XeonPhi Meets Astrophysical Fluid Dynamics

Evghenii Gaburov\textsuperscript{a}, Yuri Cavecchi\textsuperscript{b}

\textsuperscript{a}SURFsara, Science Park 140, 1098XG Amsterdam, the Netherlands
\textsuperscript{b}Anton Pannekoek Institute, University of Amsterdam, the Netherlands

Abstract

This white paper reports on ours efforts to optimize a 2D/3D astrophysical (magento-)hydrodynamics Fortran code for XeonPhi. The code is parallelized with OpenMP and is suitable for execution on a shared memory system. Due to complexity of the code combined with immaturity of compiler we were unable to stay within the boundaries of Intel Compiler Suite. To deliver performance we took two different approaches. First, we optimized and partially rewrote most of the bandwidth-bound Fortran code to recover scalability on XeonPhi. Next, we ported several critical compute-bound hotspots to Intel SPMD Program Compiler (ISPC), which offers performance portability of a single source code across various architectures, such as Xeon, XeonPhi and possibly even GPU. This approach allowed us to achieve over 4x speed-up of the original code on dual-socket IvyBridge EP, and over 50x speed-up on the XeonPhi coprocessor. While the resulting optimized code can already be used in production to solve specific problems, we consider this project to be a proof-of-concept case reflecting the difficulty of achieving acceptable performance from XeonPhi on a “home-brewed” application.

Project ID: 2010PA1744/MagHydroBurn

1. Introduction

Accreting neutron stars have a thin layer of matter coming from a companion donor star. This layer, under the proper conditions, ignites and the resulting flame burns unstably propagating sideways via deflagration. This is a challenging astrophysical fluid dynamics problem due to vast range of length-scales that have to be resolved: from a few meters in the vertical direction to \(\sim 10\) kilometers in the horizontal direction. Furthermore, the problem also covers a wide range of time-scales: from ultra-fast nuclear burning reactions to sound-wave propagation. Additionally, the neutron star is rotating around its axis which requires inclusion of rotational effects, in particular the confining mechanism of Coriolis force is of crucial importance. We have already simulated, for the first time, deflagrating flames, while previous research was only focused on detonations, which is not what is believed to occur on the surface of neutron stars in nature.

Another important aim of this research is to understand the influence of the magnetic fields. For example, the magnetic field may confine the fluid by helping, or even substituting, Coriolis force, but it may also have negative effects on flame propagation. The hydrodynamic simulations, which initially were only carried out in 2D, must be extended to 3D in order to understand whether instabilities develop which can disrupt the flame due to the strong shear at its front. We also intend to simulate the propagation on the full geometry of the star, which may lead to more accurate interpretations and/or predictions of observations. Finally, the code is developed for any astrophysical configuration in which the fluid under analysis has an extreme aspect ratio, such as planet atmospheres or shells in stars.

In order to proceed with the aforementioned research plans, we need to make sure the application runs efficiently on the state-of-the art processors, such as Xeon, XeonPhi and GPU. This PRACE preparatory project was designed to have a closer look at the application and see what can be done to make the application run efficiently on XeonPhi. The resulting code optimizations will also be beneficial on Xeon, as well as the knowledge and experience, while not the source code, will be transferable to GPU should we wish to have a GPU port. The paper is structured as follows: In Section 2., we report on our analysis of the code and the resulting optimization strategies. The results and conclusions are presented in Sections 3. and 4. respectively.

2. Code description

The original programme is written in Fortran 90 and is parallelized with OpenMP for execution on shared memory systems. A quick runtime analysis on Xeon SandyBridge (SNB) and XeonPhi (KNC) processors
revealed a rather poor application performance\(^1\): the original application achieved \(\lesssim 5\%\) and \(\lesssim 0.5\%\) of the peak double precision performance on SNB and KNC respectively. A quick code analysis revealed that the application consists of hotspots that are somewhat evenly distributed between bandwidth- and compute-bound workload. To sum up, the initial analysis suggested that there is a potential for enormous speedup on KNC, and if achieved, the resulting optimizations will likely be of benefit for SNB as well.

### 2.1. Profiling with VTune

The first step was to profile the application, and for this task we used Intel VTune Amplifier (VTune). Fig. 1 and Fig. 2 display VTune screenshots from SNB and KNC runs respectively\(^2\). The application was run on a full chip in both cases, namely single socket SNB CPU, with 8 cores/16 threads (due to Intel HyperThreading Technology (HTT)), and all 60 cores/240 threads for KNC chip. We profiled the code on a large problem size \((1536 \times 3840)\) that has sufficient amount of work for both XeonPhi and Xeon.

Few important observations can be made from the hotspot data: in both cases most of the runtime is spent in \_kmp\_wait\_sleep and \_kmp\_x86\_pause calls, which indicates severe load imbalance in application. This is also visually confirmed by the concurrency distribution shown below the hotspot information. In particular, on SNB about half of the time the number of active processing cores were less than half of the grand total available. With KNC, the concurrency distribution is fatal: \(\lesssim 10\%\) of processing cores are active. This is also confirmed by the KNC hotspot information, where \(\gtrsim 80\%\) of the runtime is spent in \_kmp\_wait\_sleep. The data suggests that just by fixing the application concurrency we may be able to achieve up to \(\sim 2x\) and \(\sim 5x\) speed-up for SNB and KNC respectively.

In addition of identifying the application’s load imbalance, the hotspot information revealed four hotspot functions that warrants closer inspection, we call these the big four: helmeos, prognosys\_velocities, sig99\_sca and prognosys\_xfraction; same list stands for KNC. We also noticed that substantial amount of time is spent in memory copies shown as \_intel\_ssse3\_rep\_memcpy on SNB and \_intel\_lrb\_memcpy, yet the application did not appear to have many explicit copies, thus warranting further investigation.

\(^1\) We used likwid-perctr stethoscope mode from https://code.google.com/p/likwid/ to measure application performance.

\(^2\) Unless mentioned otherwise, we used Intel Compiler Suite 14.0.1 with "-O3 -xavx -ipo -parallel -openmp" and "-O3 -mmic -ipo -parallel -openmp" for SNB and KNC respectively.
2.2. Source code analysis and optimizations of bandwidth-bound workload

After analysing the big four we concluded that two of them, helmeos and sig99_sca, are compute-bound, while the remaining two, as well as most of the code, are bandwidth-bound. While inspecting the source we also identified the cause of the excessive amount of memory copies. Specifically, the user application has dozens of calls to various functions that perform some category of stencil operations. While details of computations differ, all these functions bear an affinity to one another:

```
1 ! function definition
2 function foo(fin, ...)
3     use params0 ! use a pre-defined parameter module
4     implicit none
5     real, dimension(mx,my,mz) :: foo
6     real, dimension(mx,my,mz),intent(in) :: fin
7     integer :: i,j,k
8     ..
9     !$omp parallel do private(i,j,k)
10     do k = 1,mz
11         do j = 1,my
12             do i = 1,mx
13                 foo(i,j,k) = fin(i+1,j,k) + fin(i-1,j,k) + fin(i,j-1,k) + ...
14             end do
15         end do
16     end do
17     !$omp end parallel do
18     end function foo
19 ! function use
20 fres = foo(pos, ...);
```

List 1. Sample stencil function form in the original code

We would like to draw attention to the line 20 in List. 1. It is important to understand what the compiler does here in order to understand the source of the implicit copy operations. Every time foo is called, the compiler allocates a temporary 3D array on the stack frame (line 5), writes result into it (line 13), and then copies it to fres (line 20). This results in unnecessary read/write operation, just to copy results from temporal storage to fres. If data, as in the application at hand, doesn’t fit the Last Level of cache (LLS), this can be heavily penalized. Indeed, consider the following simple change to the function:

```
1 ! function definition
2 subroutine foo(fin, ..., foo_out)
3     use params0 ! use a pre-defined parameter module
```

```
Here, we eliminate the unnecessary copy by passing \texttt{fres} directly to a subroutine. We also added dynamic scheduling to the OpenMP loop. This may have little effect on SNB, but it is of significant help on KNC due to larger core count sharing memory bus. This may easily lead to load imbalance among the threads, and the dynamic scheduling is (partially) able to rectify this source of imbalance.

In addition to these, we also replaced other unnecessary copies, such as

With this increases the number of code lines\footnote{This can be abstracted in C++, which will result in a compact code in comparison to Fortran or C equivalent.}, this optimization substantially decreases the number of implicit copy operations\footnote{We would like to stress that the application was compiled with \texttt{-parallel} flag, which means all implicit copy operations were parallelized.}. This decrease in the number of copies brings substantial performance benefits because the temporal results, which are originally stored in \texttt{scr1} and \texttt{scr2} arrays and are not eliminated by compiler (line 4 and line 6 in List. 3), are now explicitly stored either in registers or L1$. This may easily lead to load imbalance among the threads, and the dynamic scheduling is (partially) able to rectify this source of imbalance.

After we “patched” in this way nearly all parts of the code (including converting all relevant functions into subroutines as shown in List. 1 and 2) that we think mattered (circa 1k lines) we observed decrease in runtime by about 2x on SNB and substantial improvement in concurrency. While we don’t have numbers at hand, we are confident that these optimizations are of even greater benefit on KNC. While in particular, after this optimization step the wall-clock time spent in most of the bandwidth-bound workload paled in comparison to the two compute-bound functions: \texttt{helmeos} and \texttt{sig99_sca}, because the aforementioned optimizations have virtually zero influence on these two functions.
2.3. Tuning compute bound functions for XeonPhi

The XeonPhi programming manual makes it very clear that in order to harvest the computing power of the coprocessor requires the user application to be (auto-)vectorizable: nearly 90% of peak double precision performance comes from wide-vector unit (up-to 8 double precision floating points per math operation). Thus our first task was to inspect what prevents Intel Fortran Compiler to auto-vectorize the two compute-bound functions: helmeos and sig99_sca. The helmeos function is called from rhoget function which is a custom implementation of Newton-Raphson iterative root-finding algorithm, in which helmeos is called from within branches of if-statements. Without going too much into details, it became very clear from inspection of the function sources and compiler reports that the code is too complex for Intel Fortran Compiler to auto-vectorize, and the only remaining route is manual vectorization.

The first attempt was made with the help of !dir$ simd but the code logic, which includes nested branching and calls to helmeos inside the branches, is again too complicated for the compiler to vectorize this even with !dir$ simd. However, as a proof-of-concept, we were able to rewrite this functions in AVX intrinsics which demonstrated ≳2x speed-up for helmeos and sig99_sca respectively. This lead us to conclude that these two functions can be and must be manually vectorized for efficient execution on XeonPhi. While intrinsics is a fine approach when the source code will not require subsequent modification or is tailored for a particular hardware, the portability and expandability requirement did not permit us to ship intrinsics version of the code to the user; in other words, there would simply be no way for the user to easily modify the code without spending valuable time to learn intrinsics.

For this very reason we decided to use Intel SPMD Program Compiler, ISPC in short, in which to rewrite rhoget, helmeos and sig99_sca. For performance comparison we timed the runtime of both intrinsics and ISPC versions of the functions. This benchmarking showed that ISPC version was at most ∼10% slower than the intrinsics version—the price we are willing to pay in exchange for both portability and expandability.

The big anticipated bonus from this enterprise was that the same ISPC source code for ports of these three functions was also able to deliver speed-up of ∼3-4x on KNC. In the process of identifying the cause of less than expected speed-up, we stumbled upon a potentially serious performance bug in Intel C Compiler. The bug has been reported to Intel, but as of version 14.0.1 it is not fixed; although we do expect measurable performance improvement once the bug is fixed, we did not investigate it close enough to estimate by how much performance will be improved.

We refer the interested reader to inspect sig99 function in http://cococubed.asu.edu/code_pages/kap.shtml from which sig99_sca is derived.

See http://ispc.github.com. However, at the time of our porting activities, the KNC support in ISPC was incomplete for our purpose. This motivated us to fix this, and as of version 1.5.0 the support of KNC inside the ISPC is complete.

We used -xavx and --target=avx1-i64x4 compiler flags for intrinsics and ISPC functions.

The compilation process for KNC with ISPC is the following: source-to-source with ISPC (.ispc→.cpp) and source-to-object with ICC (.cpp→.o).

Fig. 3. VTune screenshot displaying profiling (top) and concurrency (bottom) information for the optimized version of the code running on a single SNB socket. For information about panels please refer to Fig. 1.
2.4. Profiling optimized code

We conclude this section by profiling the optimized code. In Fig. 3 and Fig. 4 we show hotspot lists and concurrency distributions for SNB and KNC respectively. From the bottom panel of Fig. 3 it becomes clear that the code has excellent concurrency on Xeon CPU, and from the top panel we can see that the load imbalance is nearly gone with about 20% of runtime being spent in helmeos. There are still memory copies present, but their source is known to us and we are convinced that further optimization will produce diminishing returns compared to increase in the code complexity.

We find it interesting to compare the number of instruction retired for compute-bound function. Comparison of top-panels in Fig. 1 and Fig. 3 shows that on SNB the optimized function executed 2.2 and 3.3 fewer instructions for helmeos and sig99_sca respectively. In case of KNC, however, the number of instructions is reduced by 6.8 and 7.8 for helmeos and sig99_sca. Given that both functions are dominated by double precision mathematical computations, and that KNC can execute 8 double-precision floating points per instruction, the near 8x reduction is in line with expectations. The speed-up, however, is more modest: 3.0x and 3.8x respectively. The CPI column shows that this is due to doubling amount of cycles spent per instruction on average. The reason for this is unknown to us at the moment, and must be investigated further.

Finally, we explore the concurrency of the code on XeonPhi from the bottom panel of the Fig. 4. When compared to the original concurrency distribution in Fig. 2, the improvement is phenomenal. Still, about 40% of runtime is spent in less than ∼20 threads (∼5 cores). It requires further investigation to understand what causes this. The solution to the remaining concurrency problem has a potential to deliver up to 1.6x speed-up, which certainly warrants the effort. Combined with the possibility to recover original CPI for helmeos and sig99_sca we are looking at an additional ∼2x speed-up for the whole application on KNC.

3. Results

We present the final results on different processors and different problem sizes in Fig. 5. Here, the horizontal axis represents hardware on which the application was run, the vertical axis is the runtime in seconds, and we use color-coding for different problems size with the corresponding legend on the top left corner of the figure. For the reader’s convenience we also print the run-time on each of the bars. In contrast to the previous sections, we collected the runtime information for this plot on compute node equipped with a dual-socket Ivy Bridge E5-2695 v2 with HTT enabled (IVB), which has a total of 24 cores/48 threads available for computation. This gives a fair comparison between different state-of-the-art processors from Intel.

Comparison of peak performances and peak bandwidth shows XeonPhi 5100 is about 2x faster than such a dual-socket IVB node. This number should anchor our expectations for the runtime numbers, namely we should not see more than 2x speed-up on XeonPhi. If this was the case, the only explanation would be that the Xeon code is not optimized, which is a highly unlikely proposition because most of optimizations for XeonPhi are
automatically of benefit to Xeon as well. The inverse however is possible, if less than 2x speed-up is observed on XeonPhi, this means the code likely has further optimization potential.

We measured the application runtime on different problem sizes ranging from 192x480 to 1536x3840, which correspond to problem size ratio of 64. Few observations can be made from the figure. As expected, the application efficiency of small problem sizes (192x480, 384x960) on KNC is rather poor. In particular, we note that theoretically the runtime should differ by 4x for neighbouring problem sizes (the problem size increases by 4x in each step). In the case of IVB, both old and new versions of the application show about 4x increase in run-time with every problem-size step. This is not true for the new version running on KNC. Even for the largest problem sizes we see only 3.4x increase in runtime. We find it to be sufficiently close to 4x to argue that the overhead becomes somewhat irrelevant at such problem sizes.

We now focus on the largest problem size (1536x3840). There it can be seen that XeonPhi 5100 is $\sim$10% faster than dual-socket E5-2695 v2 CPU. As we earlier mentioned, theoretically XeonPhi 5100 is approximately twice as fast as our dual-socket IVB configuration. This implies that there is still a 2x speed-up potential for the application on KNC, which is consistent with our analysis at the end of Sect. 2.4. Unfortunately, due to limited amount of time allocated for this project and complexity of the application we were not able to act on our analysis, but it is our conviction that this will likely be able to recover the remaining $\sim$2x speed-up. We leave this for future work.

4. Conclusions

We conclude this white paper by claiming that XeonPhi is an excellent throughput optimized processor, and is capable of delivering outstanding performance. However, it takes serious amount of efforts, beyond what is being currently taught in textbooks and manuals, to tap into this performance. The biggest obstacle, in our opinion, is the immaturity of Intel Compiler Suites which prevents the user to exploit full richness of vector parallelism that XeonPhi offers. In this white paper we presented a possible solution to this but this forces the user to move beyond the boundaries of the Intel Compiler Suite. However, the benefit of all this is that the resulting code may offer performance portability across different processors, such as Xeon, XeonPhi and possibly GPU, on codes with a wide range of complexities that usually encountered in the field of High Performance Computing. It is unfortunate that the lack of mature industry accepted standard to explicitly use vector capabilities of modern processors in a portable manner creates major obstacles to developing applications that might be called legacy codes in a decade from now; but this will change.

Acknowledgements

EG would like to thank Alex Duran from Intel Corporation for his help with VTune, Intel Compiler Suite and low-level details of XeonPhi that made it possible to quickly identify, understand and fix various inefficiencies in the original code–efforts that ultimately led to the success of this project. This work was financially supported by the PRACE-IIP project funded in part by the EUs 7th Framework Programme (FP7/2007-2013) under grant agreement no. RI-261557. The work was achieved using the PRACE Research Infrastructure resources at SURFsara and Cineca.