHK. In both cases, CHK controls the size of the work-group (local size), as CHK is equal to the size of the work-group.

An auxiliary kernel is needed to do the final reduction of the partial results (see Fig. 6) this kernel is called red and its execution time is considerable lower than those of the kernels $dot0$ and $dot1$. The following reduction could also be performed in CPU because it is a quick operation.

```
__kernel void red(__global float *vrp,
                  __global float *vr)
{
  unsigned int i, gid = get_global_id(0), ngrps = get_num_grps(0);
  float aux;
  if (gid==0) {
      aux = 0.0f;
      for (i=0; i<ngrps; i++)
          aux = aux + vrp[i*CHK];
      vr[0] = aux;
  }
}
```

In this kernel, the work-item with $gid = 0$ realizes the final reduction. From the example of Fig. 6, the work-item 0 will sum the partial results of the vector vrp on positions 0, 64, 128 and 192, and store the result on position 0 of the vector vr.

The SpMV function

In this work, the storage format used to store the sparse matrix is the *Compressed Sparse Row* format (CSR) [9]. As illustration of this format, an example is given next.

Suppose the sparse matrix $\mathbf{A}$ has the following elements:

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 0 & 0 \\ 0 & 3 & 4 & 0 \\ 0 & 0 & 5 & 6 \\ 0 & 7 & 8 & 9 \end{bmatrix} \tag{1}$$

The CSR format stores the sparse matrix $\mathbf{A}$ as:

$$\begin{aligned} \text{AA} &= \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{bmatrix} \\ \text{JA} &= \begin{bmatrix} 1 & 2 & 2 & 3 & 3 & 4 & 2 & 3 & 4 \end{bmatrix} \\ \text{IA} &= \begin{bmatrix} 1 & 3 & 5 & 7 & 10 \end{bmatrix} \end{aligned}$$

where AA is an array that stores the values of the elements of matrix $\mathbf{A}$, JA is an array that stores the values of the columns of each component of the matrix $\mathbf{A}$, and IA stores how many elements has each row of the matrix $\mathbf{A}$.

Therefore the CPU version of the sparse matrix-vector product is:

```
for (i = 0; i < ROWS; i++)
{
    r[i] = 0.0f;
    for (j = IA[i]; j < IA[i+1]; j++)
        r[i] = r[i] + AA[j]*v[JA[j]];
}
```

The first OpenCL kernel of the SpMV function is shown next:

```
__kernel void spmv0(__global float*AA,
                    __global int *JA,
                    __global int *IA,
                    __global float *v,
                    __global float *r)
{
    __local float LB[32];
    unsigned int lid = get_local_id(0);
    unsigned int grpid = get_group_id(0);
    unsigned int indI,indF,i;
    if (grpid<ROWS)
    {
        indI = IA[grpid];
        indF = IA[grpid+1];
        LB[lid] = cero;
```

```
        for (i=(indI+lid); i<indF; i=i+32)
            LB[lid] = LB[lid] + AA[i]*v[JA[i]];
        barrier(CLK_LOCAL_MEM_FENCE);
        if (lid<16) LB[lid] = LB[lid] + LB[lid+16];
        if (lid< 8) LB[lid] = LB[lid] + LB[lid+ 8];
        if (lid< 4) LB[lid] = LB[lid] + LB[lid+ 4];
        if (lid< 2) LB[lid] = LB[lid] + LB[lid+ 2];
        if (lid< 1) LB[lid] = LB[lid] + LB[lid+ 1];
        if (lid< 1) LB[lid] = LB[lid] + LB[lid+ 1];
        if (lid==0) vr[grpid] = LB[0];
    }
}
```

In this kernel, one work-group, of 32 work-items, is assigned to one row of the sparse matrix, then a reduction is done by unrolling the loop for corresponding to 32 elements. The for loop will iterate if the length of the row is larger than 32 elements. The memory accesses of this kernel are shown in Fig. 7. This kernel is based on [17].

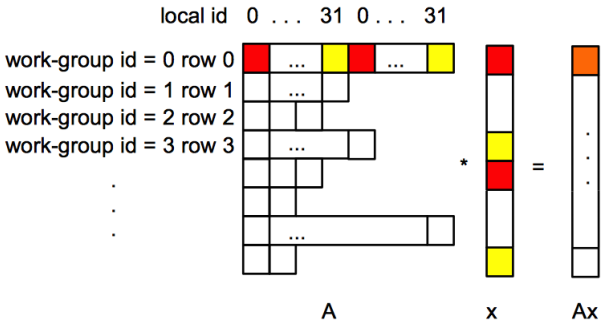

Figure 7 Array accesses by work-items in kernel $spmv0$. This figure contains the example of a sparse matrix with n rows, up to 64 non zero elements per row accessed by work-grups of size $CHK = 32$.

The second OpenCL kernel of the SpMV function is shown next:

```
__kernel void spmv1(__global float*AA,
                    __global int *JA,
                    __global int *IA,
                    __global float *v,
                    __global float *r)
{
    unsigned int gid = get_global_id(0);
    unsigned int indI = IA[gid];
    unsigned int indF = IA[gid+1];
    unsigned int i;
    float aux = 0.0f;
    for (i=indI; i<indF; i++)
      aux = aux + AA[i]*v[JA[i]];
    vr[gid] = aux;
}
```

In this kernel, one row is assigned to one work-item. From the previous example and using four work-items for the matrix $\mathbf{A}$ the access would be as shown in Fig. 8. This is known as the *scalar* kernel [17] because each work-item performs the scalar product of a row's vector by the vector multiplying the matrix.

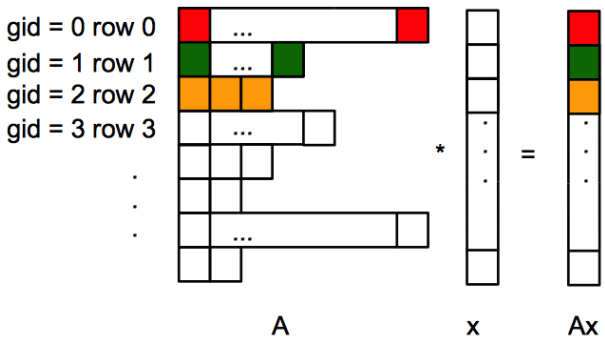

Figure 8 Array accesses by work-items in kernel $spmv1$. This figure contains the example of a sparse matrix with n rows, the non zero elements of each row are accessed by an individual work-item.

Results

It is worth to mention the methodology used to measure execution times. For the CPU routines, the POSIX function *gettimeofday* is used. For measure running times of OpenCL kernels, events profiling is used. Profiling requires to create an event for each en-queued kernel that registers the starting and ending time of the kernel, and the use of synchronizing methods (i.e. *cl_wait*, *cl_finish*) to make to host wait for the kernel's execution and termination. Also, to try to avoid the overhead due to initialization the first executions are discarded previous to the kernel performing the operation being tested. Each kernel was executed 10,000 times to get a good estimated of the arithmetic mean of its execution time.

The AXPY function

Due to the fact that the kernel for AXPY function has not dependence of the size of the work-group (or local dimension) to test its performance various lengths of the vectors are used for computation. The sizes of the vector were: 1 K, 10 K, 100 K, 1 M, 10 M and 50 M of floats. As Fig. 9 shows, the NVIDIA Tesla GPU performed better for vectors of small size, it ran approximately 10 times faster than Intel Xeon Phi coprocessor for the 1 K case. However as the vector size increases both devices reach a point of interception at around 50 K of elements. Finally, Intel Xeon Phi surpass the NVIDIA Tesla performance reaching an acceleration of 2.5 times faster for the 50 M case. The better performance of the NVIDIA GPU for vectors' size under 50 K could be explained due to a larger cost of the overhead initialising the cores of the Intel Xeon Phi coprocessor, the almost horizontal segment between 1 K and 10 K elements for the Intel line hints to a minimum time expended initialising the cores for this kernel's execution.

The DOT function

To test the performance of the DOT function, the size of the work-groups were varied. Using vectors of a fixed size of 2^{20} floats, the executions were done with work-groups size of: 32, 64, 128, 256, 512 and 1024 work-items for the NVIDIA Tesla GPU and the Intel Xeon Phi coprocessor.

From Fig. 10, can be appreciated that the kernel $dot0$ is faster on the NVIDIA Tesla architecture as it makes a better use of the local memory. The graph also shows that 256 is the optimum size for work-groups for the NVIDIA device. The apparently independence showed by the Intel coprocessor when executing kernel $dot0$ could be due to overhead generated by using local memory on this type of architecture.

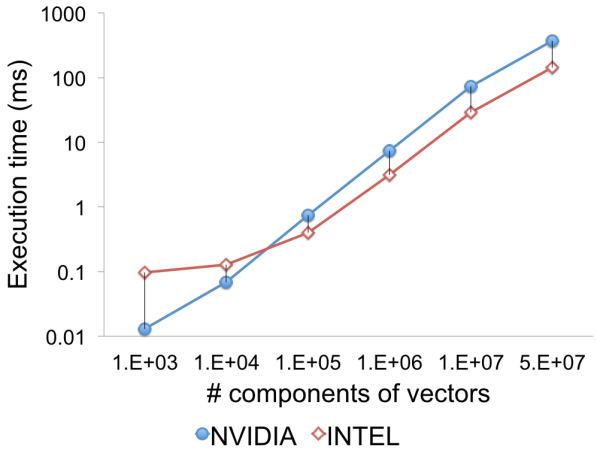

Figure 9 Execution of the Kernel $axpy$ on both devices vs size of vectors.

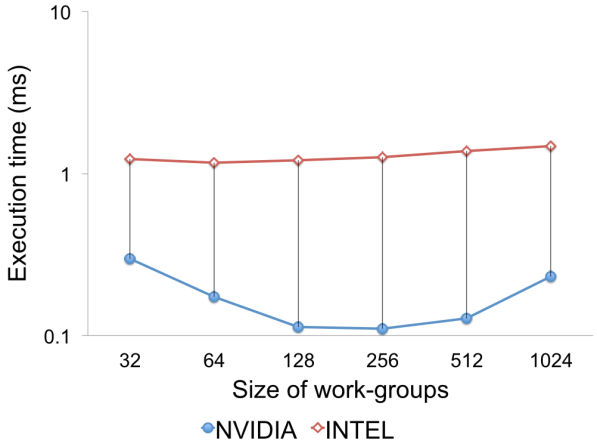

Figure 10 Execution of the Kernel $dot0$ on both devices vs size of work-groups.

On the other hand, Fig. 11 shows that the Intel coprocessor performs faster using global memory only, which agrees with the recommendations on [12] to avoid the use of local memory, and using work-groups of size 64. The figure also confirms the optimum work-groups' size of 256 for the NVIDIA architecture, and shows that the NVIDIA GPU is faster for the reduction of vectors of 2^{20} elements.

Once the optimum versions for the DOT function had been identified for each device, a comparative of these version is done varying the number of componentes of the vectors processed. As can be appreciated on Fig. 12 the NVIDIA GPU performed the reduction operation 10 times faster for the small size vectors than Intel and the gap narrows to 5 for the large size vectors.

The SpMV function

Testing the kernel $spmv0$ was similar done as the DOT function, due to the fact that these kernels share dependancy from the size of the work-group. However, the vectors were replaced by a square sparse matrix of 2^{14} rows, with each row having 128 elements, that results in arrays AA and JA of 2^{21} floats and integers respectively; and a vector of 2^{14} elements. In Fig. 13 the graphs are shown.

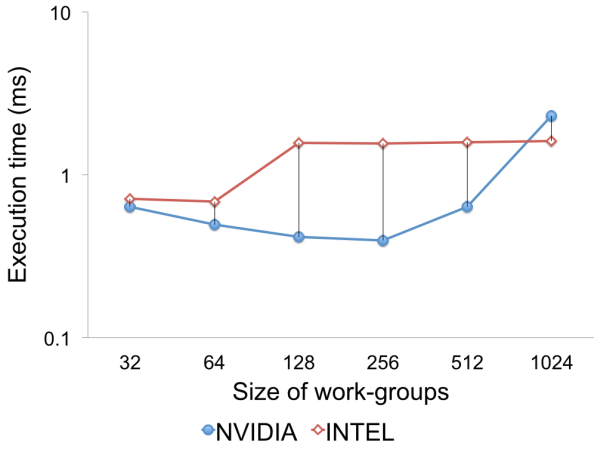

Figure 11 Execution of the Kernel $dot1$ on both devices vs size of work-groups.

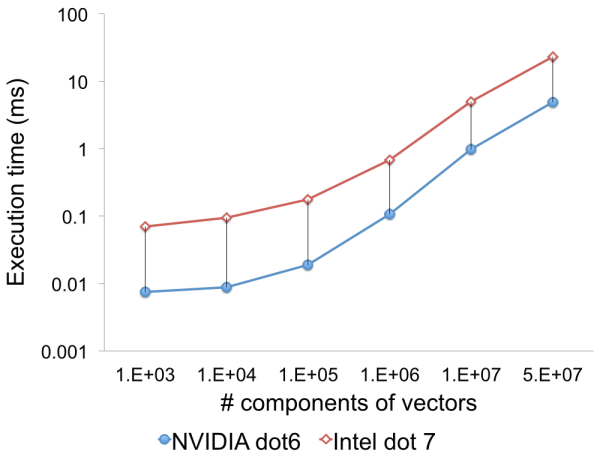

Figure 12 Comparative of the reduction operation on both devices vs size of vectors.

As expected, the $spmv0$ was faster on the NVIDIA Tesla GPU due to its use of local memory in the reduction of the partial sums. However, as the size of the work-group increases, so the total quantity of work-items, creating a lot of idle threads since the maximum number of elements per row is 128. This affected the execution time negatively as it increased with a larger number of threads available to process each row.

To test the kernel $spmv1$ the size of the matrix was varied as the kernel has no dependency on the work-group's size. The matrices are described on TABLE 2.

The results of the $spmv1$ were according to the behaviour seen by the AXPY function. For the 1K case the NVDIA Tesla GPU is approximately 10 times faster than the Intel Xeon Phi coprocessor, until both reach a common performance point with matrices of 100 K of non zero elements. Above that number Intel Xeon Phi coprocessor performs faster than NVIDIA, reaching a speed up of 6 versus the NVIDIA GPU for matrices with 50 M coponents (see Fig 14).

The results verify the impact of using local memory on the NVIDIA architecture, avoid it on the Intel architecture; the effect of the work-group's size where local memory is used; and the effect of the synthetic data's size on the performance of each device.

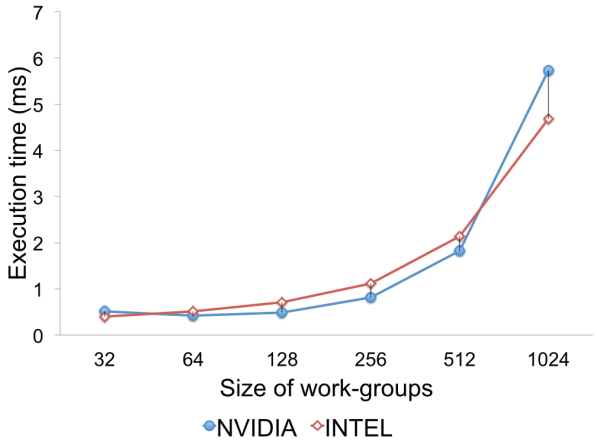

Figure 13 Execution of the $spmv0$ kernel on both devices vs size of work-groups.

h!

Table 2 Matrices to test kernel $spmv1$.

Matrix	Rows	elements per row	Non zero elements
M001	100	10	1 K
M002	1 K	10	10 K
M003	10 K	10	100 K
M004	10 K	100	1 M
M005	10 K	1 K	10 M
M006	10 K	5 K	50 M

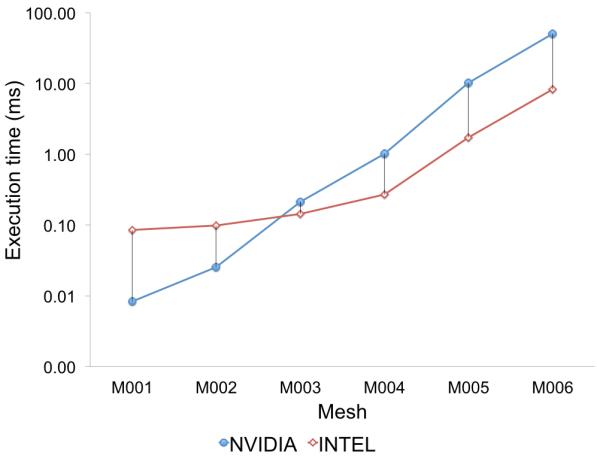

Figure 14 Execution of the $spmv1$ kernel on both devices vs different matrices.

Conclusions

Clearly, the optimization of kernels for each device's architecture pays off, as each kernel has a speed up on the target device. For the AXPY function results showed that smaller vectors' size benefits the NVIDIA architecture as it performs the operation 10 times faster than the Intel architecture at vector's size of 1K and that larger vectors' size benefits the Intel architecture as it performs the operation 2.5 times faster than NVIDIA at vectors' size of 50 M (see Fig. 9). The better performance

of the NVIDIA GPU for vectors' size under 50 K could be due to a larger cost of initialising overhead of the Intel Xeon Phi coprocessor's cores.

For the DOT function the use of local memory benefits the NVIDIA architecture as it performed the operation faster with work-groups's size from 32 to 1024 and using vectors of 2^{20} elements; achieving its peak performance with work-groups' size of 256 (see Fig. 10). On the other hand, the Intel architecture performs better when using only global memory in the same work-groups' size range, achieving its peak performance with a work-group size of 64 and using vectors of 2^{20} elements (see Fig. 11). The results also shows that NVIDIA GPU performs the reduction operation 10 times faster that NVIDIA for small size vectors and 5 times faster for large size vectors than the Intel coprocessor.

For the SpMV function the NVIDIA the use of local memory benefits the NVIDIA architecture as it performs the operation 15 times faster than Intel with work-groups' size of 4 and using a sparse matrix of 2^{14} rows, 2^{21} non zero elements and a vector of 2^{14} elements (see Fig. 13). On the other hand, without using local memory, Intel architecture performs faster than NVIDIA for larger matrices' sizes reaching a speedup of 6 for a matrix of 50 M non zero elements meanwhile the NVIDIA architecture performs the operation 10 times faster than Intel with a matrix of 1 K non zero elements.

The versions of the DOT and SpMV functions that are not using local memory show a vectors's size where both devices reach the same performance. This is an interesting point since energetic considerations could be the decisive factor to select one of these devices to perform the same computational task.

Competing interests
The authors declare that they have no competing interests.

Acknowledgements
This work has been supported by FEDER funds and Xunta de Galicia under contract GRC 2014/008, and by Spanish Government (MCYT) under project TEC2010-17320 and TIN-2013-41129-P.

References

1. Top 500. http://www.top500.org
2. https://www.khronos.org/opencl/
3. Coronado, E., Garcia, A., Indalecio, G., Seoane, N.: Implementation of numerical methods for nanoscaled semiconductor device simulation using opencl. Spanish Conference on Electron Devices (2015)
4. Seoane, N., Garcia-Loureiro, A.J.: Study of parallel numerical methods for semiconductor device simulation. International Journal of Numerical Modelling: Electronic Networks, Devices and Fields **2006**(19), 15–32 (2005)
5. Garcia-Loureiro, A.J., Lopez-Gonzalez, J.M., Fernandez-Pena, A.T., Prat-Viñas, L.L.: Numerical analysis of abrupt heterojunction bipolar transitors. International Journal of Numerical Modelling: Electronic Networks, Devices and Fields **11**(4), 221–229 (1998)
6. Garcia-Loureiro, A.J., Kalna, K., Asenov, A.: 3D parallel simulations of fluctuation effects in pHEMTs. Journal of Computational Electronics, 369–373
7. Indalecio, G., Garcia-Loureiro, A.J., Aldegunde, M., Kalna, K.: 3D simulation study of work-function variability in a 25 nm metal-gate finfet with curved geometry using voronoi grains. Proc. 17th Int. Conf. Simul. Semicond. Proc. Devices (SISPAD), 149–152
8. Rodriguez, J.A., Otero, P., Vetter, M., Andreu, J., Comesana, E., Garcia-Loureiro, A.J.: Simulation of the effect of p-layer properties on the electrical behaviour of a-si: H thin film solar cells. Electron Devices Conference (CDE) (2011)
9. Saad, Y.: Iterative Methods for Sparse Linear Systems, 2nd edn. Philadelphia: Society for Industrial and Applied Mathematics, Philadelphia, PA (2003)
10. NVIDA Tesla GFLOPS. http://www.techpowerup.com/gpudb/1538/tesla-s2050.html
11. Intel Xeon GFLOPS. http://www.dellhpcsolutions.com/dellhpcsolutions/static/xeonphi.html
12. Intel Xeon Phi Coprocessor OpenCL Programing Guide. https://software.intel.com/en-us/articles/opencl-design-and-programming-guide-for-the-intel-xeon-phi-coprocessor
13. Scarpino, M.: OpenCL in Action: How to Accelerate Graphics and Computation, 2nd edn. Manning Publications Co., Shelter Island, NY (2012)
14. Gaster, B., Howes, L., Kaeli, D.R., Mistry, P., Schaa, D.: Heterogeneous Computing with OpenCL. MORGAN KAUFMANN, Waltham, MA (2012)
15. Banger, R., Bhattacharyya, K.: OpenCL Programming by Example. Packt Publishing Ltd., Birmingham, UK (2013)
16. Mark Harris: Optimizing Parallel Reduction In CUDA. http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pd
17. Bell, N., Garland, M.: Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, Dec. 2008, 149–152 (2008)
18. http://ark.intel.com/es-es/products/75799/Intel-Xeon-Phi-Coprocessor-7120P-16GB-1_238-GHz-61-core