Kernel Density Estimation in Accelerators

Implementation and Performance Evaluation

Unai Lopez-Novoa · Alexander Mendiburu · Jose Miguel-Alonso

Received: date / Accepted: date

Abstract Kernel Density Estimation (KDE) is a popular technique used to estimate the probability density function of a random variable. KDE is considered a fundamental data smoothing algorithm, and it is a common building block in many scientific applications. In a previous work we presented S-KDE, an efficient algorithmic approach to compute KDE that outperformed other state-of-the-art implementations, providing accurate results in much reduced execution times. Its parallel implementation targeted multi and many-core processors. In this work we present an OpenCL implementation of S-KDE, targeting modern accelerators in a portable way. We also analyze the performance of this implementation on three accelerators from different manufacturers, to find out to what extent our code exploits the performance offered by those devices.

Keywords Kernel Density Estimation · Performance Analysis · OpenCL · Many-core Processors · GPGPU

1 Introduction

Kernel Density Estimation (KDE) is a popular statistical technique to estimate the probability density function of a random variable with unknown characteristics [23]. It is considered a fundamental data smoothing problem

This work has been partially supported by the Saiotek and Research Groups 2013-2018 (IT-609-13) programs (Basque Government), TIN2013-41272P (Ministry of Science and Technology), COMBIOMED-RD07/0067/0003 network in computational biomedicine (Carlos III Health Institute) and by the NICaiA Project PIRSES-GA-2009-247619 (European Commission).

U. Lopez-Novoa, A. Mendiburu, J. Miguel-Alonso
Intelligent Systems Group, Department of Computer Architecture and Technology, University of the Basque Country UPV/EHU
P. Manuel Lardizabal 1, 20018 San Sebastian, Gipuzkoa, Spain
E-mail: {unai.lopez,alexander.mendiburu,j.miguel}@ehu.es

in statistics and an alternative to other density estimation techniques such as the histogram, that relies on a simple binning. KDE is used in a wide variety of research areas, such as climatology for environmental model evaluation [16], computer vision for image segmentation and tracking [5] or biometry to estimate the effectiveness of a medical treatment[26].

Some of the problems that use KDE codes as building block usually require processing large datasets, which translates into long execution times. The literature shows different approaches to computing KDE. Given the complexity of the algorithm, a trade-off must be found between accuracy and execution time [22]. In a previous work, we introduced a novel algorithm to compute KDE whose complexity is lower than that of state-of-the-art KDE implementations, providing accurate results with shorter execution times. We called this algorithm S-KDE [15]. As modern scientific codes run in multi-core processors, we implemented and tested an OpenMP implementation of S-KDE that exploited the parallel capabilities of current CPUs, and many-core co-processors such as the Intel Xeon Phi [7]. The combined effect of the novel algorithmic approach and the exploitation of parallel processing resulted in impressive reductions in execution times.

We cannot ignore, though, that many of the computing platforms used nowadays, and those expected to be used in a near future, will integrate other classes of accelerator devices, not only the Xeon Phi [18]. The spectrum of devices is wide and includes platforms such as Graphics Processing Units (GPUs), FPGAs and other classes of many-cores. In order to make S-KDE available to a larger community, we decided to produce a new version of the code targeting the wider possible set of accelerators, being OpenCL the most logical choice of programming environment.

Two are the main contributions of this paper. First, the description of the porting of S-KDE to OpenCL. Redesigning a code for accelerators usually requires major changes due to the massive data parallel processing model they are aimed for [11], and not every application fits into it. Second, we evaluate the performance of our code when running it in three state-of-the-art accelerators: an AMD GPU, a NVIDIA GPU and an Intel Xeon Phi co-processor. We rely on some popular performance models and benchmark suites to characterize the devices, and provide some insights about how well our code exploits the performance achievable from each accelerator.

The remainder of this paper is organized as follows. We provide an overview of the state-of-the-art accelerator devices in Section 2, and describe briefly the fundamentals of our S-KDE approach in Section 3. We present the OpenCL implementation of KDE in Section 4, and conduct a performance analysis over it in Section 5. Finally, we summarize conclusions and future lines of work in Section 6.

2 Accelerator devices

Current supercomputers and data processing facilities are being built around hybrid compute nodes that include accelerator devices. The current landscape of devices includes reconfigurable circuits (e.g. FPGAs), discrete co-processors (e.g GPUs), hybrid chips (e.g. AMD HSA systems) or low power consumption systems (e.g. ARM or Intel Atom based systems)[14]. In this work we are going to focus on the most popular classes of accelerators, GPUs and Intel's Xeon Phi, due to their wide presence in HPC systems and for the extensive body of literature and ecosystem of tools around them.

GPUs are hardware devices designed to make efficient image processing. They are composed of hundreds of SIMD cores, capable of handling thousands of active threads, with lightweight context switching [11]. Since their adoption as general purpose coprocessors (coining the term GPGPU, from general-purpose processing on GPUs), they have been enhanced with features such as dedicated double-precision units or large cache hierarchies that make them ready to run efficiently a wide variety of HPC workloads. GPUs can be found as discrete coprocessors, connected to a host processor through PCI-Express, or integrated in the same die with other type of processing cores, as in the AMD APU architectures. In this work we will use two different discrete GPUs connected through PCI-Express.

The Xeon Phi is a many-core processor presented by Intel in 2012. It holds up to 61 x86 cores, 16 GB of dedicated memory and it is connected to a host system through PCI-Express. Compared to the cores in Intel multi-core CPUs, Xeon Phi cores make in-order processing and hold several differences in the instruction set, such as using AVX-512 for vector computations. Current Xeon Phi devices present a theoretical peak performance of 1 TFLOP/s in double precision, and support a wide set of development frameworks, such as MPI, OpenMP, OpenCL or Cilk [7].

3 Kernel Density Estimation

KDE has been applied since the 80's as a density estimation technique in different environments [22]. It creates smooth density estimations, in contrast with other techniques, such as the histogram. Intuitively, given an evaluation landscape and a dataset of samples, KDE places in the landscape a "bump" around each sample, aggregating the effect of those bumps to create the estimated Probability Density Function (PDF). An example for a one-dimensional case is depicted in Figure 1, where a density estimate is created for a dataset with three samples. A kernel (a red "bump") is placed over each sample, and then the influence of all of them is summed creating the black thick line, which is the estimated PDF [23].

The resulting density estimation is continuous, but most KDE implementations provide it as a set of discrete values. The user defines the boundaries of the landscape and the separation between the points where the PDF will

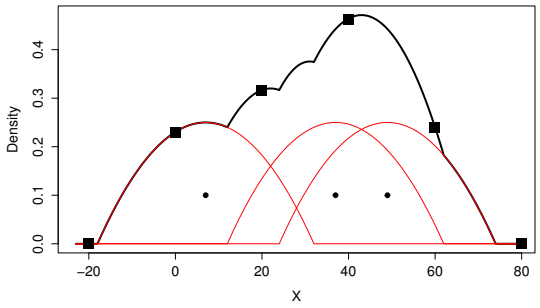

Fig. 1 Example of KDE for 1D data

be evaluated (an array of per-dimension steps). Therefore, the output is actually a (discrete) evaluation grid, an array of evaluation points. In the example shown in Figure 1, the 1D evaluation space spans from -20 to 80, and the evaluation step is 20. Thus, the estimated function will be represented as a vector containing the densities in the evaluation points $-20, 0, 20, 40, 60$ and 80.

The most common way to compute KDE is to traverse every point in the evaluation grid, and compute and add, for each of them, the density influenced by each and every sample. This approach is completely parallelizable using a data parallel approach, but in many cases implies a vast number of useless computations. This is due to the fact that a sample affects only a portion of the evaluation space, a set of points around its position. The size of this influence area depends on the kernel of choice (the shape of the "bump") and other parameters. Thus, a more efficient approach is to define the influence area of a sample as a set of evaluation points, and then traverse just the evaluation points inside that area. The first approach has an $O(k_d mn)$ computational complexity, being k_d a dimensionality constant, m the number of evaluation points (the size of the evaluation grid), and n the number of samples. The second approach has an $O(k_d np)$ complexity, where k_d is a dimensionality constant, n the number of samples and p the number of evaluation points in the influence area of a sample. We must take into account that usually p is *much smaller* than m. In this work we will implement the second approach. We call it S-KDE (from sample-wise KDE), and include some additional features to further avoid unneeded computations.

The KDE literature includes different proposals for kernel functions. Depending on the chosen one, the technique to confine the influence area of a sample will be different. In this work we use an Epanechnikov kernel and a technique based on the eigenvalues of the covariance matrix of the dataset to calculate a rectangular shaped box that delimits the influence area of a sample [6]. We will refer to this rectangle as the *bounding box* of a sample. In addition, we apply a technique called *Chop & Crop* that minimizes the size of the bounding box by removing evaluation points not belonging to the influence area of the kernel in spaces of dimensionality three or higher. It works

by first reducing the d-dimensional bounding box to a set of 2D slices, and then cropping the slice to the minimum squared box. This two-step process is represented in Figures 2 and 3 respectively. The interested reader is referred to [15] for a detailed explanation of the S-KDE algorithm and its implementation for multi-core CPUs.

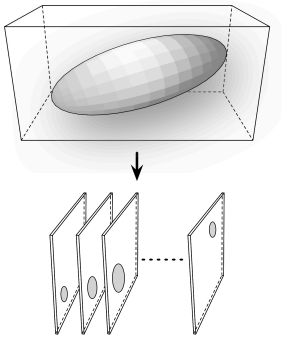

Fig. 2 Chopping a 3D bounding box into 2D slices

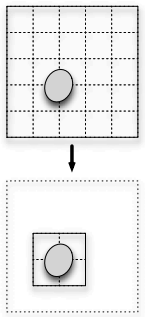

Fig. 3 Cropping a 2D slice to obtain a minimum-size bounding rectangle

We can provide some example figures to illustrate the efficiency of S-KDE. We will assume a 3D dataset with 500k samples, and an evaluation space (grid) with 194.81 million evaluation points. Using the traditional KDE approach that traverses every evaluation point of the grid would lead to $9.74 * 10^{13}$ sample-evaluation point operations. In contrast, a rectangular 3D bounding box around each sample in the mentioned scenario contains on average 102461 points, and using the sample-wise KDE approach would require $5.12 * 10^{10}$ computations. On top of this, if we apply the *Chop & Crop* technique, the number of evaluation points per bounding box is reduced to 53511 on average, and the resulting total number of computations is $2.67 * 10^{10}$. Thus, S-KDE improves KDE efficiency by several orders of magnitude. We can go even further, exploiting massively parallel accelerators to run S-KDE.

4 An OpenCL KDE implementation for accelerators

In this section we describe the process followed to adapt S-KDE to accelerators. Our aim has been to create an algorithm fitting into the data-parallel computation model that most accelerators use, together with an implementation portable across the spectrum of existing hardware devices. Currently, the only two frameworks that provide portability in accelerators are Microsoft DirectCompute and OpenCL, a standard proposed by the Khronos Group [9]. Given that the former is only available for Microsoft Windows platforms, we chose OpenCL.

4.1 OpenCL in a nutshell

An OpenCL application consists of a host part and a device part, which is called a *kernel* (not to be confused with the kernel functions used for density estimation). The host part orchestrates data movements from host to device and vice-versa, and manages the execution of kernels. The device is able to run simultaneously multiple threads or *work-items*, all of them running the same kernel code. Work-items are arranged in groups called *work-groups*, which may have a 1D, 2D or 3D structure; the developer chooses the shape and size of the work-groups.

An OpenCL *platform* is a collection of devices managed by a single host. OpenCL defines a device model in which the device is composed of a set of *processing elements*, arranged in *compute units*. At run time, the OpenCL framework assigns each work-group to a compute unit for its execution. Internally, work-items are mapped to processing elements.

Regarding memory, there is host memory and per-device global memory. The latter is accessible by all threads running in the device. Additionally, there is local memory shared by all treads in the work-group. Finally, each thread has its own, small, private memory and registers. The host code is in charge of moving data from host to device memory (and vice-versa), and threads can move data from the device's global memory to other memory zones.

All these abstractions enable the OpenCL framework to launch any data-parallel application (kernel) over any device, given a mapping between the device's characteristics and the OpenCL model, and also given that hardware requirements are fulfilled (for example, a certain amount of memory per thread is necessary, and while some devices can provide it, others could be more limited). Further information about OpenCL can be found in [17].

4.2 Implementing S-KDE with OpenCL

The starting point for the OpenCL code is the serial implementation of S-KDE. The steps that this code follows have been depicted in Figure 4. Note that steps 4 and 6 have been surrounded with parentheses, as they are operations required in the OpenCL code but not in the serial one.

Step 1 is the initialization, where the evaluation grid is set to zeros and the size of the bounding box is computed; this is a "generic" bounding box, that must be customized per sample, in order to deal with the discrete and bounded nature of the evaluation space. Then, the code traverses the samples of the dataset in an iterative way.

For each sample, the bounding box is first fitted to the grid, and then the chopping is applied (Step 2). This way, no matter the dimensionality of the problem, it is reduced to a computation of a series of bi-dimensional slices. Then, each slice of the bounding box is processed.

For each slice, cropping is applied to reduce it to its minimum size (Step 3). Once the size and coordinates of the slice are defined, its evaluation points are

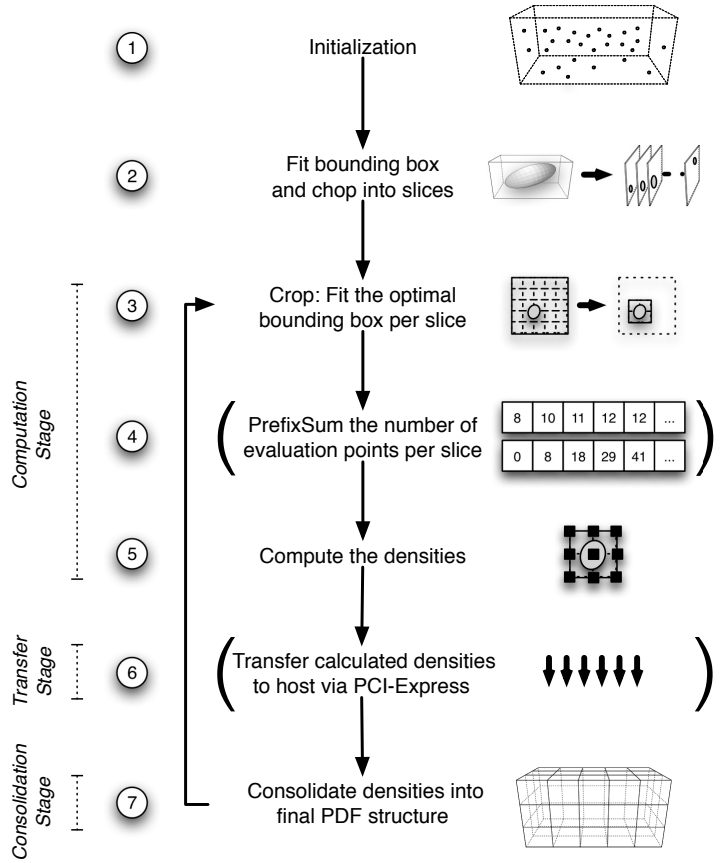

Fig. 4 Workflow of KDE implementation

traversed. For each point within the slice, its distance to the sample and the density that the sample influences on the point are computed (Step 5). Finally, all partial densities affecting a point are accumulated in the corresponding position of the evaluation grid (Step 7).

The described serial algorithm presents clear opportunities for parallelization, but it also poses some challenges for its adaption to the accelerator model due to its limited data reuse and the very low compute to memory access ratio. That being said, a major rework has been done to adapt it to the OpenCL model, which has required some algorithm re-structuring as well as the inclusion of additional support operations.

The main opportunity for adapting the algorithm to a data-parallel model comes from the fact that the bounding boxes around each of the samples can be processed simultaneously, without any kind of dependency. Therefore, we can have as many parallel threads as samples in the dataset. However, all these bounding boxes must be aggregated into a common landscape matrix

in which the final PDF grid is computed, and as influence areas of samples (and, therefore, bounding boxes) may overlap, the accumulation step must be somehow synchronized to avoid memory write collisions. The way we have addressed this issue is explained later.

The OpenCL code has been structured as depicted in Figure 4. It includes new steps, as well as modifications to some of those described for the serial code.

1. *Initialization:* In a first step, the entire sample dataset is copied into the accelerator, along with the required support structures. We assume that all the dataset fits in the memory of the accelerator (note that this is not the evaluation grid). In this step we also compute the size of the generic bounding box. This is host code (executed in the CPU).
2. *Box fit and Chop:* For every sample, its bounding box is fitted to the grid and chopping is applied. At this point, the problem has been reduced to a collection of 2D slices. This is implemented as an OpenCL kernel (executed in the accelerator in a data parallel way).
3. *Crop:* Cropping is applied to reduce the number of evaluation points in each slice. Additional information about each slice is computed, such as its coordinates in the evaluation space and the number of evaluation points it contains. This is an OpenCL kernel.
4. *PrefixSum:* A PrefixSum is applied to the vector that contains the number of evaluation points per slice. This is a support computation required by the next kernel for its threads to make ordered stores. We have used the OpenCL implementation of PrefixSum available in the SHOC benchmark suite [4].
5. *Density Computation:* Each thread calculates the influence created by a sample on an evaluation point of a given slice. The resulting densities are stored in an auxiliary vector and not consolidated into the global PDF structure. This is an OpenCL kernel.
6. *Densities Transfer:* The resulting vector of partial densities is transferred through PCI-Express from the accelerator to host memory. This is managed by the host.
7. *Consolidation:* The host reads the vector of partial densities and accumulates them into the evaluation space. This is host code.

As explained before, a critical step of S-KDE is the consolidation of partial densities into the global landscape (Step 7). If done in parallel without the proper synchronization mechanisms, results can be invalid, because the influence areas of samples overlap and, thus, threads may incur in memory write collisions. To avoid this issue, the current OpenCL implementation of S-KDE leaves this task to be performed by the host CPU in a serial way. This is pragmatic because the host has the whole output structure in main memory, and the serialized access guarantees the absence of memory write collisions.

We tried alternative approaches to run the consolidation phase in the accelerator, but they required either some sorting of partial results (an expensive operation that resulted in even longer execution times) or the use of atomic

adds (an operation not supported in OpenCL for double precision floats [9], although some devices have specific extensions for it). Therefore, we have kept the consolidation part in the CPU.

As a side note, the presented workflow is intended for KDE problems of dimensionality three or higher. However, our implementation targets as well two dimensional spaces using the same workflow but without applying the Chop & Crop technique – because it is not needed. In this case, the workflow is exactly the same but without the second and third steps.

In terms of the data structures used in the OpenCL code, it should be clear that the threads running in the accelerator have access to the sample dataset and to auxiliary structures containing partial, non-consolidated densities. The final density matrix is managed exclusively by the CPU. The main program is iterative after the initialization step. Each iteration consists of processing a "chunk" of the problem, which is nothing more than a subset of the samples. This is done in a data-parallel fashion, using a chain of OpenCL kernels. The per-chunk intermediate results are stored in the accelerator and, later, transferred to the CPU for consolidation.

The iterative nature of the code has two main advantages. The first one is the ability to deal with accelerators with different memory sizes, that in many cases are not capable of holding the complete output matrix. This makes our code limited by the RAM managed by the CPU, but not by the device's memory, provided that the selected chunk size uses intermediate data structures that fit into the device. The second advantage is that it opens the possibility of working in a pipelined fashion: while the device computes a chunk, the CPU can be consolidating the results of the previous one. We will explore this possibility later in Section 5.3.

An important parameter of our program is the chunk size, or number of samples to process in each iteration of the algorithm. This size must be defined in such a way that the intermediate results from an iteration fit into a data object in the memory of the device, before being transfered to the CPU. Therefore, the chunk size has to consider characteristics of the device (maximum allocatable size) and size (number of points) of each bounding box around a sample. Note that the maximum allocatable size can be smaller than the device's global RAM, and that some space may be already allocated to other required data structures. The chunk size is rounded down to the closest power of two, to better match the work-group sizes managed by the devices.

For example, in a 3D dataset with 500k samples and an evaluation space with 194 million points, each per-sample bounding box could have up to 106609 points or 832.8 kB; this is problem-dependent, and computed at the initialization phase. In a device with 256 MB of maximum object size, our heuristic would assign a chunk size of 256 samples.

5 Performance analysis

This section presents a performance analysis of the KDE implementation described above on three different accelerators. The code was designed for portability, so that it can run, unmodified, in any modern co-processor supporting OpenCL. We will first present the characteristics of the platforms and datasets used in the experiments, and then carry out a performance analysis in top-down manner, i.e. getting first global performance measures and, afterwards, digging into details.

5.1 Settings used in the experiments

Our experiments have been conducted with three accelerator devices: an AMD Radeon HD 6950 GPU, a NVIDIA GTX 650 GPU and an Intel Xeon Phi 3120A Coprocessor, whose main features are summarized in Table 1. In addition, our code has been limited to the API features of OpenCL v1.1. Even though some of these devices support OpenCL 1.2 or 2.0, version 1.1 is the most supported one in currently available processors, including the Xeon Phi, GPUs, FPGAs and ARM-based systems.

	Radeon HD 6950	**GTX 650**	**Xeon Phi 3120A**
Architecture	Cayman	Kepler	MIC
Cores	1408	384	57
Core Clock	800 Mhz	1.05 Ghz	1.1 Ghz
Memory	2 GB GDDR5	1 GB GDDR5	6 GB GDDR5
DP Performance[1]	563 GFLOP/s	67 GFLOP/s	1 TFLOP/s
Max. Allocatable Size	445.5 MB	255.8 MB	1435.2 MB
Host CPU	Intel Core i5-2400S	Intel Core i7-3820	Intel Core i7-3820
OpenCL SDK	AMD APP v2.9.1	CUDA v6.0.37	Intel OpenCL v3.2.1

Table 1 Hardware features of the accelerators used in the experiments

As a reference, we also run in some experiments the serial implementation S-KDE algorithm (including Chop & Crop) compiled with gcc v4.7.2 and executed in a Intel Core i7-3820 (3.60 Ghz clock frequency).

As explained in Section 3, the complexity (or problem size) of a KDE execution depends mainly on the number of samples in the dataset and on the size of the evaluation space (determined by its boundaries and step size). The latter will also determine the size of the bounding box that limits the influence area of a kernel. In this work we have performed several tests varying the size of the evaluation space, for two different 3D datasets. We have fixed the boundaries of the evaluation space and modified the step size to increase/decrease the number of evaluation points, see Table 2. The datasets have been created synthetically, sampling a multivariate normal distribution. The first one contains 500k samples, and the second one contains 1M samples.

[1] Theoretical peak performance in double precision, as declared by the manufacturer

Finally, there is a parameter in every KDE computation that must be taken into account, and that has not been mentioned in Section 3: the bandwidth or smoothing parameter. This value modifies the smoothness and size of the kernel and, therefore, the number of evaluation points inside a bounding box. An exploration for the choice of the right bandwidth value is out of the scope of this work, and we have selected it using the heuristics detailed in [23].

Dim X	Dim Y	Dim Z	Total
110	220	322	7792400
110	440	322	15584800
220	440	322	31169600
220	440	805	77924000
220	1100	805	194810000

Table 2 Size of the different evaluation spaces (number of evaluation points in the grid) used

5.2 Initial assessment

To get an initial assessment of the performance of our OpenCL S-KDE, we compare its total execution time against that of the serial program, for the three target devices. Results are shown in Figures 5 and 6 for the dataset of 500k and 1M samples respectively.

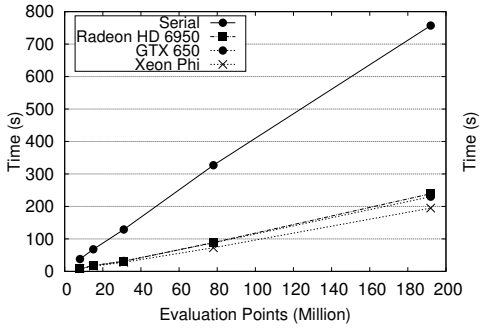

Fig. 5 Comparison of execution times for dataset 500k

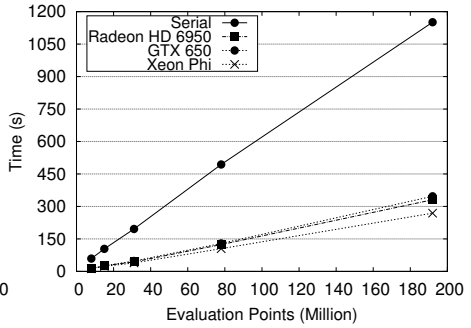

Fig. 6 Comparison of execution times for dataset 1M

The first conclusion is that the OpenCL code runs significantly faster than the serial code. Speed-ups obtained with the largest problems are 3.47x, 3.31x and 4.27x for the Radeon, the GTX and the Phi respectively. These results, being good, are not as impressive as those reported for other HPC applications implemented in accelerators [13]. We can be (partially) satisfied because we

make use of the extra muscle provided by the accelerator, but we also want to explore if we could do better.

After this black-box assessment, we try to understand more in detail the limits and bottlenecks of the code when running in the accelerated platforms. In Figures 7 and 8 we show the accumulated time spent in each of the three stages depicted in Figure 4: Computation (steps 3, 4 and 5, executed in the accelerator), Transfer (step 6, moving data from the accelerator to the CPU) and Consolidation (step 7, executed in the CPU). It is to be highlighted that the most expensive stage in all cases is the non-accelerated part of the code: the Consolidation of partial results in the global density matrix, carried out by the CPU to avoid memory write collisions.

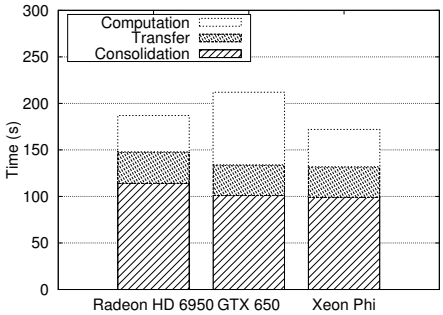

Fig. 7 Dissected execution time of OpenCL S-KDE. 500k dataset and 194M evaluation points

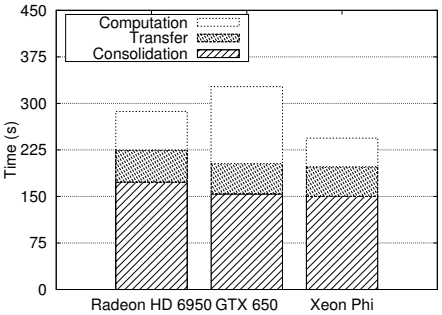

Fig. 8 Dissected execution time of OpenCL S-KDE. 1M dataset and 194M evaluation points

If we focus on each stage separately, we can observe the effects in the execution time of the different elements participating in the computation:

- The time required for the Computation stage shows how the Intel Xeon Phi co-processor is the fastest of the tested accelerators, while the NVIDIA GTX is the slowest.
- The time used for accelerator-to-CPU transfers is approximately the same in the three tested platforms. This is to be expected, as all of them use the same PCI-Express interconnect.
- For the Consolidation stage, the i7 used in the Phi and GTX platforms is slightly faster than the i5 used in the Radeon platform.

Therefore, the good comparative results of the Phi platform comes from a combination of a fast CPU and a fast co-processor.

The main conclusions obtained from these experiments is that we have been partially successful with our OpenCL implementation of S-KDE: the program is faster than the serial version, but not as fast as we would like. We dig further inside the different parts of the code in order to understand what is limiting its performance.

5.2.1 Analyzing compute efficiency

In this subsection we focus on the Computation stage: we want to understand how the OpenCL kernels use the capabilities of the accelerators, and to discover if we are exploiting them in an effective way. To do so, we have relied on the popular *roofline model*, presented by Williams et al. [27] in 2008. It provides a way to visually describe the features of a machine and a program. In particular, it is a diagram with two axes: the *Operational Intensity* of an application (FLOP/Byte) in the X-Axis and the *Attainable GFLOP/s* in the Y-Axis. An application will be positioned somewhere in the X-Axis depending on its operational intensity, and the maximum attainable performance will be given by the roof of the machine.

Even though the roofline model was originally presented for multi-core processors, several papers have extended it to characterize accelerators and massively parallel applications. In [8] GPURoofline is proposed, an adaption of the roofline model for NVIDIA and AMD GPUs that takes into consideration GPU-specific features. In [10] Kim et al. use the roofline model to explore the scalability of a code for electromagnetic field simulations in a NVIDIA GPU. In [3] Cramer et al. use the roofline to characterize the Intel Xeon Phi. In [25] Wang et al. use the roofline model to characterize a GPU and an Intel Xeon Phi, and the scalability of an OpenACC code in them. However, none of these works presents a generic way to build a roofline model for accelerator devices. This is what we do in this section using a method applicable to any OpenCL-capable device.

	DP Performance (GFLOP/s)	Off-chip Bandwidth (GB/s)
Radeon HD 6950	556.02	130.09
GTX 650	36.25	66.39
Xeon Phi 3120A	964.48	94.36

Table 3 OpenCL Benchmarking Results

To characterize the device we need to measure the maximum attainable performance in terms of GFLOP/s and the maximum bandwidth to the off-chip memory in GB/s. These values can be retrieved using a benchmark suite that stresses the devices and provides the effective peak values, which are usually lower than the theoretical ones advertised by the manufacturer. In particular, we use the tests from the ClPeak benchmark suite[2]. Relevant results for the three target accelerators have been listed in Table 3. The roofline plot is computed as *Min(Bandwidth * Operational Intensity, GFLOP/s)*.

The characterization of the application is done kernel by kernel. Each kernel is parsed using the LLVM compiler [12] to generate its intermediate representation, which we parse to count the number of floating point operations and memory accesses.

[2] https://github.com/krrishnarraj/clpeak

Figures 9, 10 and 11 show the roofline plots for the AMD Radeon, NVIDIA GTX and Xeon Phi respectively. Each figure shows the roofline of the machine as a line, the position of the Density Computation kernel (square tick), and the position of the whole Computation stage (without data transfer, cross tick). We can see how those ticks overlap, because the accelerators use most of their time to compute densities. Other kernels are not depicted in the graph because they are fast and, besides, some of them (i.e., PrefixSum) operate with integer type values and, thus, their operational intensity is zero.

We can see how the NVIDIA GTX accelerator is way below the others in terms of raw performance. We can also observe how our application has a very low operational intensity and, therefore, it is far from reaching the top performance in all cases. This is particularly harmful in powerful and expensive accelerators, such as the Xeon Phi. The low compute-to-memory ratio and low data reuse makes S-KDE a memory bound algorithm – but it would be even worse using the classic, evaluation point-wise approach, or without implementing Chop & Crop.

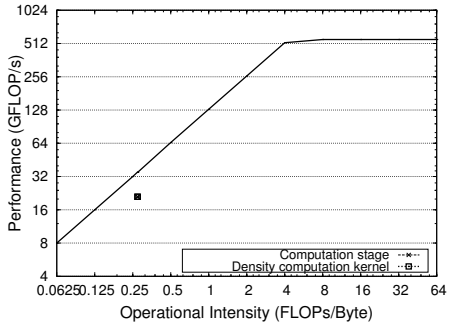

Fig. 9 Roofline of Radeon HD 6950

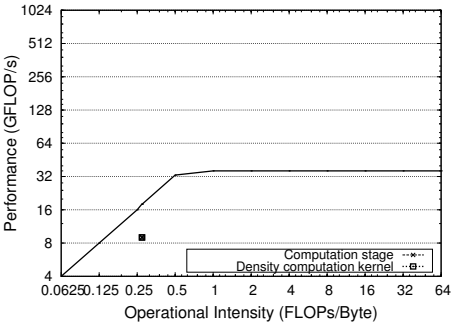

Fig. 10 Roofline of GTX 650

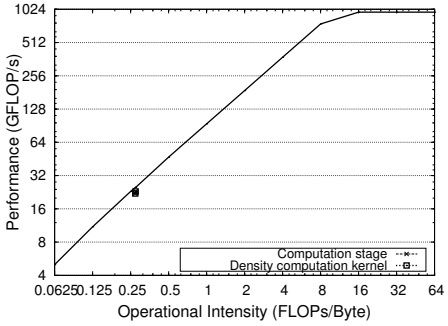

Fig. 11 Roofline of Xeon Phi 3120A

In order to better exploit the accelerators, modifications in the applications would be required to increase its operational intensity: running more (double precision) floating point operations per moved data item. Given the current S-KDE algorithm, we have not discovered any way of doing this.

In addition, we must highlight the choice of an OpenCL parameter that affects performance: the work-group size. As explained in Section 4.1, OpenCL assigns work-groups to Compute Units (CUs) for their execution. Assuming that hardware constraints are fulfilled, an excessively large work-group size might leave CUs unused in the device, and an excessively small one might cause stalls in some architectures such as GPUs. In the design of the OpenCL KDE code we made an exploration on the work-group size to find the one that minimized the execution times. We found 128 threads per work-group to be the best size for the three tested devices. This size has given a good CU occupancy / load balancing tradeoff in all tested cases. We refer the interested reader to [17][24][21] for more information on this topic.

5.2.2 Analyzing PCI-Express efficiency

Let us analyze now how the OpenCL S-KDE code makes use of PCI-Express. The data transfer stage is mandatory when using discrete accelerators and, depending on its use, it can turn into a bottleneck. As we did in the previous section, we first characterize the hardware and then the way our application exploits it. For the former, we used the *BusSpeedReadback* benchmark from the SHOC suite [4], which measures the attainable GB/s when reading from the accelerator, for different block sizes. Then, we measured the GB/s achieved by our application, using the block size (size of the intermediate data structures) determined by the chosen chunk size. Results are depicted in Figures 12, 13 and 14 for AMD Radeon, NVIDIA GTX and Intel Xeon Phi respectively. Note how the square tick is not in the same position in the three graphs, because of the different chunk sizes used.

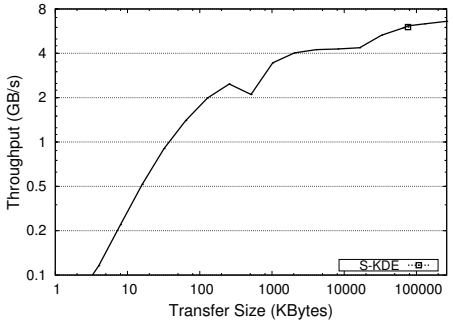

Fig. 12 PCI-Express Throughput Test in Radeon HD 6950

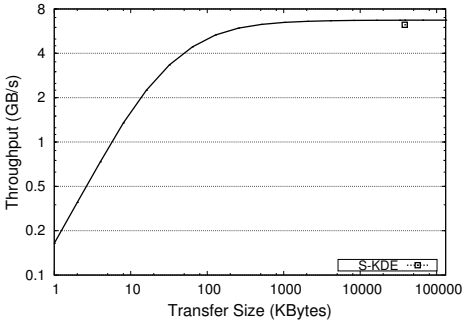

Fig. 13 PCI-Express Throughput Test in GTX 650

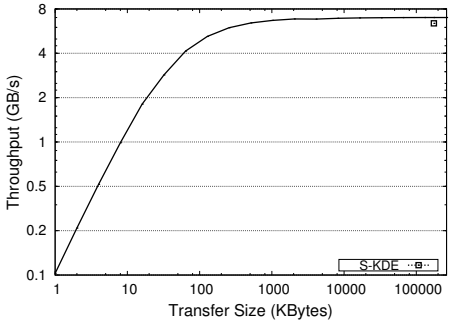

Fig. 14 PCI-Express Throughput Test in Xeon Phi 3120A

Regarding the hardware side, we see how maximum efficiency of PCI-Express is only achieved when moving large data blocks. However, the curves for each device are different, due to differences in the OpenCL run-times and in the PCI-Express management routines in each platform. The chunk sizes used in our programs allow the transfers to be in the most efficient regions.

5.3 Overlapping stages

As described in Section 4, the iterative behavior of our code makes it suitable for working in a pipelined mode, where operations over different chunks of data are overlapped. This mode of operation allows the simultaneous use of CPU and accelerator, thus further accelerating program execution.

We have implemented a two-stage pipeline as depicted in Figure 15: the computation of the partial densities corresponding to a chunk in the accelerator, followed by the transfer of the partial results through PCI-Express, is overlapped with the consolidation in CPU of the results from a previous chunk. This pipeline configuration has been motivated by the execution times of the different stages shown in Section 4. We have implemented it using the Pthreads API, launching a thread for the OpenCL-related operations, and a separate one for the Consolidation.

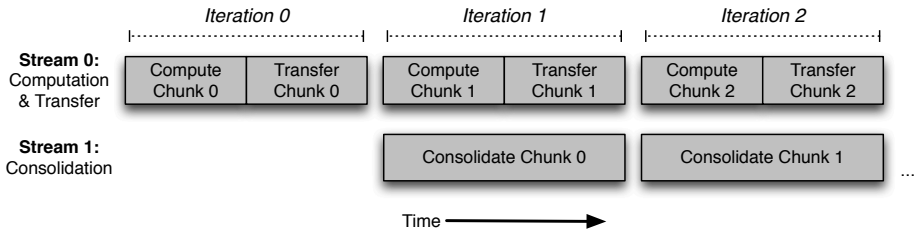

Fig. 15 Pipelined execution of the OpenCL implementation of S-KDE

We illustrate the efficiency of the pipelined program with the execution times for the dataset with 1 million samples, for three different sizes of the evaluation space. Results are shown in Table 4. Each column shows the accumulated execution times in seconds for each of the stages, and the total accumulated execution time (excluding, for the sake of clarity, initialization and finalization). Improvements over the non-pipelined operations are important, for the three devices.

It is to be remarked that running two operations simultaneously cause, in some cases, extra delays. For example, in the three devices, the Consolidation costs are higher in the pipelined program than in the non-pipelined version. The same thing happens with the Transfer stage, but only in the Radeon GPU. The Computation kernels executed in the accelerators require the same time for both modes of operation. These overheads seem to be caused by the high pressure on the memory bus, as both the Transfer and the Consolidation are memory intensive operations. We tried leaving the Transfer stage out of the pipeline, overlapping in each iteration just Computation and Consolidation (doing the Transfer immediately afterwards) and then per-stage execution times where the same obtained without the pipeline. However, this last approach resulted in worse performance results, and we left the numbers out of the tables.

In a further step, we implemented a three-stage pipeline, where all the stages are overlapped. However, the global performance results where similar to the ones given by the two-stage pipeline. This was to be expected as, in the tested platforms, the Consolidation stage takes longer than the summed execution times of the Computation and the Transfer stages. This approach would be beneficial if the relative durations of the different stages was different, for example if Consolidation times were shorter.

The use of the presented two-stage pipeline improves the execution time differently in each device. Interestingly, the GTX GPU is the one offering a more efficient simultaneous operation of CPU and accelerator (1.81 performance gain), while the Xeon Phi and the Radeon are not that efficient (1.36 and 1.40 respectively). Compared to the serial version, total speed-up values (for the largest problem size) are now improved to 4.42x, 5.74x and 5.67x for the Radeon, the GTX and the Xeon Phi respectively. Note how an efficient pipelined operation results in the GTX being the best performed, when in theory this is the least powerful accelerator.

5.4 Discussion

We can summarize the analysis stating that we have reached acceptable levels of efficiency given the memory-bound nature of the S-KDE algorithm, that severely limits the attainable performance. The resulting OpenCL code can be considered useful (it offers 4.42x-5.74x speedup using the pipelined operation), but it does not make a good use of current accelerators: operational intensity is too low, and PCI-Express data transfer costs are high. An additional factor

Ev. Space Size	Radeon HD 6950				GTX 650				Xeon Phi 3120A			
	Comp.	Transfer	Cons.	Total	Comp.	Transfer	Cons.	Total	Comp.	Transfer	Cons.	Total
No Pipeline												
7M	2.49	2.41	5.17	10.06	5.71	2.34	3.66	11.71	2.44	2.24	3.43	8.11
31M	9.13	8.48	22.24	39.86	19.38	8.19	17.61	45.19	8.94	7.96	17.00	33.90
195M	62.38	52.37	173.29	288.05	124.99	49.73	154.85	329.58	46.05	48.61	150.34	245.00
2 Stage Pipeline												
7M	2.47	3.85	6.46	6.78	5.68	2.33	3.46	8.05	2.40	2.27	4.59	4.80
31M	9.13	13.50	27.04	27.09	19.34	8.19	17.14	27.57	9.02	8.07	21.64	21.75
195M	63.72	75.04	204.69	204.77	125.88	50.90	181.62	182.05	47.10	48.81	178.71	178.86

Table 4 Runtimes (s) of different stages for the accelerators. for the problem with 1M samples. for different sizes of the evaluation spaces

that prevents a really fast S-KDE code is the Consolidation stage: additional recoding efforts are necessary to improve this CPU-side code.

One of our main objectives when writing this program was portability in terms of both code and performance. We achieved it, without spending excessive time carrying out per-device optimizations. The performance analysis carried out has also been done with device-independent tools. More detailed information could have been obtained using tools (profilers) provided by the device manufacturers; for example CodeXL[3], Visual Profiler[4] or VTune[5] for AMD, NVIDIA and Intel platforms respectively. These tools are essential when coarse-grain optimizations are not applicable, or to fine-tune for a specific device.

From the previous analysis we can also draw some conclusions about the cost of accelerating S-KDE. We have not included in the previous tables the prices of the tested devices, because they change constantly, but they currently are around $100 for AMD and NVIDIA GPUs (these are consumer-grade devices, now discontinued) and $1700 for the Intel Xeon Phi (a server-grade device designed for a different, smaller market). In theory, the high price tag of the Xeon Phi should be balanced with the ease with which its peak performance can be reached. GPUs are more difficult to exploit, if the application does not show some specific characteristics (high data-parallelism, high operational intensity, low memory contention, low use of the PCI-Express, and so on). For S-KDE, which is not particularly well suited for accelerators using the OpenCL programming model, we can see that the performance of the cheapest GPU is as good as that of the most expensive accelerator.

6 Conclusions

In this work we have presented briefly S-KDE, an efficient algorithm for kernel density estimation, together with its OpenCL implementation targeting modern accelerators. This work complements [15], which discussed the implementation of S-KDE on multi and many-core devices. We have tested the code in three accelerators: AMD Radeon HD 6950 (GPU), NVIDIA GTX 650 (GPU) and Intel Xeon Phi 3120A (many-core).

The S-KDE code does not match particularly well with the OpenCL programming paradigm. It carries out simple operations over massive volumes of data, and it presents memory write contention issues that makes it difficult to delegate important parts of the code (the Consolidation stage) to the accelerator. Despite this, we have achieved significant acceleration (5x) in platforms with very modest GPUs (around $100). We have been unable, though, to exploit efficiently the capabilities of more expensive accelerators, such as the Xeon Phi.

[3] http://developer.amd.com/tools-and-sdks/opencl-zone/codexl/

[4] http://developer.nvidia.com/nvidia-visual-profiler

[5] http://software.intel.com/en-us/intel-vtune-amplifier-xe

We have analyzed thoroughly the characteristics of our code when running on the target hardware, understanding its limits. The major issues with the current program are (1) the need of transferring data through PCI-Express, derived from the use of a discrete accelerator; (2) the low computational intensity of the kernels running in the accelerators, that do not exploit efficiently the floating point capabilities of those devices; and (3) the need to run at the CPU, to avoid memory write issues, a costly Consolidation step. Both the code and the tools used to understand its behavior are device and application independent.

Our future work will address two main issues. The most urgent one is to improve the Consolidation stage, which is currently the one determining the global execution time. This is not a trivial task, because we must deal with synchronization issues. Then, we would like to extend the work related in this paper to convert it into an simple but powerful analysis methodology, to be used with any OpenCL code, but device independent. The literature of code optimization for accelerators includes excellent device-specific resources (such as the good practices manuals [2][19] shipped by the manufacturers) and also application-specific works (e.g. [1] for cryptographic primitives or [20] for mathematical functions), but device and application-independent tools are still missing.

References

1. Agosta G, Barenghi A, Di Federico A, Pelosi G (2014) Opencl performance portability for general-purpose computation on graphics processor units: an exploration on cryptographic primitives. Concurrency and Computation: Practice and Experience DOI 10.1002/cpe.3358
2. AMD (2013) App opencl programming guide. `http://developer.amd.com/tools/hc/AMDAPPSDK/assets/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide.pdf`
3. Cramer T, Schmidl D, Klemm M, an Mey D (2012) Openmp programming on intel xeon phi coprocessors: An early performance comparison. In: Proceedings of the Many-core Applications Research Community Symposium, pp 38–44
4. Danalis A, Marin G, McCurdy C, Meredith JS, Roth PC, Spafford K, Tipparaju V, Vetter JS (2010) The scalable heterogeneous computing (shoc) benchmark suite. In: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, ACM, New York, NY, USA, GPGPU '10, pp 63–74
5. Elgammal A, Duraiswami R, Davis L (2003) Efficient kernel density estimation using the fast gauss transform with applications to color modeling and tracking. IEEE Transactions on Pattern Analysis and Machine Intelligence 25(11):1499–1504
6. Fukunaga K (1990) Introduction to statistical pattern recognition (2nd ed.). Academic Press Professional, Inc., San Diego, CA, USA

7. Jeffers J, Reinders J (2013) Intel Xeon Phi Coprocessor High Performance Programming, 1st edn. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA
8. Jia H, Zhang Y, Long G, Xu J, Yan S, Li Y (2012) Gpuroofline: A model for guiding performance optimizations on gpus. In: Euro-Par 2012 Parallel Processing, Lecture Notes in Computer Science, vol 7484, Springer Berlin Heidelberg, pp 920–932
9. Khronos OpenCL Working Group (2008) The opencl specification. A Munshi, Ed
10. Kim KH, Kim K, Park QH (2011) Performance analysis and optimization of three-dimensional FDTD on GPU using roofline model. Computer Physics Communications 182(6):1201–1207
11. Kirk DB, Hwu WmW (2010) Programming Massively Parallel Processors: A Hands-on Approach, 1st edn. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA
12. Lattner C, Adve V (2004) Llvm: a compilation framework for lifelong program analysis transformation. In: Proceedings of the International Symposium on Code Generation and Optimization, 2004. CGO 2004., pp 75–86
13. Lee VW, Kim C, Chhugani J, Deisher M, Kim D, Nguyen AD, Satish N, Smelyanskiy M, Chennupaty S, Hammarlund P, Singhal R, Dubey P (2010) Debunking the 100x gpu vs. cpu myth: An evaluation of throughput computing on cpu and gpu. SIGARCH Comput Archit News 38(3):451–460
14. Lopez-Novoa U, Mendiburu A, Miguel-Alonso J (2015) A survey of performance modeling and simulation techniques for accelerator-based computing. IEEE Transactions on Parallel and Distributed Systems 26(1):272–281
15. Lopez-Novoa U, Sáenz J, Mendiburu A, Miguel-Alonso J (2015) An efficient implementation of kernel density estimation for multi-core and many-core architectures. International Journal of High Performance Computing Applications DOI 10.1177/1094342015576813
16. Lopez-Novoa U, Sáenz J, Mendiburu A, Miguel-Alonso J, Errasti I, Esnaola G, Ezcurra A, Ibarra-Berastegi G (2015) Multi-objective environmental model evaluation by means of multidimensional kernel density estimators: Efficient and multi-core implementations. Environmental Modelling & Software 63(0):123–136
17. Munshi A, Gaster B, Mattson TG, Fung J, Ginsburg D (2011) OpenCL Programming Guide, 1st edn. Addison-Wesley Professional
18. Nickolls J, Dally W (2010) The gpu computing era. IEEE Micro 30(2):56–69
19. NVIDIA (2012) Opencl best practices guide. `www.nvidia.com/content/cudazone/CUDABrowser/downloads/papers/NVIDIA_OpenCL_BestPracticesGuide.pdf`
20. Pennycook S, Hammond S, Wright S, Herdman J, Miller I, Jarvis S (2013) An investigation of the performance portability of opencl. Journal of Parallel and Distributed Computing 73(11):1439 – 1450

21. Seo S, Lee J, Jo G, Lee J (2013) Automatic opencl work-group size selection for multicore cpus. In: Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT), pp 387–397
22. Sheather SJ (2004) Density estimation. Statistical Science pp 588–597
23. Silverman BW (1986) Density estimation for statistics and data analysis, vol 26. Chapman & Hall/CRC
24. Torres Y, Gonzalez-Escribano A, Llanos DR (2013) ubench: exposing the impact of cuda block geometry in terms of performance. The Journal of Supercomputing 65(3):1150–1163
25. Wang Y, Qin Q, SEE SCW, Lin J (2013) Performance portability evaluation for openacc on intel knights corner and nvidia kepler. In: HPC China 2013
26. Weissbach R (2006) A general kernel functional estimator with general bandwidth-strong consistency and applications. Journal of Nonparametric Statistics 18(1):1–12
27. Williams S, Waterman A, Patterson D (2009) Roofline: An insightful visual performance model for multicore architectures. Communications of the ACM 52(4):65–76