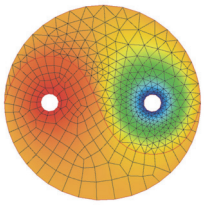
Publishing House
AKAPIT

COMPUTER METHODS IN MATERIALS SCIENCE
Informatyka w Technologii Materiałów

Vol. 17, 2017, No. 2



EFFICIENT TRANSFER OF C++ OBJECTS ON INTEL XEON PHI COPROCESSOR IN OFFLOAD MODE

VÍT HANOUSEK* TOMÁŠ OBERHUBER

Department of Mathematics, Faculty of Nuclear Sciences and Physical Engineering, Czech Technical University in Prague
**Corresponding author: hanouvit@fjfi.cvut.cz*

Abstract

Intel Xeon Phi is a modern coprocessor designed for the high performance computing similar to GPUs by NVidia. TNL (Template Numerical Library) is a library providing abstract layer for accessing multicore CPUs and GPUs in numerical solvers. Our aim is to develop interface for Intel Xeon Phi coprocessor consistent with NVidia CUDA in TNL. In this paper we describe efficient method for bitwise copying of C++ objects similar to NVidia CUDA using Intel offloading extension of C++. As a working example we use the heat equation problem to demonstrate efficiency of the implementation on Intel Xeon Phi Knights Corner and to compare CPU with this coprocessor.

Key words: Intel Xeon Phi, HPC, MIC, Offload

1. INTRODUCTION

The Intel Xeon Phi Knights Corner (KNC) is a modern coprocessor for high performance computing servers. It is an extension card for the PCIe 16x slot and it was published in 2012 (Chrysos, 2012). It contains up to 61 cores with a 4 way multithreading and it is equipped with up to 8 GB of multichannel RAM. A BusyBox based GNU/Linux operating system runs on this device and it is connected to the host operating system using a virtual ethernet network. Intel Xeon Phi is first device with the Intel Many Integrated Core (MIC) architecture, so abbreviation MIC denotes Intel Xeon Phi KNC in this paper, which was used for development.

There are two basic ways, how to run a code on this device. The first one is compiling a native application for the operating system of coprocessor. In this way user needs to connect to the device via ssh and run the application on it. The whole code runs on the coprocessor. The other method is *Offload*. In this way, user compiles application which runs on host processor and only special parts run on the coprocessor. There are several standards to create an application with an offloaded code in C/C++ language. The first standard is the Intel offload extension. It uses `#pragma` statements in the manner of OpenMP. The second one is a part of the OpenMP standard 4.0 (OpenMP, 2013) and the syntax is similar. The OpenMP standard was published later than Intel Xeon Phi. This APIs allow programmer to control entirely the data transfers between coprocessor RAM and host RAM but they do not support transfer of C++ objects. Finally, the offload can be created using Intel Cilk C++ extension. It fully supports C++ objects, but the programmer has not full control on the data transfers. The Intel Cilk C++ extension is supported by Intel Parallel Studio (Intel Corportaion).

TNL[1] (Oberhuber T., in preparation) is numerical library for solving partial differential equations on multicore CPUs and GPUs. It is actively devel-

[1] http://www.tnl-project.org

 ISSN 1641-8581

oped at the Department of Mathematics, Faculty of Nuclear Sciences and Physical Engineering, Czech Technical University in Prague. The goal of our work is to provide a similar interface of Intel Xeon Phi Coprocessor and NVidia CUDA for this library. In this paper, we study how to perform transfer of simple C++ objects between the host and the Intel coprocessor using Intel offload extension of C++. Our methods support bitwise copying of objects, similar to NVidia CUDA. We do not use Intel Cilk C++ extension, which fully support C++ objects, because the TNL library needs only transferring of bitwise objects for design and our aim is to fully control data transfers. This library does not use inheritance with virtual functions or abstract classes because it reduces options of optimization and it is not supported by GPU.

```
1  int a,b;
2  int arr [10];
3  int*darr=malloc(5*sizeof(int));
4
5  #pragma offload target(mic)
                  in(a,arr) out(b)
           in(darr:length(5))
6  {
7     ...
8  }
```

Listing 1: The offload example.

The paper is structured in the following way. In the section 2 the transfer of objects using Intel offload syntax is described. In the section 0 the heat equation problem and basic structure of Euler solver in the TNL library are described. Finally, the efficiency of an application written using the TNL library on host CPU and on Intel Xeon Phi KNC is compared in the last section.

2. OFFLOADING DATA TRANSFERS

Firstly, let us focus on the Intel offload extension syntax. An example is shown in the Listing 1. Variables which should be transferred are stated in `in`, `out`, `inout` clauses. These variables need to be *bitwise copyable*, i.e. basic types (int, double, etc.), structures of basic types and arrays of basic types. Furthermore, it can be pointer to dynamically allocated array, nevertheless in this case the name of variable needs to be followed by a `length` clause. It cannot be a C++ object. The goal of this section is to describe how to bypass this limitation.

The data transfer and the offload run are driven by the COI daemon (Chris J. Newburn, 2013), which is a part of the Intel manycore platform software stack (MPSS) - drivers of the Intel Xeon Phi. The communication between application and the daemon is added by the compiler, which supports offload extension. The *OFFLOAD_REPORT* environmental variable (Intel Corporation, 2015) can be set to a level 2 or 3. Then the application will write information about every offload to the standard output.

Next, two methods how to pass a simple C++ object to the offloaded region will be presented. After that, we focus on memory allocation on the Intel Xeon Phi coprocessor.

```
1  class myclass{
2
3  private:
4     int number;
5     int *pointer;
6  public:
7     myclass()
8     {
9           number=5;
10    }
11
12    __attribute__((target(mic)))
            int get()
13    {
14          return number;
15    }
16 };
```

Listing 2: An example of the class, which is not bitwise copyable.

2.1. Object copying

Assume that we want to copy an instance of the simple class from Listing 2 to the coprocessor and to call a method *get* on it. The method *get,* at line 12, has an attribute which allows us to call it in offload region. However, this class is not bitwise copyable, because it contains pointer (line 5) and it has constructor (line 7), therefor compiler does not allow to copy it to coprocessor. Our methods of copying assume that the class has the same size on the coprocessor and on the host device. At the end we obtain bitwise copy, thus any sub-allocated object will not be copied, constructor or destructor will not be called on the device and finally the table of virtual methods will not be updated.

```
1  (...)
2  myclass a;
3
4  uint8_t *phost_a=(uint8_t*)&a;
5
6  #pragma offload target(mic)
inout(phost_a:length(sizeof(a)))
7  {
8     myclass
*mic_a=(myclass*)phost_a;
9
10    cout << mic_a->get() <<endl;
11 }
```

Listing 3: Direct method of passing object to offload region.

The first method can be seen at Listing 3, we call it *direct method*. It creates a pointer (line 4) and passes the object into an offload as a dynamically allocated array of a basic type. We selected one-byte-length type *uint8_t*. This selection allows us to use the size of the class as a length of the array (line 6). Inside the offload, we need to retype the pointer back to our class in order to be able to use it (line 8).

```
1  myclass a;
2
3  uint8_t
host_hide_a[sizeof(myclass)];
4  memcpy((void*)&host_hide_a,
      (void*)&a,sizeof(myclass));
5
6  #pragma offload target(mic)
                  inout(host_hide_a)
7  {
8     myclass
*mic_a=(myclass*)&host_hide_a;
9
10    cout << mic_a->get() <<endl;
11 }
```

Listing 4: Indirect method of passing object to offload region.

The second method can be seen at Listing 4 and we call it *indirect method*, because it creates a bitwise copy of an object inside a locally created array of *uint8_t* (line 3-4) and then it passes this array into the offload (line 6). Inside the offload, it creates pointer to our class and fill it with the address of the array on the coprocessor (line 8). Subsequently, it can call the method *get*. This method has limitation for large objects. Because we create copy to the local variable, which is allocated on the stack, the size of the object is limited by the size of the stack on the host and the size of the stack on the device.

The key difference between these two methods is that the first method copies data from the host to the coprocessor as dynamically allocated array, so underlying system probably expects that the data are placed on the heap. The second method copies data from the host to the coprocessor as a local variable, so underlying system probably expects that they are placed on the stack. The indirect method is faster as can be seen in Section 4. To conclude, our observations imply that copying data from the stack is probably more efficient compared to copying data from the heap.

2.2. Memory allocation

We would like to manage memory allocations on the Intel Xeon Phi coprocessor in similar way as in NVidia CUDA. It means to allocate memory only on the device and pass its address in a pointer from the device to the host code. This would allow us to use the same data on coprocessor from different offloads. The Intel Offload extension of C++ contains system which allows persistent allocation of memory on the coprocessor by mapping host variables (or dynamic allocated arrays) to coprocessor variables (Davis, 2013). However, the mapping system maps only host variables to coprocessor variables and it is not compatible with NVidia CUDA, which use pointers with coprocessor addresses.

Consider construction from Listing 5. Let us create a structure which contains a pointer (lines 1-4) and pass it into the offload (line 12). Then use *malloc* function in the offloaded code (line 14). It is necessary to use this construction since pointers inside structures are not updated by mapping system. This method allows user to have full control on persistence heap memory allocation on the device. Nonetheless, all of these constructions bypass some offload rules thus source code still needs to be compiled with the compiler's switch `wd2568` (Davis, 2013), which forces compiler to ignore errors about not bitwise copyable structure with pointer.

```
1  template< typename Type >
2   struct hideptr{
3     Type *pointer;
4   };
5
6   (...)
7
8   double * mic_memory;
9   int size=15;
10  hideptr<double> hide;
11
12  #pragma offload target(mic)
out(hide) in(size)
13  {
14 hide.pointer=(double*)malloc
            (size*sizeof(double));
15  }
16
17  mic_memory=hide.pointer;
```

Listing 5: Memory allocation on coprocessor bypassing translation system of pointers.

3. TESTING PROBLEM

In this section, we will describe a problem and its numerical solution that we have used as a testing application in section 4. The heat equation problem in two dimensions is well known problem, but we recall it here. Let $\Omega = (0, X_{max}) \times (0, X_{max})$ be two-dimensional square region and J be a time interval from an initial time 0 to a final time t_f. The heat equation problem consists of finding a function u satisfying following equations:

$$\frac{\partial u}{\partial t} = \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) \quad \text{in} \quad \Omega \times J, \tag{1}$$

$$u = 0 \quad \text{in} \quad \partial\Omega \times J, \tag{2}$$

$$u|_{t=0} = u_{ini} \quad \text{in} \quad \Omega, \tag{3}$$

Where (2) is the Dirichlet boundary condition and (3) is an initial condition. Let h be a space step and $v_{i,j}$ be a point with coordinates $[ih, jh]$. Then we can set up a square mesh of $N+1 \times N+1$ points as
$\bar{\omega} = \{v_{i,j} | i = 0, \dots, N \quad j = 0, \dots, N\}$
and set of inner points as
$\omega = \{v_{i,j} | i = 1, \dots, N-1 \quad j = 1, \dots, N-1\}$.

Let τ be a time step and k be a time level. Time of the k-th time level is $t_k = k\tau$. Let I be the set of all time levels
$I = \left\{k\tau \middle| k = 0 \dots \left\lfloor \frac{t_f}{\tau} \right\rfloor\right\}$.

Now let $\tilde{u}: \bar{\omega} \times I \to R$ be a function which approximates the function u as
$u_{i,j}^k = \tilde{u}(v_{i,j}, t_k)$.

Under the previous conditions the finite difference explicit numerical scheme solver of the heat equation problem reads as:

$$u_{i,j}^{k+1} = u_{i,j}^k + \frac{\tau}{h^2}\left(u_{i+1,j}^k + u_{i-1,j}^k + u_{i,j+1}^k + u_{i,j-1}^k - 4u_{i,j}^k\right) \quad \text{on} \quad \omega \times I, \tag{4}$$

$$u_{i,j}^{k+1} = 0 \quad \text{on} \quad (\bar{\omega} \setminus \omega) \times I. \tag{5}$$

Now, we describe implementation of the Euler solver. A basic pseudo code of the Euler solver is summarized in Listing 6. The pseudo code is running on the host processor except for following parts running on the coprocessor: *Allocate space on device, Update every inner mesh nodes, Update every boundary mesh node.* This distribution of work between the processor and the coprocessor comes from the PDE solver from the TNL Library. The function *Create Snapshot of solution* copy data from coprocessor to the file on file system. The *Update* functions take several objects as arguments, e.g. object describing ω. They have to be transferred in every time step which is imposed by the design of TNL. Since it cannot currently recognize whether an object has been changed during the last time step or not. Hence, every delay in transfers of these small objects has significant impact on performance of the solver for small and medium size problems as can be seen in the next section.

- Initialize numerical mesh ω
- Allocate space on device for u
- Setup the initial condition u_{ini}
- Set $t = 0$
- **While** $t < t_f$ **do**
  - Update every inner mesh node by (4)
  - Update every boundary node by (5)
  - $t = t + \tau$
- **End While**
- Create Snapshot of solution u

Listing 1: The pseudo code of the Euler solver in the TNL library

4. EFFICIENCY EVALUATION

In this section, we present results of performed tests. All tests were running on the following hardware. Intel Xeon E5-2630v3 @ 2.4GHz was used as the host processor and Intel Xeon Phi P5110 was used as the coprocessor.

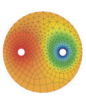
Table 1. *Comparison of two described methods for copying objects. Time of transfers was measured for different size of objects. Time is in the first and the second column. The third and the fourth column contain computed transfer speed in kB/s.*

		Time [ms]		Transfer speed [kB/s]	
		Direct	Indirect	Direct	Indirect
Object size [B]	8	0.607	0.069	0.01	0.11
	16	0.616	0.067	0.03	0.23
	32	0.612	0.067	0.05	0.47
	64	0.660	0.068	0.09	0.92
	128	0.620	0.066	0.20	1.89
	256	0.620	0.069	0.40	3.65
	512	0.608	0.068	0.82	7.41
	1k	0.630	0.068	1.59	14.64
	2k	0.620	0.070	3.23	28.61
	4k	0.618	0.069	6.47	58.31
	8k	0.608	0.071	13.16	113.15
	16k	0.637	0.072	25.13	222.22
	32k	0.621	0.078	51.57	412.90
	64k	0.627	0.086	102.07	747.66
	128k	0.647	0.106	197.96	1 205.27
	256k	0.705	0.125	363.22	2 046.36
	512k	0.759	0.180	674.22	2 844.44
	1M	0.905	0.404	1 131.87	2 537.79
	2M	1.202	0.741	1 703.26	2 763.83
	4M	1.620	1.658	2 528.40	2 470.30
	8M	3.278	2.875	2 499.16	2 849.69
	16M	5.455	E	3 003.76	E
	32M	10.899	E	3 006.49	E
	64M	20.939	E	3 129.85	E

Firstly, we compared two presented methods of copying objects. They were tested by a simple application which copy an object to the coprocessor 10 000 times. We tested both described methods for various size of the object. As can be seen from the Table 1 the indirect method for small objects is about 9 times faster than the direct one. Recall now that the indirect method cannot be used for large objects due to the limit of the stack size.

The next test adds basic support of Intel Xeon Phi to the TNL library using the indirect method of coping objects and consequently runs the Euler solver of the heat equation problem. The test was running with following parameters: $\tau = 0.00005$, $t_f = 0.04$, $h = 0.015625$ and the initial condition $u_{ini} = \sin x \cdot \sin y$. It was running with following sizes of mesh: 64×64, 128×128, 256×256, 512×512, 1024×1024, 2048×2048, 4096×4096 and 8192×8192. The space step h was kept constant for all meshes, thus physical size X_{max} of area was growing with mesh size, but the solver had the same number of iterations for all sizes of meshes.

Table 2 presents the comparison of the host processor and the coprocessor using the test application based on TNL library. The first four columns contain times of computations on 1, 2, 4, and 8 cores of the host processor. Due to the *TurboBoost* technology it was running on various frequencies: 3.2GHz for single core, 3.0GHz for two cores, 2.6GHz for four cores and 2.4GHz for 8 cores. These frequencies were observed to be constant during the computations. We decided not to switch off *Turboboost* of processor to allow processor to produce its best results for comparison with the coprocessor best results. The fifth column (MIC) contains time of computations on Intel Xeon Phi P5110 with full parallelization, i.e. it was running on 60 cores with 240 threads. The last two columns stand for computed speed-up on Intel Xeon Phi. The column with name *MIC speed-up 1 core* compares Intel Xeon Phi with single core of CPU, the column with name *MIC*



speed-up best compares time on Intel Xeon Phi with best measured time on the host processor for the selected mesh. We can conclude that the TNL library is faster on the Intel Xeon processor than on the Intel Xeon Phi coprocessor.

application and Table 5 contains computed efficiency. As can be seen the maximum speed-up was about 64 for mesh 8192x8192 and 240 threads. The application started scaling for meshes 512x512 and

Table 2. *Comparison of Intel Xeon Phi with multicore CPU. Columns 1-8 core contain time of computation on CPU, column MIC contains computation time on coprocessor. Columns speed-up contains speed-up of computation on Xeon Phi versus single core of CPU and Xeon Phi versus best CPU time.*

		Time [s]					MIC speed-up	
		1 core	2 core	4 core	8 core	MIC	1 core	Best
Mesh size	64x64	0.03	0.02	0.02	0.02	0.89	0.03	0.02
	128x128	0.11	0.06	0.04	0.23	0.93	0.12	0.04
	256x256	0.42	0.22	0.13	0.18	1.06	0.39	0.12
	512x512	1.64	0.83	0.43	0.29	1.21	1.36	0.24
	1024x1024	6.4	3.23	1.67	0.9	2.33	2.75	0.39
	2048x2048	28.54	14.5	7.79	4.81	6.66	4.28	0.72
	4096x4096	113.99	57.87	31.1	19.07	22.55	5.05	0.85
	8192x8192	473.95	240.2	128.4	78.02	93.51	5.07	0.83

Table 3. *Time of computation for various numbers of cores of Intel Xeon Phi. Time is in seconds.*

		Number of cores								
		1	2	4	8	15	30	60	120	240
Mesh size	64x64	0.86	0.76	0.69	0.67	0.69	0.73	0.81	0.89	6.20
	128x128	1.85	1.27	0.94	0.81	0.78	0.78	0.83	0.93	6.41
	256x256	5.91	3.31	1.95	1.37	1.02	0.92	0.93	1.06	6.11
	512x512	20.96	10.97	5.99	3.63	2.18	1.48	1.26	1.21	6.80
	1024x1024	82.43	40.79	20.93	11.00	6.92	3.71	2.66	2.33	9.53
	2048x2048	333.89	167.55	84.30	46.60	23.43	12.33	8.27	6.66	15.85
	4096x4096	1347.3	681.46	343.33	167.39	99.44	45.90	29.56	22.55	27.39
	8192x8192	5347.1	2696.2	1386.36	689.71	373.83	201.59	118.29	93.51	82.50

Table 4. *Speed-up for various numbers of cores of Intel Xeon Phi. It is compared with single core of Intel Xeon Phi.*

		Number of cores							
		2	4	8	15	30	60	120	240
Mesh size	64x64	1.13	1.25	1.29	1.25	1.17	1.07	0.97	0.14
	128x128	1.46	1.96	2.29	2.38	2.36	2.22	1.99	0.29
	256x256	1.78	3.04	4.33	5.80	6.42	6.36	5.56	0.97
	512x512	1.91	3.50	5.77	9.62	14.20	16.67	17.35	3.08
	1024x1024	2.02	3.94	7.50	11.91	22.22	31.05	35.41	8.65
	2048x2048	1.99	3.96	7.16	14.25	27.07	40.38	50.10	21.06
	4096x4096	1.98	3.92	8.05	13.55	29.36	45.58	59.74	49.20
	8192x8192	1.98	3.86	7.75	14.30	26.52	45.20	57.18	64.82

Table 3 compares time on computation of the TNL based testing application running on Intel Xeon Phi on various numbers of threads. We selected following numbers of threads: 1, 2, 4, 8, 15, 30, 60, 120 and 240. Table 4 contains computed speed-up of

larger. Good efficiency is reached when using less than 30 threads and mesh larger than 512x512.

Table 5. *Efficiency for various numbers of cores of Intel Xeon Phi. It is compared with single core of Intel Xeon Phi.*

		Number of cores							
		2	4	8	15	30	60	120	240
Mesh size	64x64	0.57	0.31	0.16	0.08	0.04	0.02	0.01	0.00
	128x128	0.73	0.49	0.29	0.16	0.08	0.04	0.02	0.00
	256x256	0.89	0.76	0.54	0.39	0.21	0.11	0.05	0.00
	512x512	0.96	0.88	0.72	0.64	0.47	0.28	0.14	0.01
	1024x1024	1.01	0.98	0.94	0.79	0.74	0.52	0.30	0.04
	2048x2048	1.00	0.99	0.90	0.95	0.90	0.67	0.42	0.09
	4096x4096	0.99	0.98	1.01	0.90	0.98	0.76	0.50	0.20
	8192x8192	0.99	0.96	0.97	0.95	0.88	0.75	0.48	0.27

5. CONCLUSION

In this paper we presented two methods of copying objects to Intel Xeon Phi in Offload mode with Intel Offload extension of C++. The second method was found faster by simple test. Subsequently, support of the Intel Xeon Phi was experimentally added to the TNL library using the second method of copying objects. Comparison of main processor and Intel Xeon Phi coprocessor was done using the heat equation problem solver which is a part of the TNL library. The TNL library with experimental support of Intel Xeon Phi KNC is faster on host processor than on coprocessor.

ACKNOWLEDGMENT

This work was partially supported by Project No. 16-16772S of the Grant Agency of the Czech Republic and Project No. 15-27178A of Ministry of Health of the Czech Republic.

REFERENCES

Davis, K. D., 2013, *Effective Use of the Intel Compiler's Offload Features*. Accessed: 30. 9 2016, Available form: Intel Developer Zone: https://software.intel.com/en-us/articles/effective-use-of-the-intel-compilers-offload-features

Chris J. Newburn, R. D., 2013, *Offload Compiler Runtime for the Intel® Xeon Phi™ Coprocessor*. Accessed: 11. 10 2016, Available form: Intel Developer Zone: https://software.intel.com/sites/default/files/article/366893/offload-runtime-for-the-intelr-xeon-phitm-coprocessor.pdf

Chrysos, G., 2012, *Intel® Xeon Phi™ X100 Family Coprocessor - the Architecture*. Accessed: 11. 10 2016, Available form: Intel Developer Zone: https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner11122

Intel Corporation., 2015, *Generating an Offload Report*. Accessed: 11. 10 2016, Available form: Intel Developer Zone: https://software.intel.com/en-us/node/522521

Oberhuber T., K. J. (in preparation). TNL: Framework for the finite difference method on modern parallel architectures.

OpenMP., 2013, *OpenMP Application Program Interface*. Accessed: 12. 12 2016, Available form: OpenMP: http://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf

EFEKTYWNY TRANSFER OBIEKTÓW JĘZYKA C++ NA KOPROCESORZE INTEL XEON PHI W TRYBIE OFFLOAD

Streszczenie

Intel Xeon Phi jest nowoczesnym koprocesorem przeznaczonym do obliczeń wysokiej wydajności w dużej mierze podobnym do układów GPU NVidia. TNL (Template Numerical Library) jest biblioteką dostarczającą abstrakcyjną warstwę umożliwiającą dostęp do wielordzeniowych procesorów CPU i GPU przez solwery numeryczne. Naszym celem jest stworzenie interfejsu dla koprocesora Intel Xeon Phi zgodnego z Nvidia CUDA w TNL. W pracy przedstawiono wydajną metodę kopiowania bitowego obiektów języka C++, podobną do tej zaimplementowanej w NVidia CUDA z wykorzystaniem rozszerzenia offload języka C++. Jako przykład wykorzystano rozwiązanie problemu przewodzenia cieplnego w celu demonstracji efektywności implementacji opartej o Intel Xeon Phi Knights Corner oraz porównanie obliczeń CPU z koprocesorem.

Received: December 31, 2016
Received in a revised form: May 5, 2017
Accepted: June 6, 2017