Publishing House
AKAPIT

COMPUTER METHODS IN MATERIALS SCIENCE
Informatyka w Technologii Materiałów

Vol. 15, 2015, No. 2

AMD APU SYSTEMS AS A PLATFORM FOR SCIENTIFIC COMPUTING

FILIP KRUŻEL[1]*, KRZYSZTOF BANAŚ[2]

[1]Cracow University of Technology, Warszawska 24, 31-155 Kraków, Poland
[2]AGH University of Science and Technology, al. Mickiewicza 30, 30-059, Kraków
**Corresponding author: fkruzel@pk.edu.pl*

Abstract

In our current work we investigate the possibility of using modern AMD APU architecture in scientific and technical computing. The architecture combines both a CPU and a GPU in a single Accelerated Processing Unit, which theoretically allows for shortening the time of exchanging the data between the two hardware units. This capability solves the problem of performance bottleneck related to the exchange of data between the CPU and GPU memory. Due to the structure of this architecture, it can be considered as a natural evolution of the concept presented in the IBM PowerXCell processors that have been tested during our past research (Krużel & Banaś, 2013). As reference systems we use both a system based on similar AMD architecture and a specialized Nvidia Tesla Accelerator card. Moreover, due to comparable characteristics of the CPU and GPU parts of APU we have run our computations on both hardware units separately to see the difference in performance. For testing we used our previously developed finite element numerical integration algorithm implemented in OpenCL programming framework. This algorithm has been tested with various organizations of memory and computing techniques to fully check the hardware capabilities of the APU architecture, both in terms of data exchange and calculations acceleration. Our research brings an answer to the question whether this architecture is the right future for scientific computing and whether in the next few years will be able to play a significant role in many areas of computational science.

Key words: AMD, Accelerated Processing Unit, APU, OpenCL, Heterogeneous System Architecture, Finite Element Method, Numerical Integration

1. INTRODUCTION

In recent years we can notice great changes in computer systems architectures. From the large number of new technologies we can observe the two main trends. One is connected with the increasing number of computing cores and can be easily seen in the new powerful graphic cards and also general purpose CPUs. The second trend is connected with the miniaturization of the specialized hardware used in modern smartphones, tablets or laptop computers.

Increasing computing power of this types of hardware is associated with the need to minimize the energy consumed while maintaining high efficiency. At the junction of these two trends there can be observed very interesting designs that combine multiple specialized cores with fewer cores for general use. With the use of such a combined architecture we can save the energy when using normal cores and maintain very high computing power during more complicated tasks. Besides energy efficiency another advantage of using such architectures is that they are equipped with a new technologies for transfer of data between different types of cores and the memory. Hence, there is a possibility of using these architectures for scientific computing with the hope of efficient use of the fast inter-core and memory buses.

ISSN 1641-8581

With the introduction of different systems with integrated graphic and general purpose cores on one chip there appear a need for handling the access to the RAM. The first interesting system with the full unification of special purpose and normal cores was the IBM Cell processor used in Playstation 3. The architecture was equipped with very fast element interconnect bus (EiB) and Direct Memory Access (DMA) for the specialized Synergistic Processor Units (SPU). The architecture allows for a very fast memory to SPU data transfers and efficient data exchange between the normal IBM Power cores. These capabilities, combined with energy efficiency, resulted in a wide use of this architecture in scientific computing. An example of such usage was the Roadrunner cluster, which was the first computing system that reached the petaflop performance barrier (Barker et al., 2008).

In our previous work we investigated the possibility of using this architecture for a specific parts of the Finite Element Method computations (Krużel & Banaś, 2010, 2013). Despite its high performance. the architecture was lacking a convenient programming model - there was a possibility to make correct use of the DMA and to use the same areas of memory for exchanging the data with the SPU and CPU (Heirich & Bavoil, 2007), but it required using a complicated programming model and proper memory alignment. This could be avoided by the use of the OpenCL programming model, but with it we cannot properly use all the advantages of unified memory.

The problem with exchanging data between different types of computing cores plays a significant role in all heterogeneous architectures. Hence, several different strategies appeared both in software and hardware layers. In June 2012 some of the main players in heterogeneous computing, including AMD and ARM, formed Heterogeneous System Architecture Foundation that tries to develop common technologies for handling computations on cores of different types (HSA Foundation, 2013). Cooperation resulted in developing heterogeneous Unified Memory Model (hUMA) which was first introduced in AMD Accelerated Unit processor unit described in more details in the later part of this article (Kyriazis, 2012).

Almost simultaneously with developing hUMA model, Khronos Group presented OpenCL 2.0 specification with Shared Virtual Memory capabilities, that unifies the CPU and GPU memory on the programmer side and, on HSA systems, can use hardware capabilities of hUMA (Howes & Munshi, 2014). In the meanwhile, Intel developed his new integrated graphic architecture codenamed GT3 which can be optionally equipped with embedded DRAM (eDRAM) memory. With the OpenCL 2.0 this can be used as a shared memory for both CPU and GPU cores (Graczyk, 2013). In opposite, Nvidia with Maxwell GPUs and CUDA 6.0 had presented Unified Virtual Memory model which is a direct answer to their main competitor AMD hUMA. Unfortunately this solution is still based mostly on software capabilities and it is generally developed for easing the GPU programming model (Landaverde et al., 2014). Despite that, during the Supercomputing 14 conference in November of 2014 Nvidia presented new Pascal GPU architecture equipped with NVlink - new connector for communication between CPU and GPU and between different GPUs (NVIDIA, 2013). All these technologies are developed for the purpose of reducing the performance bottleneck in exchanging the data between different types of computing cores. Although they are developed mostly for gaming purposes, they also can be useful for scientific computing. Hence, we have decided to test and compare new Heterogeneous System Architecture capabilities present in the AMD Accelerated Processing Unit A10-7850 codenamed 'Kaveri'.

2. ADVANCED MICRO DEVICES ACCELERATED PROCESSING UNIT

AMD APU is an architecture that combines the computing power of SIMD GPU cores and a versatility of classical CPU cores. It has evolved from the AMD Fusion project which tried to introduce more integration between GPU and CPU architectures. As a result of this work the first generation of Accelerated Processing Units was introduced in 2011(Van Winkle, 2012). In 2012 AMD changed the codename of their architecture to Heterogeneous System Architecture and along with several other companies established the HSA Foundation. The main aim of the foundation is to develop and define features and interfaces for various types of computer processors, including CPUs, graphics processors and DSPs, as well as the memory systems that connect them. It also sets up the goal of making the programming model of the HSA easy and transferable between the different types of devices (HSA Foundation, 2013).

With the third generation 'Kaveri' APUs the GPU and CPU memories were fully integrated, enabling the possibility of passing pointers between CPU and GPU cores and using whole virtual memory address space by GPU. This memory model is called heterogeneous Unified Memory Access (hUMA), opposite to usually used by accelerators Non-Uniform Memory Access (NUMA) model. Additionally HSA establishes cache coherency between CPU and GPU, which eliminates the need to do a DMA flush every time the programmer wants to move data between CPU and GPU (Kyriazis, 2012). Moreover, all computing cores have equal flexibility to create and dispatch work, in opposition to normal CPU/GPU interaction where CPU through a kernel and drivers dispatch tasks for GPU. This technique is called Heterogeneous Queuing (hQ).With that capability a given application can generate task queues directly on the GPU without the CPU getting involved. Additionally the GPU can also generate its own workload just like the CPU. Equally, with the use of hQ the CPU can push work into the GPU task queue without the use of operating system. This create a bi-directional queuing system which dramatically reduces latency and allows applications to easily push jobs to whichever processor is most appropriate. With this hardware features HSA should be very similar in programming to normal CPUs (Halfacree, 2013). Figure 1 shows all new features of the AMD 'Kaveri' APU.

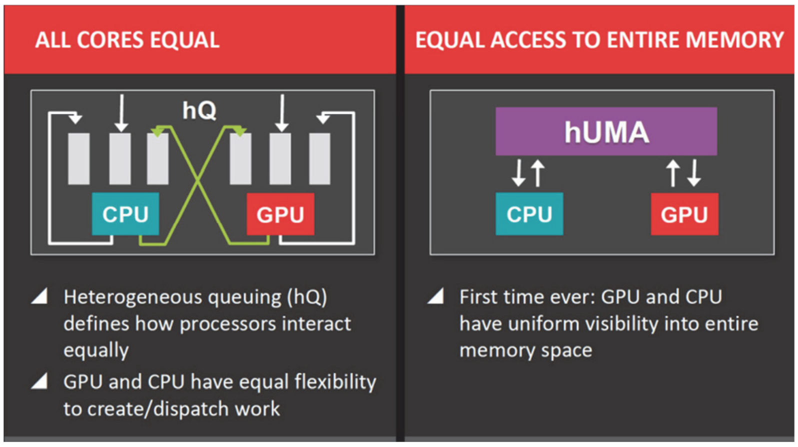

ALL CORES EQUAL
hQ
CPU
GPU
Heterogeneous queuing (hQ) defines how processors interact equally
GPU and CPU have equal flexibility to create/dispatch work
EQUAL ACCESS TO ENTIRE MEMORY
hUMA
CPU
GPU
First time ever: GPU and CPU have uniform visibility into entire memory space

***Fig. 1.** AMD APU Features (AMD, 2013)*

The Accelerated Processing Unit possessed by us is equipped with 12 compute cores - 4 CPU and 8 GPU. CPU cores are a brand new x86-compatible AMD architecture codenamed Steamroller which runs at 3,7 GHz (with Turbo-boost technology to 4.0 GHz) and are equipped with 4MB of L2 cache. GPU cores are cores in GCN (Graphic Core Next) technology and are equivalent to Radeon R7 cores with 720MHz of cores speed. We decided to test the main HSA features on this unit using new OpenCL 2.0 Shared Virtual Memory capabilities.

3. OPENCL 2.0

Among the many programming methods used in accelerators programming OpenCL stands out as a very versatile method. It supports many modern architectures, including GPUs, coprocessors (eg. Xeon Phi) or CellBE. OpenCL is gradually developed since 2009 and its current 2.0 version was introduced in 2014 bringing out several important improvements. In our previous works we focused on porting our FEM algorithms to CellBE (Krużel & Banaś, 2010, 2013), GPU (Banaś et al., 2014; Banaś & Krużel, 2014) and Xeon Phi (Krużel & Banaś, 2014).

OpenCL memory model defines several types of memory regions available for programmers to use. Because of the origins of the model in GPU programming, the memory model is based on physical organization of typical GPU memory. Due to the OpenCL portability each of the memory objects can be mapped differently, depending upon the available hardware resources. Variables defined inside the kernel belong to private memory and can be stored in scalar or vector registers. The other memory regions are assigned through specific qualifiers. OpenCL defines three types of memory – global, constant and shared (local). Global memory stores variables that are visible to all threads executing the kernel, constant memory is also available for all threads but it is only accessible for reading. The fastest local memory stores the variables shared by threads in a single work-group.

Because of the portability of created code, OpenCL contains procedures that allows for adapting to different platforms and devices even without physical equivalents of specific memory types (Rul et al., 2010). So far, typical behavior in OpenCL computing was copying the data necessary for calculations from the host memory to the device global or constant memory for further use. It required proper preparations of the memory buffers on the host side from the programmer side and a lot of hardware or system solutions for copying the data from the host to the accelerator. Even for the integrated solutions with CPU and Accelerator on one die (eg. CellBE, APU) the memory was partitioned into host and device parts which prevents zero-copy operations and takes precious time. Despite software and hardware solutions for passing the data from the host to device and back, the whole procedure was very complicated and time-consuming.

The new feature of Shared Virtual Memory (SVM) presented in OpenCL 2.0 solves both problems with programming OpenCL memory model. From the programmer side it reduces the necessary preparations on data and the whole mapping and passing pointers operations that takes a lot of code lines. From the performance point of view, on the HAS compatible hardware it is using the full potential of hUMA, which means that there is absolutely no copying between the CPU and the accelerator. Of course if the device is not HSA compatible the framework will take care of the data passing in the best possible way. Shared Virtual Memory defines a buffer that can be used directly both by a host and an accelerator without unnecessary mappings and copying the data (figure 2).

Detailed information about SVM OpenCL features can be found in (Intel, 2014; Howes & Munshi, 2014). AMD APU A10-7850 tested by us is fully compatible with Heterogeneous System Architecture, the reason why we decided to test our previously ported to OpenCL numerical integration algorithm on it.

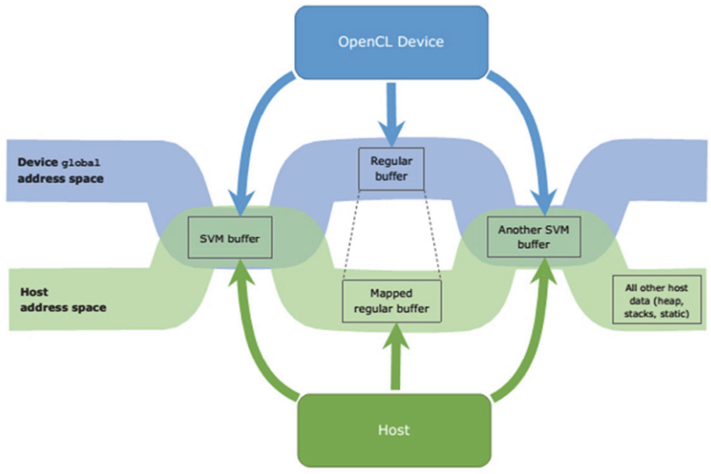


Fig. 2. *OpenCL Shared Virtual Memory (Intel, 2014)*

4. FINITE ELEMENT NUMERICAL INTEGRATION

Numerical integration algorithm forms an important part of almost every Finite Element Method (FEM) code. The FEM solution process includes the procedure of creation of the system of linear equations, where individual entries of the system matrix are obtained through integration of the terms from the weak statement, specific to the problem solved by the FEM. Thanks to the fundamental to FEM, division of the whole computational domain into finite elements, the integrals can be obtained as sums of local, element contributions. The whole system matrix (the stiffness matrix) is obtained by assembling element stiffness matrices, with the entries corresponding to pairs of element basis functions. The element stiffness matrices can be computed separately, making the problem of finite element numerical integration embarrassingly parallel.

The final formula to calculate each entry of the element stiffness matrix depends on the formulation of the problem and in our paper we accept its form as (1)

$$A^e_{i_S\, j_S} = \int\limits_{\Omega_e} \sum_{i_Q i_D} C^{i_Q i_D} \frac{\partial \phi_{i_S}}{\partial x_{i_D}} \frac{\partial \phi_{j_S}}{\partial x_{j_D}} d\Omega \qquad (1)$$

After the change of variables, from the real to the reference element, and applying the numerical quadrature the formula above can be transformed into the sum (2)

$$A^e_{i_S\, j_S} = \sum_{i_Q}^{N_Q} \sum_{i_S, j_S}^{N_S} \sum_{i_D, j_D}^{N_D} C^{i_Q i_D j_D} \times \psi^{i_Q i_D i_S} \psi^{j_Q j_D j_S} vol^{i_Q} \qquad (2)$$

Where C is an array of a problem dependent coefficients, each ψ are global derivatives of element shape functions and vol^{iQ} is the determinant of Jacobian transformation matrix multiplied by a Gaussian quadrature weights. All indices are denoted by i with the corresponding subscripts: Q – Gaussian points, S – shape functions, D – different spatial derivatives for test and trial function. The corresponding formula for the right hand side vector (the load vector) is calculated using (3)

$$b^e_{iS} = \sum_{i_Q}^{N_Q} \sum_{i_S}^{N_S} \sum_{i_D}^{N_D} D^{i_Q i_D} \psi^{i_Q i_D i_S} vol^{i_Q} \qquad (3)$$

```
1:  read quadrature data $\xi_Q$ and weights $w_Q$ for the reference element of particular
    type.
2:  for $e=1$ to $N_e$ do
3:    read problem dependent coefficients common for all integration points (e.g.
      material data, previous iterations (or time steps) degrees of freedom etc.)
4:    read element geometry data
5:    initialize element stiffness matrix $A^e_{i_S j_S}$ and element load vector $b^e_{i_S}$
6:    for $i_Q=1$ to $N_Q$ do
7:      read or calculate(on a basis of the coordinates of integration points) the
        values of shape functions and their derivatives with respect to their local
        coordinates for a given integration point.
8:      read or calculate Jacobian matrix, its determinant and inverse.
9:      calculate $vol^{i_Q}$
10:     using the Jacobian matrix calculate the derivatives of shape functions with
        respect to the global coordinates for a given integration point
11:     based on the values of unknowns obtained through the use of shape
        functions derivatives calculate the $C^{i_Q}$ and $D^{i_Q}$ coefficients for a given
        quadrature point
12:     for $i_S=1$ to $N_S$ do
13:       for $j_S=1$ to $N_S$ do
14:         for $i_D=0$ to $N_D$ do
15:           for $j_D=0$ to $N_D$ do
16:             $A^e[i_S][j_S]$ +=$C[i_Q][i_D][j_D]\times\psi[i_Q][i_S][i_D]\times\psi[j_Q][j_S][j_D]\times vol[i_Q]$
17:           end for ($j_D$)
18:         end for ($i_D$)
19:         if $i_S=j_S$ then
20:           for $i_D=0$ to $N_D$ do
21:             $b^e[i_S]$+=$D[i_Q][i_D]\times\psi[i_Q][i_S][i_D]\times vol[i_Q]$
22:       end for ($j_S$)
23:     end for ($i_S$)
24:   end for ($i_Q$)
25: end for ($e$)
```

Fig. 3. *FEM numerical integration algorithm*

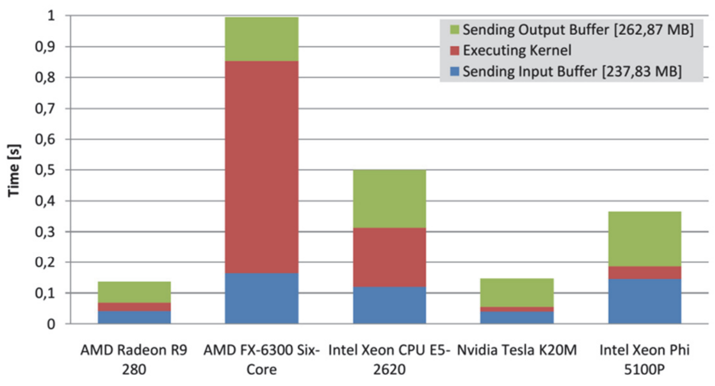


Fig. 4. *Execution time of the numerical integration algorithm for OpenCL 1.2 and several tested reference platforms*

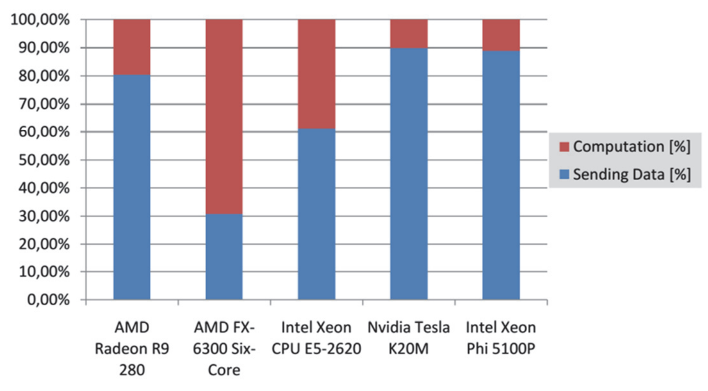


Fig. 5. *Percentage of time spent on data transfer and calculations during numerical integration for OpenCL 1.2 and several reference platforms*

As a consequence we can present a generic numerical integration algorithm as follows on figure 3.

In our previous works we focused on increasing performance of the algorithm by changing the order of the loops and the way of storing the variables in different types of memory, but in all methods there was always a delay connected with the data transfer between the host and device global memory. With the introduction of HSA there is a chance that the accelerators can improve the total time of execution of the program by eliminating this delay.

5. NUMERICAL INTEGRATION ON APUS

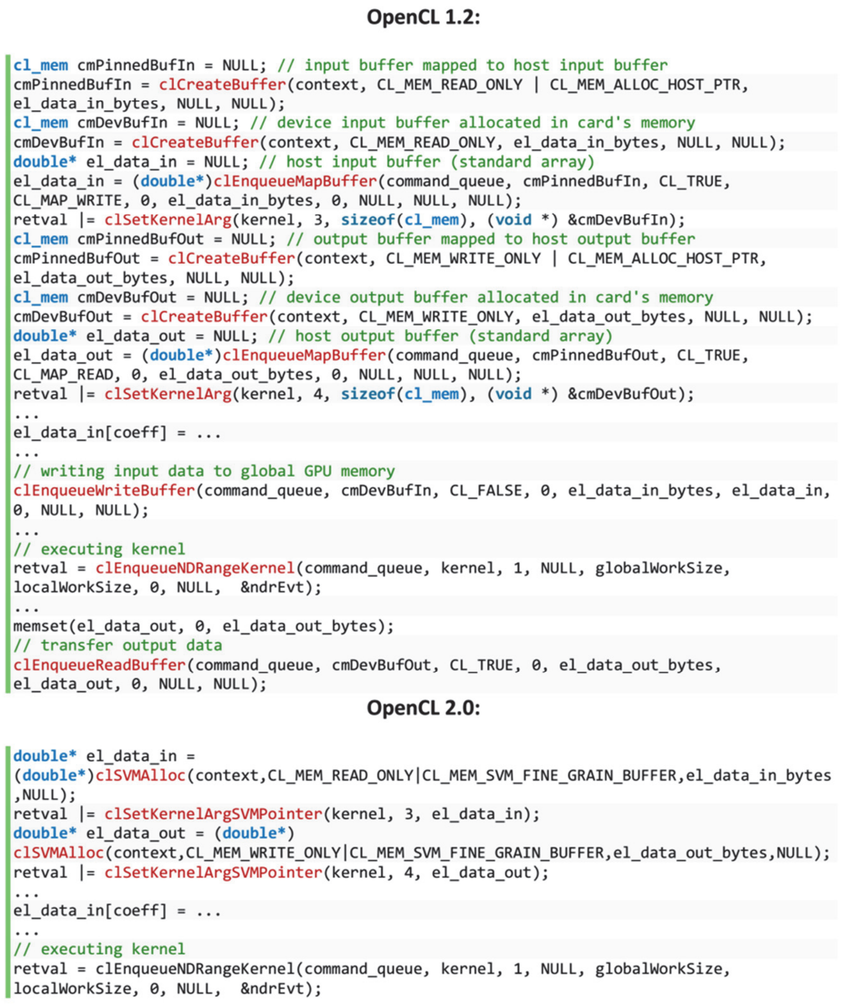
For testing the full potential of HSA features of a new AMD Accelerated Processing Unit we used numerical integration algorithm to solve a convection-diffusion problem which is memory and time consuming. This algorithm is a part of our ModFEM computational framework which was presented in details in (Michalik et al., 2013). In the example, computational domain is triangulated using prismatic elements with 6 degrees of freedom. The algorithm was tested on several different architectures with OpenCL properties shown in table 1.

As we can notice from the table 1, AMD APU is visible in OpenCL as two devices with separate memories which can indicate that despite having full HSA compatibility the RAM memory of a computer system is still partitioned into two parts. The results of execution of numerical integration algorithm in the most efficient versions for the five architectures that we compare with AMD APU can be seen in figure 4.

From this charts we can see that the transfer of data from and to device takes significant part of the whole execution time. This is even more strongly visible for architectures developed strictly for computational purposes like Nvidia Tesla or Intel Xeon Phi on which almost 90% of the time is wasted on data transfers. Figure 5 shows the percentage of data transfer time in relation to the total computation time.

Table 1. *OpenCL properties for tested platforms and numerical integration algorithm*

OpenCL device	Xeon Phi 5110P	NVidia Tesla K20m	AMD Radeon R9 280	AMD FX-6300 CPU	Intel Xeon E5-2620 CPU	AMD APU Radeon R7	AMD APU A10 Steamroller CPU
Number of compute units	236	13	32	6	24	8	4
Number of cores per CU	1/4	192	64	1	1/2	32	1
Total number of cores	59	2496	2048	6	12	512	4
Device memory size [MB]	5773	4800	2932	7773	32083	1951	14880
Local memory size [kB]	32	48	32	32	32	32	32
Number of elements to compute	785408	785408	786432	782352	782400	782336	782336
Number of work groups	944	104	256	12	48	64	8
Work group size	16	64	64	4	4	64	4
Number of threads	15104	6656	16384	48	192	4096	32



OpenCL 1.2:

```
cl_mem cmPinnedBufIn = NULL; // input buffer mapped to host input buffer
cmPinnedBufIn = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_ALLOC_HOST_PTR,
el_data_in_bytes, NULL, NULL);
cl_mem cmDevBufIn = NULL; // device input buffer allocated in card's memory
cmDevBufIn = clCreateBuffer(context, CL_MEM_READ_ONLY, el_data_in_bytes, NULL, NULL);
double* el_data_in = NULL; // host input buffer (standard array)
el_data_in = (double*)clEnqueueMapBuffer(command_queue, cmPinnedBufIn, CL_TRUE,
CL_MAP_WRITE, 0, el_data_in_bytes, 0, NULL, NULL, NULL);
retval |= clSetKernelArg(kernel, 3, sizeof(cl_mem), (void *) &cmDevBufIn);
cl_mem cmPinnedBufOut = NULL; // output buffer mapped to host output buffer
cmPinnedBufOut = clCreateBuffer(context, CL_MEM_WRITE_ONLY | CL_MEM_ALLOC_HOST_PTR,
el_data_out_bytes, NULL, NULL);
cl_mem cmDevBufOut = NULL; // device output buffer allocated in card's memory
cmDevBufOut = clCreateBuffer(context, CL_MEM_WRITE_ONLY, el_data_out_bytes, NULL, NULL);
double* el_data_out = NULL; // host output buffer (standard array)
el_data_out = (double*)clEnqueueMapBuffer(command_queue, cmPinnedBufOut, CL_TRUE,
CL_MAP_READ, 0, el_data_out_bytes, 0, NULL, NULL, NULL);
retval |= clSetKernelArg(kernel, 4, sizeof(cl_mem), (void *) &cmDevBufOut);
...
el_data_in[coeff] = ...
...
// writing input data to global GPU memory
clEnqueueWriteBuffer(command_queue, cmDevBufIn, CL_FALSE, 0, el_data_in_bytes, el_data_in,
0, NULL, NULL);
...
// executing kernel
retval = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL, globalWorkSize,
localWorkSize, 0, NULL,  &ndrEvt);
...
memset(el_data_out, 0, el_data_out_bytes);
// transfer output data
clEnqueueReadBuffer(command_queue, cmDevBufOut, CL_TRUE, 0, el_data_out_bytes,
el_data_out, 0, NULL, NULL);
```

OpenCL 2.0:

```
double* el_data_in =
(double*)clSVMAlloc(context,CL_MEM_READ_ONLY|CL_MEM_SVM_FINE_GRAIN_BUFFER,el_data_in_bytes
,NULL);
retval |= clSetKernelArgSVMPointer(kernel, 3, el_data_in);
double* el_data_out = (double*)
clSVMAlloc(context,CL_MEM_WRITE_ONLY|CL_MEM_SVM_FINE_GRAIN_BUFFER,el_data_out_bytes,NULL);
retval |= clSetKernelArgSVMPointer(kernel, 4, el_data_out);
...
el_data_in[coeff] = ...
...
// executing kernel
retval = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL, globalWorkSize,
localWorkSize, 0, NULL,  &ndrEvt);
```

Fig. 6. *Shortening of the original host code for data transfer between the host and OpenCL device memories due to using OpenCL 2.0 SVM*

The results indicate that architectures like HSA can be the answer to the problem of transferring data between CPU and accelerators. Hence, in order to verify that conjecture, we changed our code to use OpenCL 2.0 Shared Virtual Memory buffers. The first improvement is clearly visible in the code fragments presented in figure 6.

To check the performance improvement between the normal OpenCL code with data transfer between the CPU and GPU memory and the new SVM model we have first tested our old OpenCL 1.2 implementation of numerical integration on the AMD APU system. Obtained results can be seen in figure 7.

As we can see, for normal OpenCL 1.2 implementation of the examined algorithm, transfer of input and output data takes about 30% of the execution time (figure 8). This is similar to the results for AMD FX-6300 and lower than for other tested platforms (see figure 5).

For comparison with these results we have tested our new Shared Virtual Memory implementation of the numerical integration algorithm which was designed to take the full advantage of the new Heterogeneous System Architecture features. Obtained results are presented in figure 9, where the time for the management of input and output data, done implicitly by the execution environment, is included in the execution time.

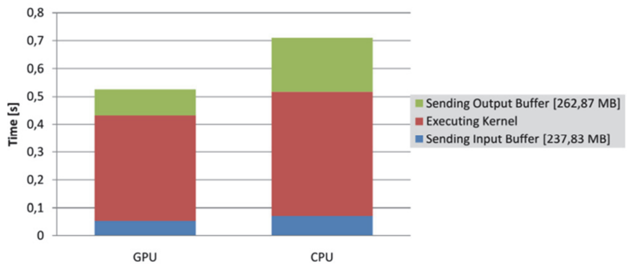


Fig. 7. *Execution time of the numerical integration algorithm for OpenCL 1.2 on each separate part of the APU A10-7850*

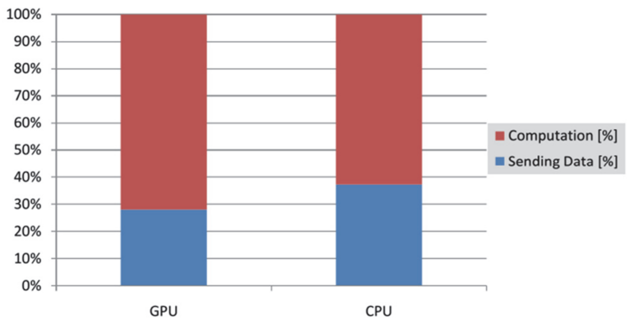


Fig. 8. *Percentage of time spent on data transfer and calculations during numerical integration for OpenCL 1.2 on each separate part of the APU A10-7850*

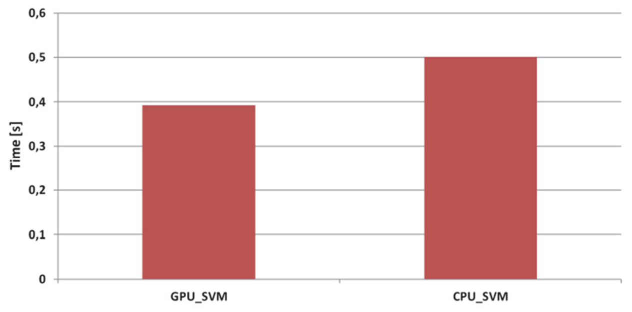


Fig. 9. *Execution time of the numerical integration algorithm for OpenCL 2.0 and SVM on each separate part of the APU A10-7850*

The detailed performance results clearly show significant improvement of the total execution time with the several interesting findings:

– Computation time on GPU part of APU for OpenCL 2.0 and SVM (that includes implicit time for memory management) takes only 3,55% more time than pure computation time for OpenCL 1.2
– Computation time on CPU part of APU for OpenCL 2.0 and SVM (that includes implicit time for memory management) takes only 12,18% more time than pure computation time for OpenCL 1.2
– On GPU part of APU there is 25,41% overall speedup for OpenCL 2.0 and SVM as compared with OpenCL 1.2

On CPU part of APU there is 29,50% overall speedup for OpenCL 2.0 and SVM as compared with OpenCL 1.2.

It can be stated that despite the fact that there is some overhead connected with the proper use of Shared Virtual Memory, the overall speed-up can be considered satisfactory.

6. CONCLUSIONS

Concluding, we can state that AMD APU architecture can significantly speed up overall computation time of numerical integration by reducing the exchange of data between host and accelerator. The overhead related to input and output data transfer forms one of the primary performance bottlenecks, not only in our example case of numerical integration, but for many scientific computing programs ported to GPUs. The solutions that use SVM mechanism should improve execution times for these codes. Moreover, the simplified method of transferring data to the accelerator is convenient for developers and encourage broader use of GPU computing power in the newly created software. All this leads to the observation that the architectures based on HSA can be a promising path for the development of scientific applications that use accelerators or special purpose cores.

ACKNOWLEDGEMENT

Financial assistance of the NCN project 2011/01/B/ST6/00674 is acknowledged.

REFERENCES

AMD, AMD Developer Summit 2013, San Jose Convention Center.

Banaś, K., Krużel, F., 2014, OpenCL Performance Portability for Xeon Phi Coprocessor and NVIDIA GPUs: A Case Study of Finite Element Numerical Integration, *Lecture Notes in Computer Science*, 8806, 158-169.

Banaś, K., Płaszewski, P., Macioł, P., 2014, Numerical integration on GPUs for higher order finite elements, *Computers & Mathematics with Applications*, 67 (6), 1319-1344.

Barker, K. J., Davis K., Hoisie, A., Kerbyson, D. K., Lang, M., Pakin, S., Sancho J. C., 2008, Entering the petaflop era: The architecture and performance of Roadrunner, *High Performance Computing, Networking, Storage and Analysis*, 1-11.

Gaster, B., Kaeli, D., Howes, L., Mistry, P., Schaa, D., 2011, *Heterogeneous Computing With OpenCL*, Elsevier Science & Technology.

Graczyk, R., Intel Iris Pro 5200 – test; Crysis 3 na integrze?, available online at: http://pclab.pl/art54267.html, *PClab.pl digital community*, 2013, [Accessed February 10, 2015].

Halfacree, G., , 2013, AMD announces Heterogeneous Queuing tech, available online at: bit-tech http://www.bittech.net/news/hardware/2013/10/22/amdhq/1,*bit-tech* [Accessed February 10, 2015].

Heirich, A., Bavoil L., 2006, Deferred Pixel Shading on the PLAYSTATION 3, available online at: http://research.scea.com/ps3_deferred_shading.pdf, *Sony Computer Entertainment US Research & Development*, [Accessed February 10, 2015].

Howes, L., Munshi, A., 2014, The OpenCL Specification, Khronos OpenCLWorking Group, version 2.0, revision 26.

HSA Foundation, [online] http://www.hsafoundation.com, 2013, [Accessed February 10, 2015].

Intel, OpenCL 2.0 Shared Virtual Memory Overview, Intel, 2014.

Krużel, F., Banaś, K., 2010, Finite element numerical integration on PowerXCell processors, *Lecture Notes in Computer Science*, 6067, 517-524.

Krużel, F., Banaś, K., 2014, Finite Element Numerical Integration on Xeon Phi coprocessor, *Annals of Computer Science and Information Systems*, 2, 603-612.

Krużel, F., Banaś, K., 2013, Vectorized OpenCL implementation of numerical integration for higher order finite elements, *Computers & Mathematics with Applications*, 66(10), 2030-2044.

Kyriazis, G., 2012, Heterogeneous System Architecture: A Technical Review, AMD, revision 1.0.

Landaverde, R., Zhang, T., Coskun, A.K., Herbordt, M., 2014, An Investigation of Unified Memory Access Performance in CUDA, *IEEE High Performance Extreme Computing*.

Michalik, K., Bana´s, K., Płaszewski, P., Cybułka, P., ModFem 2013, A computational framework for parallel adaptive finite element simulations, *Computer Methods in Materials Science*, 13 (1), 3-8.

NVIDIA, CUDA C Programming Guide Design Guide, version 6.5, August 2014.

NVIDIA, NVIDIA NVLink High-Speed Interconnect: Application Performance, Whitepaper, 2013.

Rul, S., Vandierendonck, H., D' Haene J., De Bosschere, K., 2010, An experimental study on performance portability of OpenCL kernels, *Application Accelerators in High Performance Computing*, 2010 Symposium, Knoxville, TN, USA, 3.

Van Winkle, W., 2012, AMD Fusion: How It Started, Where It's Going, And What It Means, available online at: http://www.tomshardware.com/reviews/fusionhsa-opencl-history,3262-12.html, tom's Hardware, [Accessed February 10, 2015].

SYSTEMY AMD APU JAKO PLATFORMA OBLICZEŃ NAUKOWO-TECHNICZNYCH

Streszczenie

W naszej obecnej pracy badamy możliwość wykorzystania nowoczesnej architektury AMD APU do wykonywania obliczeń naukowo-technicznych. Architektura ta łączy w sobie jednostki CPU i GPU w pojedynczym APU (Accelerated Processing Unit), co teoretycznie pozwala na przyspieszenie czasu wymiany danych pomiędzy poszczególnymi jednostkami obliczeniowymi. Możliwość ta rozwiązuje problem „wąskiego gardła", który związany jest z wymianą danych pomiędzy pamięciami CPU i GPU. Ze względu na budowę architekturę tę można uznać za naturalną ewolucję rozwiązania zaprezentowanego w procesorach IBM Power XCell, które były przez nas badane wcześniej (Krużel & Banaś, 2013). W celu porównania uzyskanych wyników użyliśmy zarówno systemu opartego na podobnej architekturze AMD, jak i systemu wyposażonego w specjalistyczną kartę Nvidia Tesla. Ponadto, ze względu na porównywalne cechy CPU i GPU wbudowanych w APU przeprowadziliśmy nasze obliczenia dla każdej z części oddzielnie, aby zobaczyć różnicę pomiędzy obliczeniami na CPU a GPU w tak zintegrowanym układzie. Do testów użyliśmy opracowanego przez nas wcześniej algorytmu całkowania numerycznego zaimplementowanego w środowisku programistycznym OpenCL. Algorytm ten został przetestowany z różnymi opcjami organizacji pamięci i obliczeń, aby w pełni sprawdzić-możliwości sprzętowe architektury APU, zarówno w zakresie wymiany danych, jak i przyśpieszenia obliczeń. Wynikiem pozytywnych rezultatów naszych badań jest stwierdzenie, że nowoczesne architektury AMD APU są przyszłościowe w kontekście obliczeń naukowych i wnastępnych latach będą mogły odgrywać znaczącą rolę w dziedzinie przyspieszania obliczeń.

Received: February 15, 2015
Received in a revised form: April 2, 2015
Accepted: April 24, 2015