Hybrid strategy for stencil computations on the APU

Pacôme Eberhart Issam Said Pierre Fortin
UPMC Univ Paris 06 and CNRS UMR 7606, LIP6
4 place Jussieu, F-75252, Paris cedex 05, France
Contact : Pacome.Eberhart@lip6.fr

Henri Calandra
Total
Avenue Larribau, 64000 Pau,
France

ABSTRACT

Stencil computations are very regular and well adapted to GPU execution. However, the PCI-E bus that connects a discrete GPU to the system memory has a relatively low bandwidth when compared to the GPU compute power. The AMD APU architecture contains both CPU and GPU on the same chip and shared memory between them, which enables to bypass this PCI-E bus. In this paper, we devise a strategy for hybrid deployments on the CPU and the integrated GPU of the APU. For the task-parallel deployment, we rely on the CPU to process the diverging parts of the application. For the data-parallel deployment, we balance the workloads of the CPU and the GPU to achieve the best performance. Our strategy is tested on different stencil computations and we achieve a 20 to 30% gain in performance in the best cases.

Categories and Subject Descriptors

C.1.3 [**Processor Architectures**]: Other Architecture Styles—*heterogeneous (hybrid) systems*; G.1.8 [**Numerical Analysis**]: Partial Differential Equations—*Finite difference methods*; G.4 [**Mathematical Software**]: Parallel and vector implementations

Keywords

APU, heterogeneous architecture, OpenCL programming, finite difference stencil, divergence, task parallelism, data parallelism

1. INTRODUCTION

In the industrial context of oil exploration for Total, the RTM (Reverse Time Migration [3]) allows to check for the presence of oil fields without the high cost of drilling. First, data is gathered in a prospect area by sending vibrations into the ground and capturing reflected waves. Then, by comparing simulated wave propagation and the captured data, the composition of underground layers is deduced.

Finite difference stencil computations are the core of the RTM, as well as of many other physics simulations. They approximate the derivation operators in physics equations into linear combinations on a discrete domain. Due to their regularity, they are very well suited for efficient executions on the massively parallel architecture of GPUs (Graphics Processing Units). Several works have thus studied the deployment of the RTM on GPUs [5, 15, 5]. However, when executed on a GPU, the RTM needs to bring data back from the GPU to the CPU. When this *snapshotting* gets very frequent, the slow PCI-E bus data transfer rate has a negative impact on snapshotting times [1], thus slowing down the overall RTM performance.

The AMD APU (*Accelerated Processing Unit*) has both CPU and GPU cores, and shared memory between them. This gives the APU an interesting potential for high performance parallel computing [9, 8]. Other works [6, 7] have shown the potential of APU integrated GPUs for the RTM, especially when considering energy consumption. The memory shared between CPU and GPU on the APU allows indeed to completely bypass the PCI-E bus for the snapshotting.

In this paper, we will try to use both the CPU and the GPU of the APU to gain further performance. The partial SIMD (Single Instruction Multiple Data) execution of a GPU will force diverging control flows to be serialized (compute divergence). Similarly, the cost of memory divergence on a GPU is due to the fact that memory accesses are optimal only when contiguous and aligned. As the costs of computation and memory divergence are lower on CPU, specific treatments on the borders of the domain could be executed on the CPU, while keeping the regular execution within the domain on the GPU. We will call this deployment task-parallel. The CPU could also simply be used for its additional compute power and memory bandwidth in a data-parallel deployment. To decide between those two alternatives, we devise a strategy for hybrid deployments on the APU.

Using a single parametrizable OpenCL source code, we will first select the best performing version for each architecture. We will then detail how to ensure efficient hybrid deployments on the APU according to our strategy.

Section 2 will provide an overview of the APU architecture and of the OpenCL model. Section 3 explains our strat-

HiStencils 2014
First International Workshop on High-Performance Stencil Computations
January 21, 2014, Vienna, Austria
In conjunction with HiPEAC 2014.

`http://www.exastencils.org/histencils/2014/`

egy for hybrid deployments on the APU. In Section 4, we study several strategies for deploying stencil computations on only the CPU or only the GPU, in order to prepare for hybrid deployments. In Section 5, we deploy hybrid stencil computations on the APU and present performance results. Section 6 concludes and discusses future work.

2. APU ARCHITECTURE AND OPENCL PROGRAMMING

The APU A10-5700 (family codename Trinity) we have at our disposal has 4 CPU cores at a 3400 MHz frequency with SSE instructions and 4 MB L2 caches. The integrated GPU has 96 processing elements at a 760 MHz frequency and has no cache. Each of these processing elements can compute four 32 bit floating operations at the same time. The GPU has a maximum theoretical peak performance of 546 GFlop/s and the CPU has 108.8 GFlop/s. As such 83% of the compute power of the APU is contained in the GPU and 17% in the CPU.

The memory of the APU is not entirely shared between the CPU and the GPU. The sytem memory and the GPU memory are only accessible from, respectively, the CPU and the GPU, and can each contain shared locations (resp. device-visible host memory and host-visible device memory) (see Fig. 1). Accessing the memory from the CPU is done through L2 caches or Write Combine (WC) buffers. There are two memory buses for the GPU: *Garlic*, fast (maximum theoretical bandwidth of 25.6 GB/s) but without a coherence protocol with CPU caches, and *Onion*, slower (8 GB/s) but cache coherent. We classify the different locations in memory for buffers according to [6] as:

- c, the system memory, accessible from the CPU only;
- g, the GPU memory, not accessible from the CPU, accessed by the *Garlic* bus;
- p, the GPU memory shared with the CPU;
- u, the CPU memory accessible from the GPU by the *Garlic* bus;
- z, the CPU memory accessible from the GPU by the *Onion* bus.

p, u and z are refered to as *zero-copy* memory buffers. *Garlic* having no coherence protocol with CPU caches, u is read-only for the GPU. The WC buffers on the CPU ensure that writes from the CPU to u are propagated to system memory and visible from the GPU. p and z are read-write enabled locations for both CPU and GPU.

In order to use the CPU and the GPU concurrently, we use OpenCL [11], as OpenCL kernels can be executed on CPU and on GPU, and synchronized through the runtime. Using the SPMD (Single Program, Multiple Data) programming model, the kernel is executed independantly on multiple work-groups, each made of work-items that are executed on hardware threads. On an AMD OpenCL GPU, work-items of a given work-group are executed in several *wavefronts* (similar to *warps* in NVIDIA CUDA), each one being processed synchronously (SIMD). Wavefronts in which work-items execute different control flows will have their executions serialized over the different control flows. On an AMD OpenCL CPU, a thread will execute one by one each

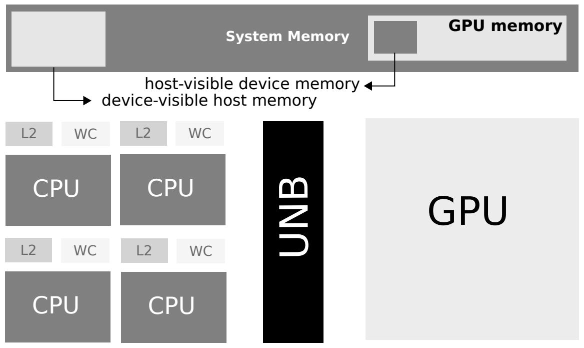


Figure 1: Architecture of the APU (UNB : Unified North Bridge)

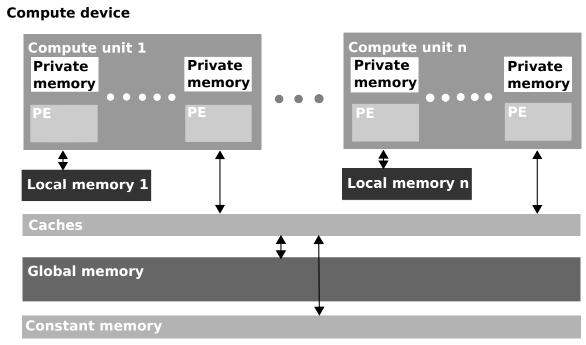


Figure 2: OpenCL memory model

work-item of a given work-group until completion or encountering a barrier. As the work-items are executed one at time, there is no overhead on CPU due to diverging control flows among the work-items.

Here, it has to be noticed that the current AMD OpenCL SDK does not provide implicit vectorization among multiple work-items on CPU. To use the SSE instructions on the CPU, and the vector instructions on the GPU, we therefore need to rely on explicit vectorization by means of vector types (`float4`).

The memory model of OpenCL allows different levels of sharing between work-items (see Fig. 2). The global memory is accessible by all work-items, the local memory is shared within a work-group and registers belong to a single work-item. On a GPU, global memory accesses are optimal when work-items in a given wavefront access a contiguous and aligned memory area at the same time (*coalesced* memory accesses on NVIDIA GPUs). Since the work-items are processed one by one on a CPU, the CPU is also not sensitive to the divergence in memory accesses among multiple work-items. Moreover the CPU caches enable to offset the overhead of irregular memory accesses. Besides, local memory is redundant on CPU with the caches of each core, especially since a work-group is executed on a single core.

An OpenCL program is launched from a host machine. Kernels are compiled for the chosen hardware, and placed in an execution queue for that hardware. Data transfers and barriers can also be enqueued. Finally, the host will synchronize with these queues to ensure that an execution is over.

3. HYBRID STRATEGY FOR THE APU

In stencil computations, the borders of the domain may require specific memory access patterns and may induce compute divergence. This will cause an increase in the cost of computation on a GPU. We here propose the use of the APU CPU for the borders, where the computation will not be serialized. We call this deployment task-parallel. We can also divide the domain into two regions to be executed on the CPU and the GPU, in a data-parallel fashion (see Fig. 3).

To choose between these two alternatives, we propose the strategy presented in Fig. 4. First, we determine if the application is compute-bound or memory-bound. For this purpose, we compare the arithmetic intensity (ratio of the number of memory accesses over the number of arithmetic operations) and the hardware specifications of the integrated GPU. We then try to quantify the divergence, either through a profiler, or through code analysis or even thanks to programmer indications. For a memory-bound application, we only consider divergent memory accesses, whereas we look only at divergent computations in the compute-bound case. If the divergence is high enough according to empirical thresholds, we choose the task-parallel deployment. Otherwise, the data-parallel one will be preferred.

Such strategy could thus be implemented in a generic software platform. In this paper, we will only apply and try to validate this strategy on our stencil computations.

The hybrid strategy applied to stencil computations requires to read from an input buffer and write to an output buffer, for both CPU and GPU. As the buffers will swap between input and output at every iteration, buffers need to be zero-copy and read-write enabled. u memory, being read-only from the GPU, is not a valid choice, only z and p are both zero-copy and read-write enabled from CPU and GPU. Previous work [6] has shown p to be significantly slower than z for CPU reads. The input and output buffers will thus be allocated as z buffers.

Also shown in [6] is the fact that z memory is only more interesting than g memory (along with explicit data copies) when snapshotting is performed at every iteration in the RTM. Memory accesses via *Garlic* (g memory) are indeed much more efficient than via *Onion* (z memory) on current APUs. In the future APU architectures, this performance gap between z and g memory locations should however be reduced, which will widen the interest of using z memory and justify our hybrid approach based on zero-copy buffers.

Before we try to apply this strategy, we study different deployments on CPU-only and GPU-only computations. We will determine which ones are the most efficient and use them as the basis for our task-parallel and data-parallel deployments.

4. STENCIL COMPUTATIONS ON CPU OR GPU

When simulating infinite or semi-infinite domains with stencils, the borders act as reflectors. To correct this, PMLs (Perfectly Matched Layers) [4] are usually used on the borders. This technique creates divergence in the computation on the borders of the domain.

To study the effect of memory access divergence, we use here a basic stencil with a parametrizable size. This enables us to control the amount of divergence by changing its size.

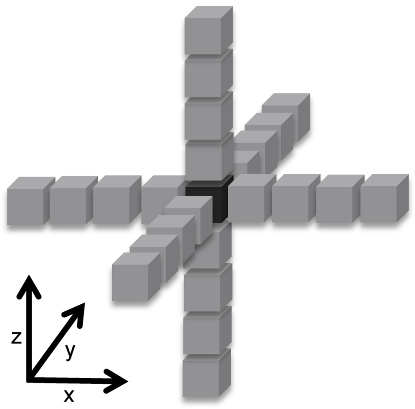


Figure 5: Memory access pattern of an 8th order (size 4) laplacian stencil

To ensure the best control over divergence we therefore do not use PMLs here. We do not use halos either (additional points on the borders of the domain containing only zero values) on the buffers, as they would cancel memory divergence. We will use a laplacian stencil (see Fig. 5), as used in the RTM. For low divergence, we will use a size 4 stencil, and a size 32 for high divergence.

We also examine different deployment strategies following [14]:

- **complete**, one single kernel executed on the 3D whole domain;
- **inout**, one kernel executed on the inside of the domain and another one encompassing all the borders;
- **sides**, one kernel executed on the inside of the domain and one for each of the six borders in 3D.

complete will be the basis for the data-parallel deployment, and the task-parallel deployment will be developed on the most performant between **inout** and **sides**.

The kernels have different versions to exploit at best the different hardware specifications of the CPU and of the GPU. A **scalar** version will allocate work-items on a 2D grid where each of them will iterate along the Z-axis of the 3D domain. The **vectorized** version adds the explicit use of OpenCL vector types (float4) and instructions to compute 4 points at the same time. The **local vectorized** uses the local memory of the work-groups to share the values of points on an XY-plan [13, 12]: this makes the execution benefit from the higher bandwidth of the local memory on common values needed by several work-items.

It has to be noticed that our kernels source codes are fully parametrizable. This enables us to define the size of the stencil and the size of the work-group at kernel compilation time by using the C pre-processor. We can then easily optimize the size of the work-groups for the A10-5700 APU model via an exhaustive search. The domains are three-dimensional and of size N^3.

Due to the existence of vectorized instructions on both the CPU and GPU, the **vectorized** kernel performs consistently better than the **scalar** one thanks to the SSE instructions and the GPU vector processing units (performance tests not presented). The **local vectorized** version does not offer performance gain over **vectorized** either. On the CPU, the

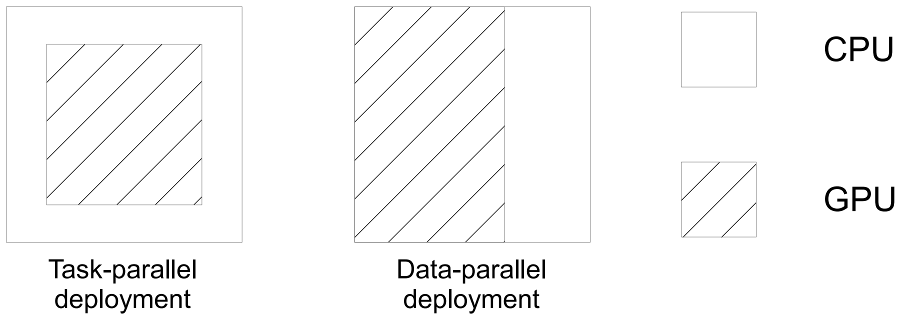


Figure 3: Example of data-parallel and task-parallel deployments on a 2D domain

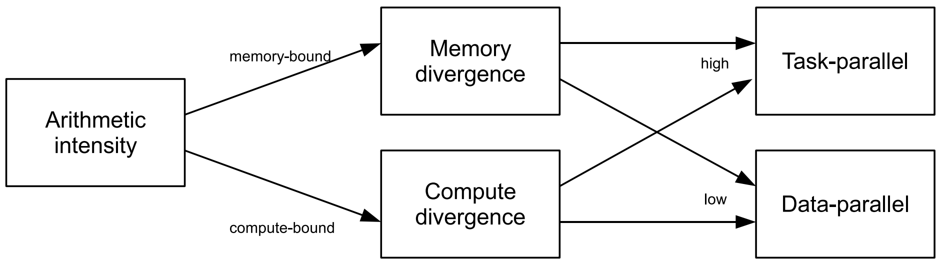


Figure 4: Hybrid strategy for the APU

redundancy of local memory with caches implies indeed an overhead. On the GPU, there is also no performance gain for **local vectorized** with a stencil of size 4. For a stencil of size 32, the amount of local memory required decreases the occupancy, hence the low performance. We will therefore consider only the **vectorized** kernel through the rest of this paper.

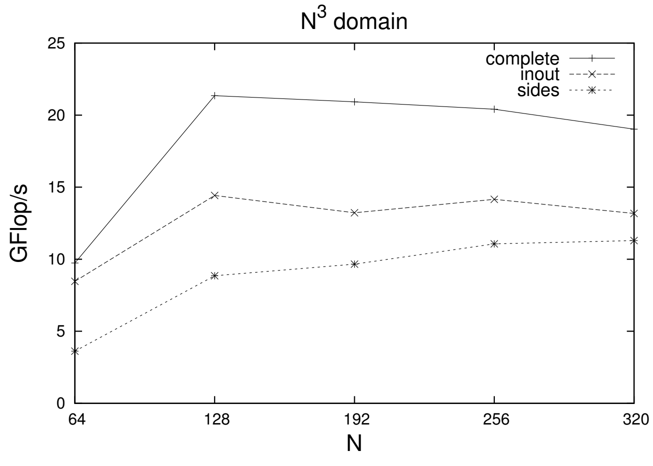
Figures 6a, 6b, 7a and 7b, present performance results of our different deployments on GPU only and on CPU only. This enables us to study the divergence impact on each architecture separately.

On GPU, **sides** is topped by the other strategies, due to the cost of launching the multiple kernels and the low occupancy of the kernels on the borders. On a size 4 stencil (see Fig. 6a), **complete** performs better than **inout**. When the size of the stencil is increased to 32 (see Fig. 6b), memory divergence rises and **inout** almost catches up with **complete** but does not outperform it: this is due to the fact that the AMD GPUs are only slightly sensitive to memory divergence ([2], section 6.4).

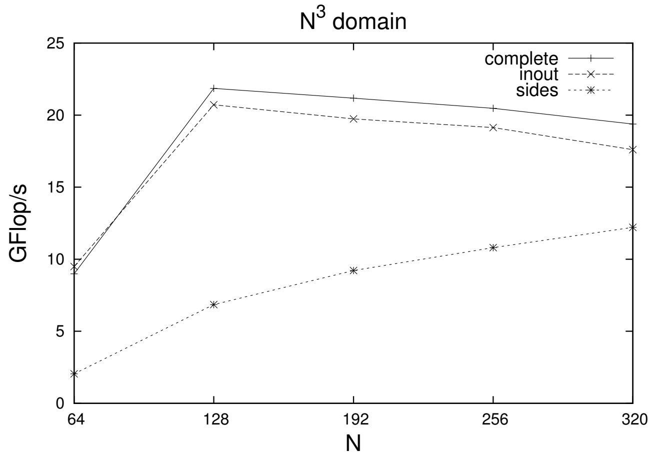
On CPU, performance drops very fast when the domain enlarges (see Figs. 7a and 7b), because a smaller domain, fitting almost entirely in cache, allows for a better memory bandwidth. There is nearly no difference between the performances of the three strategies, as they were designed to handle divergence and CPU hardware is not sensitive to divergence. Domains with sizes which are multiples of 128 offer low performance, probably due to the under-utilization of the interleaved memory banks. The effects are especially noticeable for the size 32 stencil computations, which are subject to higher cache pressure.

We have observed so far the effect of memory divergence on stencil computations, but the little sensivity of AMD hardware to memory divergence has lessened its impact. To more clearly witness the impact of our strategy, we have also studied artificial compute-bound stencils with compute divergence. In this purpose, we first include compute divergence in our kernels by dividing the computation into seven distinct parts: the inside of the domain, and each of the six sides. To ensure that the compute divergence will not be made negligible by the memory-bound nature of our stencils, we artificially raise the arithmetic intensity. To do so, we compute each point of the domain several times and accumulate all the results, while keeping the values necessary for the computation in registers.

In the memory-bound stencil, for each point in our domain, there were 26 memory accesses (25 reads and one write) for 36 arithmetic operations. For the compute-bound stencil, the number of memory accesses remains the same, whereas the number of arithmetic operations is multiplied by the number of iterations. The APU integrated GPU has

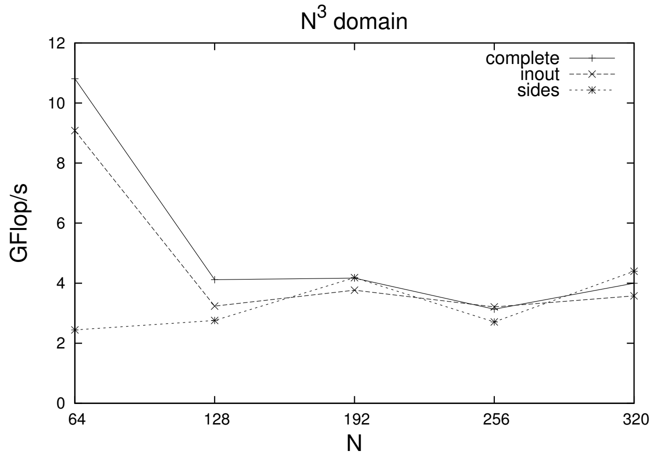


(a) Size 4 stencil

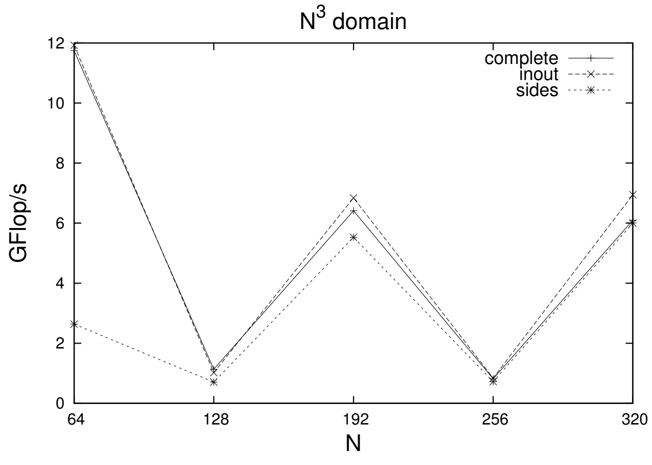


(b) Size 32 stencil

Figure 6: Performance results of stencil computations on the GPU



(a) Size 4 stencil



(b) Size 32 stencil

Figure 7: Performance results of stencil computations on the CPU

a maximum memory bandwidth of 25.6 GB/s, meaning it can access 6.4G floats per second. With a peak performance at 546 GFlop/s, its theoretical threshold between memory-bound and compute-bound applications is around 85 floating point operations per memory access. The arithmetic intensity of our previous basic stencil computation was 1.4, which is clearly memory-bound. With 128 iterations our new articifical stencil computation has an arithmetic intensity of 177 which is clearly compute-bound.

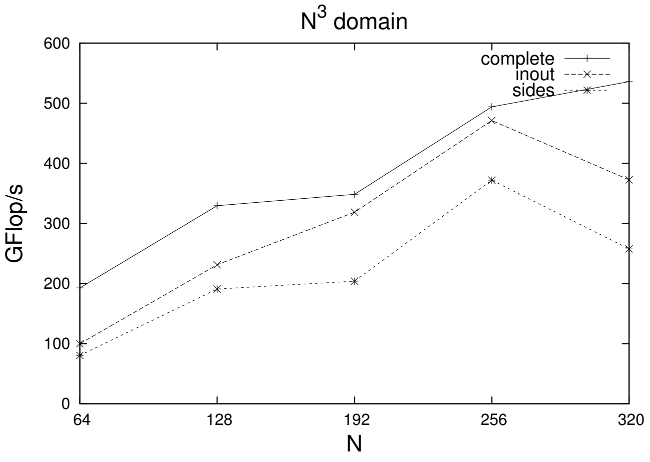
For these compute-bound stencils, we also study our three deployment strategies and we set the stencil size to 4. The relative performance of the three strategies are then similar to the previous ones on both the GPU and the CPU (see Figs. 8a and 8b). On the GPU, *sides* is again performing worse than *inout*, *inout* being closer to *complete*. On the CPU, there is no difference in the performance results of the three strategies, since the CPU is again not sensitive to divergence.

5. HYBRID DEPLOYMENT

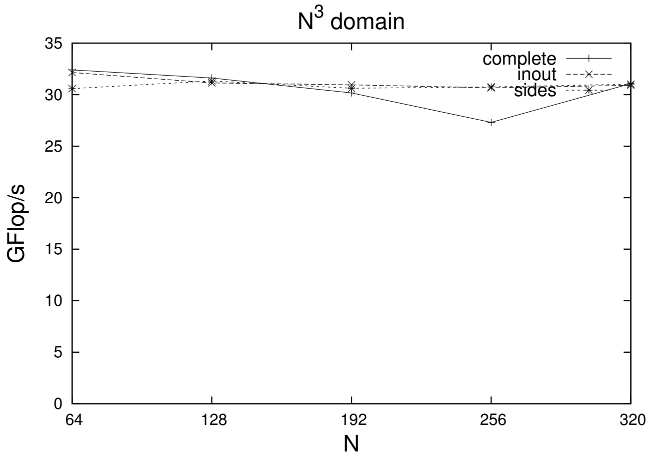
Our two strategies for hybrid deployments, task-parallel and data-parallel, will execute one or several kernels on both CPU and GPU. For the task-parallel deployment, we choose the **inout** strategy, as it consistently performs better than **sides**. We modify it to make the CPU execute a divergent kernel on the borders of the domain, while the GPU executes a regular kernel on the inside. For the data-parallel one, we divide the domain into two subdomains and execute the same kernel on them. We divide the domain along the Y axis, as cutting along the X axis could split up `float4` values, and as the Z axis is the axis work-items iterate along.

We synchronize the execution of the kernels by using a blocking function, `clFinish`, to ensure that the two execution queues (CPU and GPU) are completed. According to the OpenCL specification [11], `clFinish` also ensures that memory writes are visible to both the CPU and GPU.

As noted by [10], the OpenCL standard does not specify a way to share a buffer for concurrent accesses by multiple devices (on distinct data within this buffer). However,

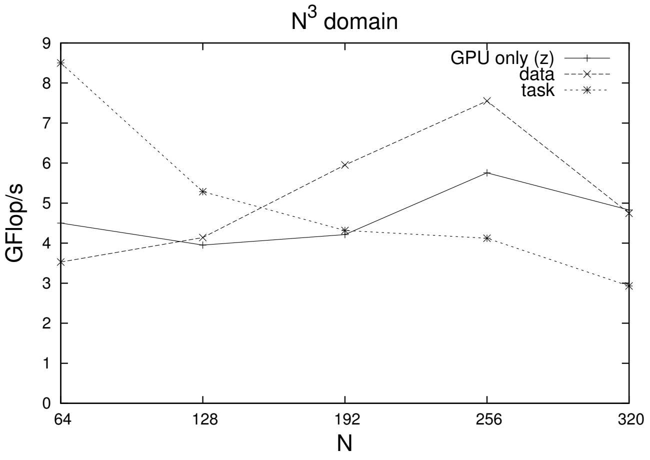


(a) On the GPU

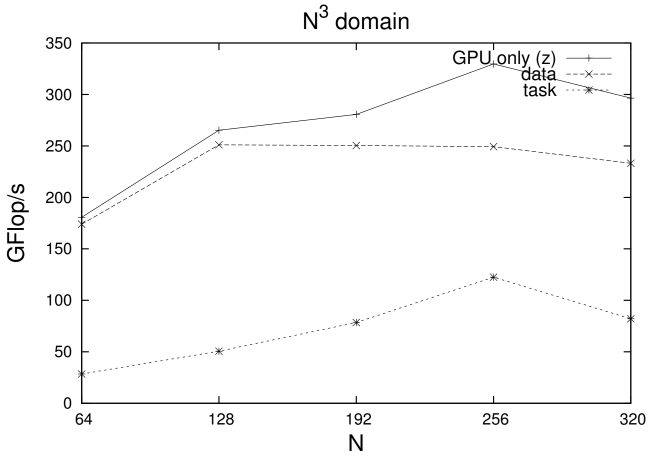


(b) On the CPU

Figure 8: Performance results of compute-bound stencil computations (size 4)



(a) Memory-bound stencil



(b) Compute-bound stencil

Figure 9: Performance results of hybrid deployments of stencil computations

the AMD implementation of OpenCL makes it possible by not deleting the reference to the physical location of the buffer when unmapping. In their case, simply unmapping the buffer from the device allowed them to use it simultaneously on the GPU, as a device, and on the CPU, as a host. In our case, we similarly write to the shared buffer concurrently by the CPU and the GPU, both used here as OpenCL devices, and we have consistently checked the correctness of our hybrid computations.

For the data-parallel deployment, we also need to determine the optimal ratio between the parts of the domain that will be computed by the CPU and by the GPU. At first, we had chosen the theoretical peak performance ratio of the CPU and the GPU, but the use of the profiler has shown an imbalance in the execution times. We have then chosen the ratio of the actual performance of the CPU-only and of the GPU-only vectorized (complete) kernels: these have been confirmed empirically as nearly optimal, by running several performance tests with various ratios. For the memory-bound case, we have thus a 0.8 ratio for the GPU, and 0.95 in the compute-bound case.

According to the AMD profiler, executing a kernel on the CPU used as a device preempts the CPU and hence prevents the CPU host from enqueuing GPU kernel executions. A possible solution would have been the OpenCL device fission of the CPU, separating it into different devices, keeping one core for the host and executing the kernel on the remaining three. In our case, enqueuing the GPU kernel first was adequate, as we have to synchronize at each iteration and cannot enqueue more than one GPU kernel at a time.

We now compare the APU hybrid performance (for stencils of size 4) with an execution on the APU integrated GPU only with buffers allocated in z memory.

For the memory-bound case (see Fig. 9a), we see a better performance in the task-parallel deployment on smaller domains, but this drops when the domain gets larger and the GPU occupancy rises. When the GPU is full enough, the CPU will not perform fast enough to keep up, and overall performance will decrease. The data-parallel deployment performs better when the domain gets larger, as the relative cost of synchronization gets lower when computation time increases. We obtain a 20 to 30% better performance compared to GPU only (with buffers in *z* memory).

For the compute-bound case (see Fig. 9b), the task-parallel deployment is slowed down by the CPU's lower compute power. Even for the data-parallel deployment, the cost of synchronization is too high due to the imbalance in the ratio between the CPU and the GPU. This may be due to the lack of optimization of the source code for the CPU: whereas its compute power is 17% of the APU, its kernel performance is only 5% of that of the integrated GPU.

Finally, it has to be noticed that the GPU-only performance (with buffers in *z* memory) is here lower than the GPU-only performance (with buffers in *g* memory) presented in Figs. 6a and 8a. *g* memory offers indeed a better bandwidth than *z* memory on current APUs. However, those previous results with *g* memory do not take into account the cost of the snapshotting necessary to applications such as the RTM. In [6], *g* memory has been shown to perform currently better than *z* memory even when the snapshotting frequency is high, but future generations of APUs will significantly improve the bandwidth of such zero-copy (*z*) memory.

6. CONCLUSIONS

In this paper, we studied how to use both the CPU and the GPU of the APU for stencil computations. We provided a strategy for hybrid deployments that takes into account the compute-bound or memory-bound nature of the application and its amount of divergence. We proposed two possible deployments, a task-parallel one and a data-parallel one, and balanced the use of the CPU and the integrated GPU in order to exploit at best the APU. When considering a classic memory-bound stencil, we obtained an up to 30% performance gain compared to a GPU only deployment.

Our strategy seems to be valid in the memory-bound case, but requires further investigation for the compute-bound case on the APU. More precisely, we are currently developping a specific OpenCL kernel for the CPU: this kernel should better exploit the CPU caches as well as the CPU *prefetch* feature, and hence offer much better performance. In the future, we will first try to confirm the validity of our strategy on the complete RTM, as well as on various applications. We also plan to apply this strategy on other hybrid hardware, such as the Intel multicore CPUs with integrated GPUs or the future Denver architecture from NVIDIA with both GPU and ARM CPU on the same chip.

7. ACKNOWLEDGEMENT

The authors would like to thank Total for funding this work, as well as AMD for providing us with the surveyed hardware.

8. REFERENCES

[1] R. Abdelkhalek, H. Calandra, O. Coulaud, J. Roman, and G. Latu. Fast seismic modeling and reverse time migration on a GPU cluster. In *International Conference on High Performance Computing Simulation, 2009. HPCS '09*, pages 36–43, 2009.

[2] AMD. Accelerated parallel processing OpenCL programming guide, May 2012.

[3] E. Baysal, D. D. Kosloff, and J. W. C. Sherwood. Reverse time migration. *Geophysics*, 48(11):1514–1524, 1983.

[4] J.-P. Berenger. A perfectly matched layer for the absorption of electromagnetic waves. *Journal of Computational Physics*, 114(2):185–200, 1994.

[5] J. Cabezas, M. Araya-Polo, I. Gelado, N. Navarro, E. Morancho, and J. Cela. High-performance reverse time migration on gpu. In *Chilean Computer Science Society (SCCC), 2009 International Conference of the*, pages 77–86, 2009.

[6] H. Calandra, R. Dolbeau, P. Fortin, J.-L. Lamotte, and I. Said. Evaluation of successive CPUs/APUs/GPUs based on an OpenCL finite difference stencil. In *2013 21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 405–409, 2013.

[7] H. Calandra, R. Dolbeau, P. Fortin, J.-L. Lamotte, and I. Said. Forward seismic modeling on AMD accelerated processing unit. In *Rice Oil & Gas HPC Workshop*, 2013.

[8] G. Cocco and A. Cisternino. Device specialization in heterogeneous multi-GPU environments. In *2012 Imperial College Computing Student Workshop*, pages 35–41, 2012.

[9] M. Daga, A. M. Aji, and W. chun Feng. On the Efficacy of a Fused CPU+GPU Processor (or APU) for Parallel Computing. In *Symposium on Application Accelerators in High-Performance Computing*, 2011.

[10] M. C. Delorme, T. S. Abdelrahman, and C. Zhao. Parallel radix sort on the AMD fusion accelerated processing unit. In *International Conference on Parallel Processing, 2013. ICPP 2013*, pages 339–348, 2013.

[11] Khronos Group. The OpenCL Specification version 1.2, November 2012.

[12] D. Michéa and D. Komatitsch. Accelerating a three-dimensional finite-difference wave propagation code using GPU graphics cards. 182(1):389–402, 2010.

[13] P. Micikevicius. 3D finite difference computation on GPUs using CUDA. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, page 79–84, New York, NY, USA, 2009. ACM.

[14] J. I. Toivanen, T. P. Stefanski, N. Kuster, and N. Chavannes. Comparison of CPML implementations for the GPU-accelerated FDTD solver. *Progress in Electromagnetics Research M*, 19:61–75, 2011.

[15] K. Yoon, S. Suh, J. Ji, J. Cai, and B. Wang. Stability and speedup issues in TTI RTM implementation. In *SEG Technical Program Expanded Abstracts 2010*, pages 3221–3225, 2010.