2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing

Evaluation of successive CPUs/APUs/GPUs based on an OpenCL finite difference stencil

Henri Calandra*, Romain Dolbeau†, Pierre Fortin‡, Jean-Luc Lamotte‡, Issam Said‡
*Total, Avenue Larribau, 64000 Pau, France.
†CAPS Entreprise, 4 Allée Marie Berhaut, 35000 Rennes, France.
‡UPMC Univ Paris 06 and CNRS UMR 7606, LIP6, 4 place Jussieu, F-75252, Paris cedex 05, France.
Contact: issam.said@lip6.fr

***Abstract*—The AMD APU (*Accelerated Processing Unit*) architecture, which combines CPU and GPU cores on the same die, is promising for GPU applications which performance is bottlenecked by the low PCI Express communication rate. However the first APU generations still have different CPU and GPU memory partitions. Currently, the APU integrated GPUs are also less powerful than discrete GPUs. In this paper we therefore investigate the interest of APUs for scientific computing by evaluating and comparing the performance of two successive AMD APUs (family codename *Llano* and *Trinity*), two successive discrete GPUs (chip codename *Cayman* and *Tahiti*) and one hexa-core AMD CPU. For this purpose, we rely on a 3D finite difference stencil, that is optimized and tuned in OpenCL. We detail the most interesting optimizations for each architecture and show very good performance in OpenCL: up to 500 Gflops on *Tahiti*. Finally, our results show that APU integrated GPUs outperform CPUs, and that integrated GPUs of upcoming APUs may match discrete GPUs for problems with high communication requirements.**

Keywords: APU, GPU, finite difference stencil, PCI Express bus, high performance scientific computing

I. Introduction

In recent years, Graphic Processing Units (GPUs) have shown their ability to provide better performance than standard CPUs for many scientific applications thanks to their huge compute power and high internal memory bandwidth. Applications with high CPU-GPU communication requirements may be bottlenecked by the *PCI Express* transfer rate (about 6 GB/s for a PCIe 2.0 x16 bus). The AMD APU (Accelerated Processing Unit) with its new hardware design (Fusion architecture), that removes the PCI Express interconnection, can address this problem. But, integrated GPUs in APUs have distinct memory partitions, are almost an order of magnitude less compute powerful and have a lower memory bandwidth than discrete GPUs. Our focus on this paper is to benchmark the first generations of APUs and compare their performance against those of two successive generations of GPUs and of a CPU using an *OpenCL* (Open Compute Language) kernel : a single precision finite difference 3D stencil which is memory bound. Finite difference stencils are widely used in numerical simulations codes in a wide range of applications such as depth imaging [1], computational fluid dynamics [2] and electromagnetics [3]. More notably in [1], the authors emphasize the impact of the slow PCI Express transfer rate on the data snapshotting times and consequently on the stencil performance. We aim to determine whether the integrated GPU can outperform discrete GPUs and CPUs depending on the data snapshotting frequency. This requires relevant data placement within an APU as well as highly efficient applicative kernels on each architecture. The rest of the paper is organized as follows. To begin with, we give a brief overview about the OpenCL compute model in Section II, where we also describe the architectures of the CPU, APUs and GPUs that we use in the present work. Second, in Section III we expose the APU memory system and discuss the different data placement strategies and their impact on APU performance. Then in Section IV, we present our OpenCL implementations of the stencil algorithm. Section V describes performance details and analysis of our stencil implementations for each architecture. In this section, we also present a performance comparison of the CPU, APUs and GPUs. Finally, we conclude in Section VI with a mention of future work.

II. OpenCL and architectures overview

In this section, we briefly summarize the architecture and the technical specifications of the different hardwares used in this paper from an OpenCL perspective.

OpenCL supports both data-parallel and task-parallel programming models. Each hardware is referred to as a *compute device* or device. Each device comprises numerous compute units (CUs), each of which has many processing elements (PEs). Ultimately they are arrays of ALUs (On AMD GPUs, floating point operations are held in ALUs) in GPUs or FPUs in CPUs. The processing elements use SIMD execution of scalar or vector instructions. Every instance of an OpenCL *kernel* (a function that executes on OpenCL devices), called work-item, executes on a PE, simultaneously with other work-items available on the same device, and operates on an independent data set that may be stored in different types of memory: *global memory*, *local memory* and *private memory*. *Work-items* are further grouped into work-groups. Each work-group executes on the same compute unit by groups of 64 work-items called wavefronts. More informations about OpenCL can be found in [4] and [5].

AMD GPUs are evolving along generations and the way they are structured varies with the device family. The Evergreen and Northern Island GPUs, present a *vector design*, which requires vector instructions to reach peak performance. Recently, AMD

1066-6192/12 $26.00 © 2012 IEEE
DOI 10.1109/PDP.2013.65

has released a new GPU family called Southern Island, with a *scalar design* (in reality it is a dual scalar/vector hardware design).

For most AMD GPUs, the processing elements are arranged in SIMD arrays consisting of 16 processing element each. Each of the SIMD arrays executes a single instruction across a block of 16 work-items. That instruction is repeated over four cycles to process the 64-element wavefront. In the Southern Island family, the four SIMD arrays can execute code from different wavefronts.

The APU has a quad-core CPU and an integrated GPU, fused in the same silicon die. The integrated GPU of Llano is an Evergreen GPU and has five GPU cores while the integrated GPU of Trinity belongs to the Northern Island family and has six GPU cores. We are only interested in the integrated GPU of an APU as it represents the major compute power (82% for Llano and 85% for Trinity). At the time of writing, zero copy on APUs is only possible on Windows based systems. Thus, we run all the tests using a Windows OpenCL driver (Catalayst 12.4). In addition, Llano APUs do not support double precision computations, therefore the results presented in this work are all in single precision.

For the CPU a compute unit is actually a CPU core and vector instructions are *SSE* instructions.

We surveyed one hexa-core CPU: *AMD Phenom TM II x6 1055t Processor*, two discrete GPUs: a Cayman GPU and a Tahiti GPU, and two APUs: Llano and Trinity. Table I shows the technical specifications of each tested device. Note that the APU specifications are only about the integrated GPUs.

III. Data placement strategies

The APU memory system is different from that of a discrete GPU. On the one hand, the CPU (host) and the GPU (device) are fused in the same socket and the PCI Express interconnection between them is removed but on another hand, the memory address space is not unified as the integrated GPU has its own dedicated memory partition which is a sub-partition of the system memory. The integrated GPU can have access to memory using one of the two following buses: *Garlic*, which is a cache non-coherent fast bus that can achieve 25.6 GB/s of data transfer rate, and *Onion*, a cache coherent slow bus with a maximum theoretical bandwidth of 8 GB/s. We recall that the PCI Express transfer rate between CPU and discrete GPUs is measured between 6 GB/s and 7 GB/s when pinned memory is used. Not only data can be explicitly copied from the system memory to the GPU partition and vice versa but also memory objects, known as *zero-copy* objects, can be shared between the CPU and the integrated GPU [5]. In fact, part of the GPU memory can be exposed to the CPU: this is referred to as *host-visible device memory* and is also called the GPU *persistent memory*. Similarly, a subset of the system memory can be made visible to the integrated GPU. GPU Read-only *zero-copy* memory objects are stored in the USWC (Uncacheable, Speculative Write Combine) memory, an uncached memory (part of the *device-visible host memory*) in which data is not stored into the CPU caches and is subject to CPU contiguous write operations using the *write combine* buffers to increase memory throughput. Besides, subsequent GPU reads from this memory are fast as the Garlic bus is used.

We refer to the different memory locations of the APU as a lower case letter: `c` refers to the regular cacheable CPU memory (always pinned here for efficient CPU-GPU data transfer), `z` to the *device-visible host memory* employed as cacheable zero-copy memory objects, `u` to the *device-visible host memory* employed as uncached zero-copy memory objects (USWC), `g` to the regular GPU memory and `p` to the GPU *persistent memory*.

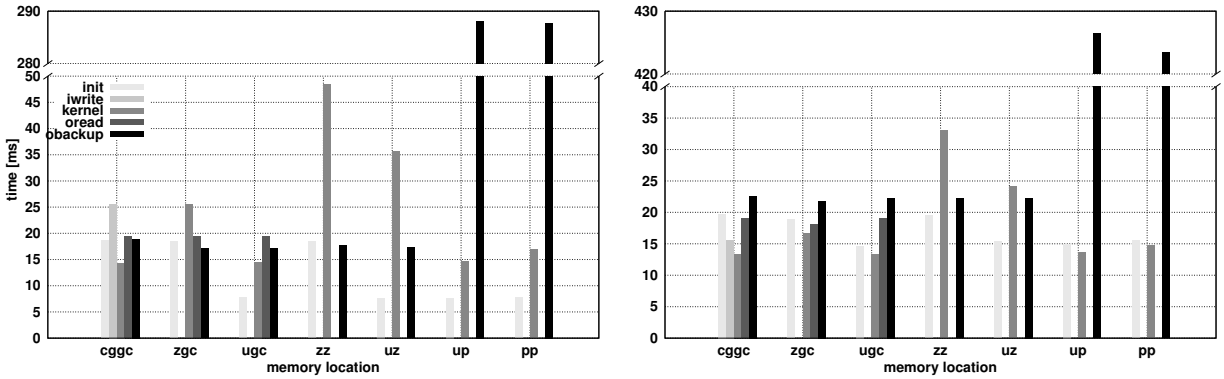
In order to test the performance of the read and write accesses to these buffers, we developed an OpenCL benchmark, a data placement benchmark, that makes the integrated GPU copy data from an input buffer stored in one APU memory location to an output buffer that lies in another APU memory location. For both the input and output buffers, `cg` (respectively `gc`) denotes an explicit data copy from the CPU partition to the GPU partition `g` (resp. from the GPU partition `g` to the CPU partition `c`), wheras `z`, `u` and `p` refer to the corresponding zero-copy buffer. For example, `zgc` describes the following data placement strategy: the input buffer is in `z` memory location, the output buffer is first created in the GPU memory (`g`) and then, is explicitly copied to the CPU memory (`c`). We tried different data access strategies. We show the results in Fig.1 for both Llano and Trinity, where *init* is the input buffer initialization (using a CPU parallel *memcpy* operation) time. *iwrite* is the input buffer transfer time (if needed) to the GPU memory. *kernel* is the execution time of the OpenCL kernel, that copies data from the input buffer to the output buffer. *oread* is the output buffer transfer time (if needed) back to the CPU. *obackup* is the time of an extra copy from the output buffer to a temporary buffer in the CPU memory in order to measure the time of reading from the memory location in which the output buffer is saved. In some cases, a *map* operation (resp. an *unmap* operation) is required before *iwrite* or *oread* (resp. after *iwrite* or *oread*). These operations have negligible times compared to the other operations and therefore are not presented in our results. We use system wall-clock for timing this benchmark, as well as for the rest of the tests presented in this paper. All the tests are run multiple times (up to 40) after devices "warm up". First, we note that GPU reads from USWC are as fast as GPU reads from GPU memory. CPU writes to GPU persistent memory are fast but reads are very slow (see *obackup* in Fig.1). Contiguous CPU writes to USWC (`u`) offer the highest bandwidth for *init*. Finally, GPU memory accesses to `z` (via Onion with a sustained bandwidth of 6 GB/s) are slower than accesses to `u` (via Garlic with a sustained bandwidth of 12 GB/s) and `g`. For the rest of the paper we select the most relevant placement strategies: *cggc*, *ugc*, *uz* and *up*.

IV. Finite difference stencil OpenCL implementations

In this section we present a description of our implementation choices and optimizations for the single precision 3D finite

TABLE I
HARDWARE SPECIFICATION

	CPU	discrete GPUs		integrated GPUs	
Architecture	Thuban	Cayman	Tahiti	Llano	Trinity
Model	Phenom	HD6970	HD7970	A8-3850	A10-5700
GPU family name	Ø	Northern Island	Southern Island	Evergreen	Northern Island
Clock rate (GHz)	2.8	0.88	0.925	0.600	0.711
Compute units	6	24	32	5	6
Global memory (MB)	8096	2048	3072	512	512
Local memory (KB)	32	32	64	32	32
Peak bandwidth (GB/s)	50	176	256	25.6	25.6
Peak flops (Gflop/s)	134	2700	3700	480	546



(a) Llano (b) Trinity

Fig. 1. Data placement benchmark results

difference stencil algorithm. Our major optimizations include use of vector instructions, use of the local memory, software pipelining, register blocking and auto-tuning (for each OpenCL kernel we search the best combination of certain parameters, such as the work-group size and the ILP (Instruction Level Parallelism) in order to make best use of each tested device). The OpenCL implementations are highly tuned in order to provide a fair performance comparison between all tested devices. Thanks to the OpenCL portability and programming model, for each test we execute the same implementations (albeit automatically tuned) on all the devices.

A stencil computation of an element in a 3D NxNxN regular mesh is a linear summation of its value and its neighboring values, along each dimension, weighed by specific coefficients. We choose to compute an 8^{th} order 3D stencil. The floating point computation and data storage complexities of the stencil kernel are both $O(N^3)$, the exact number of flops being $2 + ((3 * 13) * N^3)$. The compute intensity is therefore $O(1)$ and this algorithm is memory bound. We apply a 2D work-item grid on the 3D domain and we first implement a *scalar* version in which each work-item computes X columns of the domain (X refers to the ILP, is determined via auto-tuning and is either equal to 2 or 4 in practice) along the Z dimension which corresponds to the largest strided memory accesses[6]. All memory accesses are issued in global memory. Second, we explicitly vectorize the code and each work-item processes X columns of *float4* elements along the *Z* dimension. This implementation is referred to as *vectorized*. Finally, we implement a blocking algorithm [7]. Data is fetched slice by slice from global memory and stored in local memory. Then, each work-item traverses the Z dimension and blocks the grid values in the register file in order to reuse data and computes X *float4* elements at a time. This implementation is referenced as *local vectorized*.

V. PERFORMANCE RESULTS AND COMPARISON

In this section, we show some performance results for the OpenCL kernels on each tested device. We also emphasize the impact of data placement strategies on APUs performance. Finally, we compare the performance of the integrated GPUs against those of the CPU and the discrete GPUs.

1) Devices performance: We first show the performance numbers of each tested device based on the kernel execution time only. We use 3D domains with NxNx32 sizes (N ranging between 64 and 1024). Figure 2(a) summarizes the performance of the different OpenCL implementations on the CPU. We compare them against an OpenMP Fortran 90 code (compiled and vectorized with Intel Fortran Compiler). The OpenCL vector implementations outperform the OpenCL

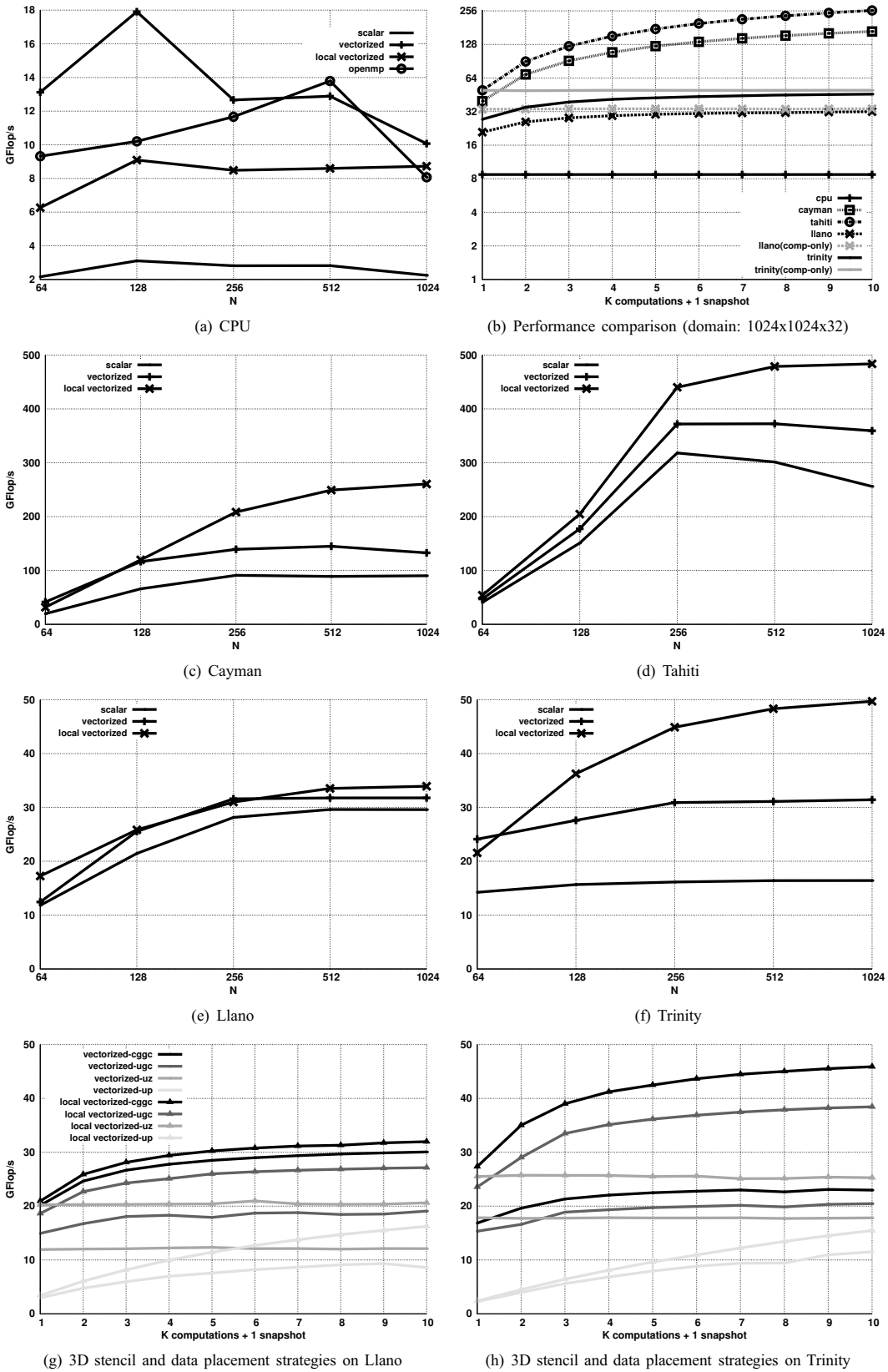


Fig. 2. Performance summary

scalar version thanks to their use of SSE instructions. The *vectorized* implementation efficiently relies on CPU caches and delivers the best performance being faster than or as fast the OpenMP implementation. Figures 2(c), 2(d), 2(e) and 2(f) illustrate the performance numbers of Cayman, Tahiti, Llano and Trinity respectively. We note that for discrete GPUs, as well as for integrated GPUs, the *local vectorized* implementation is more efficient than the *vectorized* one thanks to the blocking in the local memory. The performance of Tahiti reaches up to 484 Gflop/s. Note that OpenCL image memory objects provided very poor performance for large domains. We do not consider including them in the presented results.

2) Impact of data placement strategies on performance: Stencil computations are usually used in iterative methods. Between two subsequent iterations, data need to be resident on the CPU memory in order to be used in further operations such as *imaging condition* in RTM (Reverse Time Migration) applications. We call this process *data snapshotting*. The frequency of data snapshotting, also called temporal blocking, can also be an important performance factor. We run the *vectorized* and the *local vectorized* implementations of the stencil OpenCL kernel on APUs while taking into consideration the data placement strategies selected in Section III in one hand, and the frequency of data snapshotting in an other. Note that we use one input buffer and one output buffer. In Fig.2(g) and Fig.2(h) we show the performance results of the kernel ran on a 1024x1024x32 grid, respectively on Llano and Trinity, as a function of the number of stencil computation passes performed before the snapshot (frequency of data snapshotting). Note that the communication times (possibly required to retrieve the snapshot on the CPU) are included in those numbers. We conclude that in order to obtain the best stencil performance on APUs, we have to use the *local vectorized* implementation coupled with the *cggc* data placement strategy.

3) Performance comparison: Finally, Fig.2(b) illustrates a performance comparison of the tested devices as a function of the frequency of data snapshotting, using the best implementation on each. The domain size of the used grid is 1024x1024x32. The best performance is always obtained on the Tahiti discrete GPU. A Trinity APU with a unified memory (*trinity (comp-only)*) may match the Tahiti performance, and outperform the Cayman performance, for a snapshot retrieval after every stencil computation. Note also that integrated GPUs always outperform the CPU implementation.

VI. Conclusion

The new AMD APU architecture eliminates the PCI Express bus which bottlenecks many GPU applications. However, integrated GPUs in APUs are less compute powerful and have less internal memory bandwidth than discrete GPUs. Moreover, while the upcoming APUs will rely on a unified memory, the first APUs still have a distinct GPU memory partition. In this paper we have discussed the relevance of the APU integrated GPU for high performance scientific computing by providing a comparative study of the CPU/APU/GPU performance of an OpenCL finite difference 3D stencil kernel.

Our results indicate that we can achieve very good performance on each tested device with a set of OpenCL implementations, automatically tuned for each device. Our implementations take advantage of the hardware improvements along successive GPU generations. The new *Tahiti* discrete GPU delivers impressive performance (up to 500 Gflops).

The two integrated GPUs outperform the CPU for all data snapshotting frequencies. But, only the integrated GPU of the latest APU (*Trinity*) can match discrete GPUs for problems with high communication requirements, provided that we simulate a unified memory system for this APU. As far as performance is concerned, there is indeed a big gap, in both compute power and internal memory bandwidth, between the latest discrete GPUs and the integrated GPUs in the latest APUs. For future APUs to be competitive for this kind of applications (i.e to outperfom discrete GPUs), they should be endowed with more powerful integrated GPUs and a faster memory system.

Besides, we point out that APUs are low consuming chips (about 100 Watts of TDP, which is similar to our *Phenom* multi-core CPU) compared to discrete GPUs (about 250 Watts of TDP). Considering a server with two or three APUs against a single discrete GPU, which roughly corresponds to the same power consumption[1], our results clearly suggest that the integrated GPUs may offer the best power-performance ratio. Therefore, we aim to include power consumption measures in our future comparative studies.

Finally, the removal of the PCI Express interconnection in the APUs encourages the use of hybrid CPU - integrated GPU OpenCL implementations for this applicative kernel. We are also planning to explore this hybrid CPU - integrated GPU approach in our future implementations.

Acknowledgment

The authors are very grateful to AMD for kindly providing us with the surveyed hardware. This work was funded by Total which is kindly acknowledged.

References

[1] R. Abdelkhalek, H. Calandra, O. Coulaud, G. Latu, and J. Roman, "Fast Seismic Modeling and Reverse Time Migration on a GPU Cluster," in *The 2009 High Performance Computing & Simulation - HPCS'09*, 2009.

[2] S. Vanka, A. Shinn, and K. Sahu, "Computational fluid dynamics using graphics processing units: Challenges and opportunities," *International Mechanical Engineering Congress and Exposition - IMECE'11* , 2011.

[3] I. Tsukerman, "Electromagnetic applications of a new finite-difference calculus," *Magnetics, IEEE Transactions on*, vol. 41, no. 7, pp. 2206 – 2225, 2005.

[4] Khronos Group, "The OpenCL Specification version 1.2," 2011.

[5] AMD, "Accelerated parallel processing opencl programming guide," 2012.

[6] K. Datta, S. Williams, V. Volkov, J. Carter, L. Oliker, J. Shalf, and K. Yelick, "Auto-tuning the 27-point stencil for multicore," in *In Proc. iWAPT2009: The Fourth International Workshop on Automatic Performance Tuning*, 2009.

[7] P. Micikevicius, "3D finite difference computation on GPUs using CUDA," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU-2. ACM, 2009, pp. 79–84.

[1]The power consumption of the CPU required to drive the computations on the discrete GPU will also have to be taken into account.