Comparing Many–Core Accelerator Frameworks

G. Haase, A. Kucher, and M. Liebmann

Institute for Mathematics and Scientific Computing,
University of Graz, 8061 Graz, Austria

Feb. 15, 2012

Abstract

GPUs as general purpose processors already are well adopted in scientific and high performance computing. Their steadily increasing success caused others than GPU hardware vendors to work on many–core processors as hardware accelerators. With CUDA and OpenCL there are two frameworks available for GPU programming. Apart from potential compatibility problems with the upcoming hardware, both frameworks share a common disadvantage. It is hard to program them efficiently and it can be even harder to maintain them in existing large applications. PGI Accelerator and HMPP Workbench are two frameworks with an abstract programming model, similar to OpenMP, that allow the porting of existing sequential codes by means of preprocessor directives. Depending on the target architecture, i.e., the hardware accelerator, a code generator uses these directives to generate hardware accelerator code.

In this technical report we will present these frameworks and evaluate them in terms of performance and applicability. It will turn out, that PGI Accelerator and HMPP Workbench give similar performance results. The code generator of PGI Accelerator can perform a number of optimization strategies automatically, but HMPP Workbench is more sophisticated regarding the spectrum of target architectures and the applicability to already existing codes.

1 Introduction

In the last years, GPUs have significantly evolved from processing units dedicated to computer graphics to general purpose GPUs (gpGPUs), well applicable to many problems in scientific computing. By Nov. 2011, already three of the top five supercomputer systems worldwide are equipped with gpGPUs dedicated to high performance computing[^1].

There are two major and well established programming interfaces for gpGPUs, the proprietary Compute Unified Device Architecture (CUDA) and the Open

[^1]: This research was funded by the RP-7 (Cleansky) call: SP1-JTI-CS-2010-1-GRA-02-008 and by the Austrian Science Fund (FWF): F32-N18, by the RP-7 program: SP1-JTI-CS-2010-1

[^1]: http://www.top500.org/
Computing Language (OpenCL). Both programming interfaces allow an efficient programming of gpGPUs from a rather abstract perspective. Over the years a steadily growing community of experts developed a large number of highly optimized routines and libraries that allow treating standard problems by means of gpGPUs with very basic knowledge about the underlying hardware and programming concepts.

But significant drawbacks exist. In many cases these libraries are dedicated to a certain kind of hardware. This means that for each new hardware accelerator architecture new libraries have to be developed or modified. This consideration is of vital importance. In 2011 Intel announced Knights Corner, a co–processor for commercial use breaking 1 TFLOP double precision. It is based on Intels Many Integrated Core Architecture (MIC). This means another player appeared on hardware accelerator market and it is likely that others might follow as well, as for example the business volume of NVidia Tesla increased from $10 million in the fiscal year 2009 to $100 million in the fiscal year 2011 and future perspectives are good.

Furthermore, such libraries have to be integrated in existing code. This might be a minor problem for small codes or applications in academics and research, but a big issue for large industrial codes. Such codes often have very high standards for reliability and an accurate testing of new code is time consuming and therefore expensive. Apart from that, the success of a software vendor might be based on special knowledge and proprietary algorithms. Thus, specialized libraries might not be applicable to existing codes. Finally, code maintenance is of big importance. Consider several target architectures, i.e., hardware accelerators, a piece of software invokes to speed up computations. For each accelerator architecture, a special code has to be developed and integrated into the software. This makes software hard to maintain and in case of an algorithmic change the codes for all accelerators have to be modified.

Therefore a framework that allows an easy porting of already existing code to hardware accelerators with only basic knowledge of a hardware accelerator would be convenient. Roughly speaking, such a framework ideally should meet at least the following criteria: It ...

- is easy to use.
- can be applied to already existing codes with minor effort.
- is preferably independent of the hardware accelerator used.
- allows code maintenance in a reasonable time.

Frameworks that fulfill these requirements at least in parts do exist and we will examine two of them, PGI Accelerator by the Portland Group and HMPP Workbench by CAPS Enterprise. In 2009 M. Liebmann et al. already compared OpenGL with CUDA for a Jacobi iteration for performance and usability. In 2011 R. Membarth et al. studied several accelerator frameworks, including PGI Accelerator.
Accelerator and HMPP Workbench, for image registration considering different aspects including performance and usability in a very conclusive way [5].

The programming models of PGI Accelerator and HMPP Workbench evolved since then, which legitimates that we will revisit these aspects from a different perspective. We will consider the applicability of the frameworks for the acceleration of typical code examples and of already existing, potentially old large scale code. Each of these problems may be considered as an aspect during the porting of existing implementations of potentially complex numerical algorithms for hardware accelerators. We take into account a comparison for simple BLAS routines. This allows to measure the performance against highly optimized CPU and GPU implementations and gives an idea of the programming effort. Then we focus on a compute intense part of an existing large industrial code. As data transfers between system memory and accelerator memory are highly expensive, we will examine the flexibility of the frameworks regarding data management on the example of the requirements for multi-grid computations.

Our observations are based on Fortran 90 code. The remaining report is organized as follows. First we will introduce the hardware accelerator frameworks in §2. Then we state basic test cases for the frameworks in §3. The report finishes with the presentation of results and a final conclusion.

2 Many-Core Accelerator Frameworks

A hardware accelerator is a piece of hardware, that can perform specialized tasks more efficiently than a CPU. Within the last decade, hardware accelerators for general purpose applications experienced some kind of revival. The main reason for this development is the transition from graphics chips dedicated to computer graphics only to general purpose many-core processing units dedicated to scientific and high performance computing. Recent hardware accelerators are many-core processor cards with a specialized on-board memory. Their hardware configuration, i.e. the many-core processors, require special programming techniques, as they strongly differ from multi-core CPUs. NVidia hardware accelerators for example consist of several streaming multiprocessors (SMs), each containing multiple cores called streaming processors (SP). Every SM can independently create and execute threads, organized in blocks, performing the same instructions against different data synchronously (SIMD). Fig. 1 shows the basic architecture of such devices.

CUDA and OpenCL are two accelerator frameworks that allow accelerator programming. They both are based on the C programming language and include APIs and both give access to the accelerator.

OpenCL is an open standard and currently supported by gpGPUs by AMD and NVidia. CUDA is limited to NVidia devices only. In Dec. 2012 NVidia announced that CUDA will be opened. Both frameworks provide full access to the underlying hardware accelerator, but have significant drawbacks. Writing performant accelerator code within these frameworks requires a deep understanding of the hardware and is time consuming. Another issue is code maintainability for big applications, as for every hardware accelerator architecture, specialized code has to be integrated into the software framework.
PGI Accelerator and HMPP Workbench (Hybrid Multicore Parallel Programming) are two rather new accelerator frameworks, that allow accelerator programming with basic knowledge about the hardware. The idea of both frameworks is to add meta-information in form of directives to compute intense code parts. For the C programming language these directives are \#pragma.

Fig. 2 shows the basic concept of these accelerator frameworks. In a first step the existing sequential code is appended with data transfer directives and compute directives. Data directives indicate a data transfer from system memory to the hardware accelerator memory or vice-verse. Compute directives are then used to process the data on the hardware accelerator. These directives are usually attached to loops, i.e., loops are marked for parallel execution on the hardware accelerator. For the building process, a preprocessor is launched. It separates CPU code from code marked for acceleration. While CPU code gets compiled by a native host compiler, the accelerator code first gets translated into CUDA or OpenCL code by a code generator and compiled by the accelerator compiler. The resulting binaries then get linked by a native host linker.

Trial versions for PGI Accelerator and HMPP Workbench can be obtained on the vendor’s websites. The following discussion of the frameworks is by no means complete, but may give a rough idea about their functionalities.

2.1 PGI Accelerator

The PGI Accelerator by the Portland Group can be considered as an extension to the C and Fortran compilers of the Portland Group. Pragma directives, similar

\[ \text{http://www.pgroup.com} \] and \[ \text{http://www.caps-entreprise.com} \]
to openMP, allow an acceleration of loops by hardware accelerators. This simple programming model is especially useful for the acceleration of small portions of code. Currently, the PGI Accelerator model is limited to Nvidia hardware, i.e., it only supports Nvidia graphics cards for acceleration.

**Acceleration concept**  PGI Accelerator is based on two kinds of directives. Data region directives and accelerator region directives. Data region directives are used to establish data transfers between system memory and accelerator memory. Accelerator regions indicate that code marked by them is to be executed on a hardware accelerator. A simple runtime library allows to choose and to control the target accelerator on systems with multiple accelerators.

List. 1 presents an example for a matrix multiplication implemented with PGI Accelerator. Later, we will revisit this example. The data directives in line 16 and line 32 instruct the compiler to copy the matrices a and b to the hardware accelerator. At line 32 the resulting matrix c is copied back. The code between line 18 and line 30 is marked for an execution on the accelerator. The ordering of the i,l–loop ensures coalesced memory access for NVidia devices. For an i,l–loop ordering PGI Accelerator detects non coalesced memory accesses and permutes the loops.

Listing 1: PGI Matrix multiply

```fortran
program matrixMultiply
  ...
  matmult(m, n, q, a, b, c)
  ...
contains
  subroutine matmult(m, n, q, a, b, c)
    implicit none
    integer, intent (in) :: n, m, q
    real, intent (in) :: a(n, m), b(m, q)
    real, intent (out) :: c(n, q)
    integer :: i, j, l

    do i=1,n
      do j=1,m
        ...
      end do
    end do
end subroutine matmult
end program matrixMultiply
```

Figure 2: Basic concept of the accelerator frameworks.
One way to build this application is to simply invoke pgfortran by executing
pgfortran matrixMultiply.f90 -ta=nvidia -Minfo. The -ta flag indicates
that the target accelerator is an NVidia device and -Minfo returns a detailed
feedback on the compilation process for the hardware accelerator. This feedback
can be used to optimize the code, e.g. when loop dependencies prevent from par-
allelization. Without further specifications, PGI Accelerator generates binaries
for CUDA devices with different compute capabilities and at run time the best
version for the available hardware accelerator gets called by the run time. The
-Mcuda=keepgpu flag generates a CUDA code file for further inspections.

For a more detailed description of the functionality, we refer to [9].

Data management Data transfers between CPU memory and accelerator
memory are either automatically detected by the compiler, or may be performed
manually by !$acc data region–statements in the code. Data transfers and
actual computations can be performed in different routines. For Fortran 90 code,
routines have to provide an explicit interface. !$acc reflected–statements in
routines with an accelerator regions indicate that data is already available on the
accelerator. List. 8 on page 16 shows a basic example for the use of reflected
data. Line 47 and line 51 establish a data transfer and line 63 indicates, that
the required data already is available on the accelerator. As of now, there are certain
limitations to the framework. The resolution of CPU data and accelerator data
relies on symbols within a routine and relies on dummy array arguments in case of
reflected data.

Acceleration method As already indicated, !$acc region–statements are
used used to accelerate code fragments containing loops. Whole code blocks
are translated to CUDA code, potentially resulting in multiple CUDA kernels.
Statements like !$acc do parallel and !$acc do vector give control over the
parallelization of the loops. Comparing lines 19 and 21 in list. 1, these directives
instruct the compiler to map the i–loop to blocks of size 16 in the x–dimension
and the l–loop to blocks of size 16 in the y–dimension in terms of CUDA notation.
The j–loop is executed sequentially. For c(i,1) temporary register variables
are used automatically. PGI Accelerator can apply special implementations for
reduce operations automatically (compare list. 4). A !$acc cache–clause can be
applied for manual caching in shared memory, but it is likely that PGI Accelerator automatically finds the best use for shared memory.

Accelerator regions are executed synchronously by default, but may be marked asynchronous.

2.2 openACC

OpenACC is a potential future standard for programming many–core processors [3]. Hardware and software vendors like NVidia, Cray Inc., the Portland Group and CAPS Enterprise are involved. First implementations of OpenACC are expected for the first quarter of 2012. As the programming model is strongly based on the PGI Accelerator model by the Portland Group, we will not further discuss it.

2.3 HMPP Workbench

HMPP Workbench by CAPS Enterprise is a directive based framework for accelerating sequential CPU code by many-core processors. This does not only include GPUs, but includes all devices that can be programmed with CUDA and OpenCL. A future support for Intel MIC hardware accelerators by Pthreads currently is in development. Accelerator code is generated in a preprocessing step, while CPU code can be compiled by any compiler. In contrast to PGI Accelerator, this framework consists of a compiler and a rather complex runtime environment. This minimalizes the CPU execution time, but to invoke a hardware accelerator from a rather abstract perspective. Additionally, HMPP Workbench comes with a set of performance analysis tools.

**Acceleration concept** The programming model of HMPP is similar to PGI Accelerator in terms of directives and the acceleration of code fragments containing loops. Yet the approach is different. HMPP implements a codelet concept. Codelets are distinguished program parts, i.e., procedures can be marked for an automatic translation to hardware accelerator code. A codelet can be called by a callsite marker. This strongly differs from the PGI Accelerator approach, which is orientated on code blocks. HMPP distinguishes between !$hmpp– and !$hmppcg– directives. Simply speaking, !$hmpp– directives give control over the accelerator, data transfer and codelets. !$hmppcg– directives may be used to further instruct the parallelization of loops within a codelet, depending on the hardware accelerator. Codelets may be grouped to share common data. In this case, they have to be placed in the same source code file.

List.2 contains an implementation of a matrix multiplication with HMPP for CUDA and OpenCL. In line 19 and line 20, two codelets are defined for the subroutine matmult. One for CUDA and one for OpenCL. This means the code between line 21 and line 43 is marked for an execution on the accelerator. The !$hmpp callsite– directives in line 4 establishes a mapping between the subroutine call in the next line and the matmult.cuda codelet defined in line 19. If a CUDA enabled hardware accelerator is available, this codelet gets executed synchronously and because of the args[•].transfer=auto statement in line 19, all necessary
data transfers are performed automatically. If no accelerator is available or the
execution on the accelerator fails, a fallback routine is called, i.e., matmult gets
executed on the CPU. This makes error handling easy.

Line 7 to line 15 are dedicated to the OpenCL codelet (matmult_opencl). Now we want to perform all data transfer manually and m,n,q,a,b are copied
to the OpenCL accelerator in line 7. Line 9 indicates that the OpenCL codelet
is to be executed asynchronously.

The args[m,n,q,a,b].advancedLoad=true–directive in line 10 indicates that
all input data is already available on the accelerator. Because of the synchronize–
directive in line 14, it is save to copy the resulting matrix c to system memory
in line 15.

For the same target accelerator and the same subroutine, multiple codelets
may be defined as well.

Please note, that the OpenCL version is a poor implementation but shall only
demonstrate the callsite concept.

Listing 2: HMPP Matrix multiply

```
program matrixMultiply
  ...
  !$hmp matmult_cuda callsite
  matmult(m,n,q,a,b,c)
  ...
  !$hmp matmult_opencl advancedLoad, args[m,n,q,a,b]
  ...
  !$hmp matmult_opencl callsite, asynchronous &
  !$hmp matmult_opencl args[m,n,q,a,b].advancedLoad=true
  matmult(m,n,q,a,b,c)
  ...
  !$hmp matmult_opencl, synchronize
  !$hmp matmult_opencl, delegatedstore, args[c]
  ...
contains
  !$hmp matmult_cuda codelet, target=CUDA, args[*].transfer=auto
  !$hmp matmult_opencl codelet, target=OPENCL, args[*].transfer=manual
subroutine matmult(matmult(m,n,q,a,b,c)
  implicit none
  integer, intent (in) :: n,m,q
  real, intent (in) :: a(n,m), &
  b(m,q)
  real, intent (out) :: c(n,q)
  integer :: i,j,l
  real :: tmp
  ...
  !$hmpcg (OPENCL) permute l, i
  !$hmpcg grid blocksize 16x16
  do l=1,q
    do i=1,n
      tmp = 0.0
      do j=1,m
        tmp = tmp + a(i,j)*b(j,l)
      end do
      c(i,l) = tmp
    end do
  end do
  ...
end subroutine matmult
end program matrixMultiply
```

For a more detailed description of the functionality, we refer to [2].
Data management  Data management can be left to the preprocessor or it can performed manually. HMPP3 offers two different concepts for data management. Until HMPP2 it was only possible attach data transfers to a certain codelet/callsite. We have seen this in list. 3.

The other concept is data mirroring. When a codelet is called and data already has been transferred to the accelerator, the HMPP runtime finds the accelerator memory address to a given system memory address in the argument list of the codelet. The runtime associates these addresses. We will discuss this concept in list. 9 more detailed.

Acceleration method  Accelerator code generation is directed by $\texttt{hmppcg}$ parallelization directives. In addition to functionalities also provided by PGI Accelerator, the HMPP preprocessor allows loop transformations. This includes loop order permutations, distributions and fusing.

Now we will make a short excursion to the C programming language and CUDA enabled devices. In C, multidimensional arrays are stored in a row–major order. Therefore an access pattern for the two dimensional array $A$ in list. 3 (line 8 and line 9) is advisable. For a fast (coalesced) memory access for a CUDA version, the $i$–loop and the $j$–loop should be permuted by the $\texttt{permute}$–directive in line 7.

Listing 3: Loop permutations

```
1. . .
2 #define n 100;
3 #define m 200;
4 . . .
5 #pragma hmpp dummy_cuda, target=CUDA, args[*].transfer=auto
6 void dummy(float** A) {
7    #pragma hmppcg (CUDA) permute i, j
8    for (i = 0; i < n; i++)
9    for (j = 0; j < m; j++)
10       A[i][j] = i*j;
11 }
```

External and native functions  External C or Fortran functions may be used within a codelet. The source code for such functions has to be available, as HMPP Workbench performs inlining. In Fortran, external functions are indicated by the $\texttt{hmppcg extern}$–directive.

In the terminology of HMPP Workbench, native functions are CUDA or OpenCL native functions ($\texttt{device=}$–functions in case of CUDA). They are declared within a XML file, containing the source code language, the signature of the native function and the function definition in CUDA and/or OpenCL.

For a more detailed description of external and native functions within HMPP Workbench, we refer to the documentation of the framework.

2.4 Framework Comparison

In the following we present our impressions on the frameworks, based on the experienced gathered during their evaluation. For a comparison with already well established frameworks, we we take into account CUDA as well. Please
consider that such an evaluation can not be purely objective and is also strongly
affected by the test cases used.

Both frameworks follow an approach, which is similar to OpenMP. PGI Ac-
celerator allows acceleration of code blocks and provides basic functionalities
for data transfers. It focuses on the code generator (source-to-source compiler)
and on special features, such as automatically performing reduce operations or
exploiting shared memory on NVidia devices.

HMPP Workbench allows the acceleration of procedures, called codelets. It
comes with code generators for CUDA and OpenCL and a sophisticated run time
environment. Data is always bound to codelets or groups of codelets.

We take into account performance, community, support, documentation, porta-
bility, future availability, and flexibility. For an evaluation regarding implemen-
tational aspects, please see § 4.3.

**Performance** With sufficient knowledge about the target hardware and pro-
gramming effort, the best performance can be expected with CUDA. PGI Ac-
celerator and HMPP Workbench give good performance results (compare § 4).
However, the performance achievable depends on various factors. For example,
both frameworks can not take advantage of texture memory of NVidia devices.

**Community** CUDA is already well-established and popular in high perfor-
mancc computing. A large and competent user community exists and one may
find answers to any kind of questions in corresponding on line forums. For PGI
Accelerator, the community is limited to the user forum on the website of the
Portland Group. Either the PGI support team or regular forum users usually
give helpful answers to questions related to the framework. Yet, the PGI Acce-
lerator community still is very small. We are not aware of such a forum for HMPP
Workbench. The target group of HMPP Workbench seems to be companies.

**Support** As CUDA can be obtained free of charge, one cannot expect full sup-
port. A license of PGI Accelerator includes 1 year of support. CAPS Enterprise
provided us with full support for our trial license for HMPP Workbench. Both
support team prove to be fast and gave precise and helpful answers.

**Documentation** CUDA has the best available documentation and comes
with broad range of sample codes. The other frameworks are evolving rapidly.
Their documentation always is up-to-date, but in particular for PGI Accelerator
it sometimes lacks of precise information. We experienced the code examples for
HMPP Workbench to be very helpful and straight to the point.

**Portability** We have already refered to the fact that NVidia plans to open
the CUDA framework, which will allow other graphics card vendors to support
CUDA as well. PGI Accelerator currently is limited to CUDA, although all kinds
of hardware accelerators might be supported in future (please compare § 2.2).
HMPP Workbench currently can generate CUDA and OpenCL code. Support
for Pthreads is planned.

\[\text{http://www.pgroup.com/userforum}\]
Future proof  A reliable prognosis is not possible to make. But the future perspectives for all frameworks seem to be good. CUDA is already very well established in high performance computing and will be opened. The Portland Group and CAPS Enterprise launched open standard projects for their frameworks.

Flexibility  Using CUDA, one has the freedom to manually write code for the hardware accelerator used and has full control over it. PGI Accelerator allows basic control, but lacks a convenient data management. As of now, we have the impression that HMPP Workbench is more mature, as it implements a sophisticated data management concept.

<table>
<thead>
<tr>
<th></th>
<th>CUDA</th>
<th>PGI</th>
<th>HMPP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td>++</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Community</td>
<td>++</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>Support</td>
<td>0</td>
<td>+</td>
<td>++</td>
</tr>
<tr>
<td>Documentation</td>
<td>++</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Portability</td>
<td>-</td>
<td>0</td>
<td>++</td>
</tr>
<tr>
<td>Future proof</td>
<td>++</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Flexibility</td>
<td>++</td>
<td>-</td>
<td>+</td>
</tr>
</tbody>
</table>

Table 1: Some rough and category based estimates on the frameworks. They are based on our experience gathered during their evaluation.

3 Typical Code Examples

In the following we introduce three different kinds of examples which we treated within the accelerator frameworks presented. Each of these examples may be considered an aspect during the porting of existing implementations of numerical algorithms for hardware accelerators.

First we start with selected basic linear algebra routines such as matrix multiplications and scalar products. Such, or similar routines are contained in the example codes provided by the framework vendors. cuBLAS is a highly performant implementation of the BLAS\textsuperscript{8} for NVidia GPUs\textsuperscript{9}. A comparison between accelerator framework and this optimized library gives a conclusive benchmark.

Then we set the focus on a slightly more complex example, a discrete flux computation that arises from CFD. It can be considered as a coupled 7–point stencil computation. This of interest because we are not aware of optimized GPU libraries for such an application. Stencil computations on GPUs have been discussed in \textsuperscript{4}. In \textsuperscript{6} a parallelization approach for 3D finite difference computations using GPUs is described and a CUDA implementation for a 25–point stencil exploiting on–board shared memory is given.

\textsuperscript{7}http://www.openacc-standard.org and http://www.openhmpp.org
\textsuperscript{8}http://www.netlib.org/blas/
\textsuperscript{9}http://developer.nvidia.com/cublas
Finally we investigate the applicability of these frameworks for already exist-
ing and potentially old codes that do not meet modern programming paradigms. In a small case study we will examine this by means of the flexibility of data management. For this we consider the concrete example of multi grid computations in combination with the traditional dynamic data management strategy in Fortran 77 codes, i.e., a work array.

All illustrative code snippets presented in this section are implemented in the Fortran 90 language. The code examples (excluding the coupled 7-point stencil computation) are available on the Internet.  

3.1 Basic Linear Algebra Routines

Scalar product List. 4 is an implementation using PGI Accelerator. The !$acc reflected-statement in line 17 indicates that input vectors already are available on the accelerator memory. The actual data transfer is performed in line 3 and line 5. The loop in the !$acc region block is marked for parallel execution on the accelerator. Without further instructions, PGI Accelerator chooses to parallelize the loop in a 1-dimensional thread topology with 256 threads per block. PGI Accelerator automatically detects the + reduce operation in line 23.

Similarly, the implementation in list. 5 with HMPP Workbench relies on a gridification in line 23 by a !$hmppcg parallel-statement. The reduce operation is indicated by !$hmppcg reduce(+:res) in line 24, as the HMPP compiler does not automatically detect it. Data transfer is established before the codelet gets called in line 6.

Listing 4: PGI Scalar product

```
program scalarProduct
...
!$acc data region copyin(a,b)
inner(n,a,b,res)
!$acc end data region
...
contains
subroutine inner(n,a,b,res)
implicit none
integer, intent (in) :: n
real, intent (in) :: a(n),b(n)
real, intent (out) :: res
integer :: i
!$acc reflected(a,b)
res = 0.0
!$acc region
do i=1,n
  res = res + a(i)*b(i)
end do
!$acc end region
end subroutine
end program
```

10http://uni-graz.at/~haasegu/Lectures/GPU_CUDA/WS11/kucher_code.tar
Matrix multiplication

By default, both frameworks mark the $l$–$i$–loops for parallel execution and the $j$–loop gets executed sequentially on the accelerator (list 6 and list 7). Regarding the ordering of the $l$–$i$–$j$–loops one has to be aware of the following fact: As the $j$–loop gets executed sequentially on the accelerator, the loop ordering basically meets the requirements for a fast memory access. Comparing the results in sec. 4.1 this is not true the CPU version of the matrix multiplication.

PGI Accelerator detects the reduce operation in line 18 and replaces $c(i,l)$ by a register variable.

HMPP Workbench does not detect the repeated access to $c(i,l)$ in line 18, thus it is of advantage to introduce a temporary register variable $tmp$ for a better performance.

Listing 6: PGI Matrix multiply

```
subroutine matmult (m, n, q, a, b, c)
  implicit none
  integer, intent (in) :: n, m, q
  real, intent (in) :: a(n,m), &
                     b(m,q)
  real, intent (out) :: c(n,q)
  integer :: i, j, l
  !$acc reflected(a,b,c)
  !$acc region
  do l=1,q
     do i=1,n
        c(i,l) = 0.0
     end do
  end do
end subroutine matmult
```
3.2 A coupled 7-point Stencil

In 3D–fluid dynamics the flux of a quantity in a direction may be defined as the rate per time and area, a quantity crosses a surface normal to the direction. We can understand the flux computation for a 3–dimensional structured grid as coupled 7–point stencil computations. Fig. 3.2 shows the basic idea.

From an algorithmic point of view, computing such a stencil from a 3–dimensional grid can be realized by three nested loops for each dimension (x, y and z).

If we want to perform these computations on multiple grid levels (e.g. for several cycles of a multi–grid computation) data transfers between system memory and accelerator memory are expensive. We consider this in the next section.

3.3 Multi–Grid Data Management

These days, the Fortran 77 language certainly is not the best choice to implement numerical algorithms. Nevertheless, a significant number of implementations and software libraries still exist. They are well tested and optimized and a reimplementation in a modern programming language is time consuming. Let us assume we want to perform some multi–grid computations on grids of different sizes, all stored in different sub–arrays of a large work–array. Such a work–array is widely used for “dynamic” data management in Fortran 77, as this language does not
Figure 3: Discrete flux approximation in $\mathbb{R}^3$ with 6 neighbor points.

have a concept for dynamic data allocation. Figure 4 shows a work vector containing three grid levels, each of different size. We want to perform computations

![Figure 4: Classical Fortran 77 work array containing 3 grids.](image)

on each grid in a subroutine by means of PGI Accelerator and HMPP Workbench. As data transfers are expensive and in practice many multi-grid iterations might be necessary, we want to transfer the grids to the accelerator only once.

Note, that the examples shall give the basic idea, as a practical example is significantly more complex. Yet, the approach is similar.

**PGI Accelerator** List. 8 shows how one can perform such a “multi-grid” computation within PGI Accelerator. In the main program we allocate the work-array work. The actual computations are performed in subroutine f2. For a CPU implementation one could simply pass a sub-array, i.e., one of the grids, to a routine which performs computations on that grid. Within PGI Accelerator we set pointers on each grid (line 20–23). This requires to compile with a compiler supporting the Fortran 90 standard or later.

Now we want to transfer each grid on the hardware accelerator. This is not possible within the main program, as PGI Accelerator cannot deal with pointers, nor multiple sub-arrays properly. Therefore we have to introduce a helper subprogram f1, which takes the grid pointers as dummy array arguments. In line 47, the three grids are copied to the hardware accelerator. Line 48–50 are calls to subroutine f2 with one of the grids as parameter. The !$acc$ reflected--
directive in line 63 indicates that the grid already is available in the accelerator memory.

Remember, that the reason for this approach is to keep multiple grid levels in the hardware accelerator to avoid time consuming data transfers.

This approach has several disadvantages. On the one hand it is not possible to deal with computations where the number of grid levels is not known at compile time, as we have to specify each grid in the source code explicitly. Secondly, for real problems with a large number of data and not only 3 grids, the parameter lists of subroutines might get very long.

When we consider line 10 in list 8, then we see that routines with accelerator related directives have to be in the same module. This is essential for how PGI Accelerator performs accelerator data management. Abandoning the use mod statement would lead to an insipicious compilation with the recent release of PGI Accelerator. The code generated would result in a segmentation fault at run time.

Listing 8: Data mirroring with PGI for 3 grids

```fortran
! Module file
module mod
  contains
  include 'f1.f90'
  include 'f2.f90'
end module

! Main program
program grid
  use mod
  implicit none

  integer, parameter :: n = 100
  real, dimension(n), target :: work
  integer :: l1, l2, l3

  work = -1
  g1 => work(1:1)
  g2 => work(2:5)
  g3 => work(6:14)
  l1 = 1
  l2 = 2
  l3 = 3

  call f1(g1, g2, g3, l1, l2, l3)
  print *, g1
  print *, g2
  print *, g3
end program

! f1.f90
subroutine f1(g1, g2, g3, l1, l2, l3)
  implicit none

  integer, intent(in) :: l1, l2, l3
  real, dimension(l1, l1), intent(inout), target :: g1
  real, dimension(l2, l2), intent(inout), target :: g2
  real, dimension(l3, l3), intent(inout), target :: g3

  !$acc data region copy(g1,g2,g3)
```

16
call f2(g1, l1)
call f2(g2, l2)
call f2(g3, l3)
!$acc end data region
end

! f2.f90
subroutine f2(grid, length)
implicit none
integer, intent(in) :: length
real, dimension(:,:), intent(inout), target :: grid
integer i, j
!
$acc reflecte(d)ed(grid)
!
$acc region
do j = 1, length
  do i = 1, length
    grid(i,j) = i*j
  end do
end do
$acc end region
end

HMPP Workbench  Within HMPP Workbench we could define a callsite/-codelet for each grid level. As a result the code would be hard to read and again, we only could deal with a number of grid levels known at compile time. Another approach is to make use of the data mirroring concept within the HMPP framework. List. 9 shows an example for how multi–grid computations could be realized with data mirroring. In the main program we allocate the work-array work in line 10. The actual computations are performed in subroutine grds. In line 9 the number of grid levels is defined (not necessarily known at compile time). For each grid level, a pointer variable is set on the starting address of the particular grid in line 20. For each grid level first accelerator memory gets allocated by the allocate–directive in line 22 and then the data gets transferred to the accelerator memory in line 25. In line 34 we call the grds–codelet for each grid level. The args[pt].mirror–directive in the codelet definition (line 46) indicates that the grid data is mirrored. By CPU memory addresses, accelerator memory addresses are resolved at run time automatically.

Listing 9: Data mirroring with HMPP for multiple grids

program grid_dummy
implicit none
type pp
  real, dimension(:), pointer :: p
end type pp
integer, parameter :: n = 50
integer, parameter :: grids = 3
real, dimension(n), target :: work
integer :: i
integer :: grid_size
type(pp), dimension(grids) :: array_of_pointers
work = -1
do i=1,grids
  grid_size = (i+1)*(i+1)
do
p =⇒ work( i+i+ i+i+ gr d size −1)  
! Allocates GPU memory for the grids  
!$\text{Allocates data["p"], data["p"].size={grd.size}}$

! Transfer grids from CPU to GPU  
!$\text{AdvancedLoad, data["p"],}$  
array_of_pointers(i)%p  
end do  
do i=1,grids  
grd size = (i+1)  
! Launch codelet  
!$\text{Calls grid (array_of_pointers(i)%p, grd.size})$  
! Transfer grids back to CPU memory  
!$\text{DelegatedStore, data["array_of_pointers(i)%p"]}$  
! Print the results  
print *, 'SUBARRAY', i  
print *, array_of_pointers(i)%p  
end do  
! A dummy codelet  
!$\text{Calls codelet, target=cuda, args[ptr].mirror, &}$  
!$\text{Calls grids, args[ptr].transfer=manual, &}$  
!$\text{Calls grids, args[n].transfer=auto}$  
subroutine grds(ptr,n)  
  implicit none  
  integer, intent(in) :: n  
  real, dimension(n,n), intent(out) :: ptr  
  integer :: i, j  
! Perform some dummy computation  
do i=1,n  
do j=1,n  
  ptr[i,j] = i * j  
end do  
end do  
end subroutine grds  
end program

4 Results

For the evaluation of the hardware accelerator frameworks we used a high-end consumer PC dedicated to scientific computing. Tab. 2 gives an overview of the hardware and software configuration. For all computations we compare the performance of a single core of an Intel Core i7-2600K@3.40Ghz CPU to a NVidia GTX580 GPU.

All presented run time measurements result from averaging the timings of multiple test runs.

4.1 BLAS routines

All CPU code was compiled and linked either using gfortran -O3, pgfortran -O3 or ifort -march=native -axAVX -O3. Clearly, one can expect ifort to deliver the best performance due to a compilation for the Advanced Vector Extensions (AVX) for Intel’s Sandy Bridge processor architecture. The release ver-
Table 2: Detailed description of the hardware platform used.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CPU</strong></td>
<td>Intel Core i7-2600K</td>
</tr>
<tr>
<td># Cores/CPU</td>
<td>4</td>
</tr>
<tr>
<td>Clock [GHz]</td>
<td>3.40</td>
</tr>
<tr>
<td>System Memory [GB]</td>
<td>16</td>
</tr>
<tr>
<td><strong>GPU</strong></td>
<td>GTX 580</td>
</tr>
<tr>
<td>GPU clock [MHz]</td>
<td>602</td>
</tr>
<tr>
<td># cores/GPU</td>
<td>512</td>
</tr>
<tr>
<td>GPU Memory [MB]</td>
<td>1535</td>
</tr>
<tr>
<td>Year</td>
<td>2011</td>
</tr>
<tr>
<td>OS</td>
<td>Ubuntu 10.10</td>
</tr>
<tr>
<td>Linux kernel</td>
<td>2.6.35-30-generic</td>
</tr>
<tr>
<td>CUDA Toolkit</td>
<td>4.0</td>
</tr>
<tr>
<td>PGI Accelerator</td>
<td>11.8</td>
</tr>
<tr>
<td>HMPP Workbench</td>
<td>3.0.0alpha</td>
</tr>
<tr>
<td>GNU gfortran</td>
<td>4.4.5</td>
</tr>
<tr>
<td>Portland Group pgfortran</td>
<td>11.8</td>
</tr>
<tr>
<td>Intel ifort</td>
<td>12.0.4</td>
</tr>
</tbody>
</table>

The HMPP Workbench timings are based on the _gfortran_ and _nvcc_ compiler at optimization level 3 (_-O3_). The PGI Accelerator timings are based on the _pgfortran_ compiler and _nvcc_ at _-O3_. A tuned ATLAS BLAS library (Core i7; SSE3) gives performance numbers similar to _ifort_ with AVX.

In the following, we will denote “CPU naive” versions of a scalar product computation and a matrix multiplication, as the subroutines _inner_ from lst. 4 on page 12 and _matmult_ from lst. 6 on page 13 executed on a CPU.

**Scalar product** Tab. 3 and tab. 4 present timings and speedups for the scalar product computation in § 4.1. Clearly, cuBLAS gives the best results (a speedup of 5.7 compared to the CPU BLAS implementation; 4.5 compared to _ifort_ AVX), followed by HMPP Workbench and PGI Accelerator for a sufficiently large vector size. These numbers indicate that memory bandwidth is the limiting factor on the GPU.

**Matrix multiplication** Tab. 5 and tab. 6 present timings and speedups for the matrix multiplication in § 4.1. For 4000 × 4000 matrices, cuBLAS outperforms the CPU BLAS (_ifort_ AVX) implementation by a factor of 123 (36) and achieves an instruction throughput of approximately 980 GFlops. Compared to the cuBLAS, PGI Accelerator and HMPP Workbench generate rather slow code. Still, taking into account the low programming effort with those frameworks, the numbers are good. The poor timings of the naive CPU matrix multiplication compiled with _gcc_ indicate that one also has to be aware of CPU programming techniques as...
4.2 Coupled 7-point stencil

Tab. 7 and tab. 8 present the timings and speedups for the coupled 7-point stencil computation with and without data transfers. A comparison shows, that data transfers are a significant bottleneck.

The CPU code was compiled using the pgfortran compiler using the -O2 flag. The -O3 flag does not result in a mentionable gain in performance. gfortran and ifort with -axAVX enabled give very similar timing results at -O3 and generate code which is approximately 20% less performant than pgfortran at -O2.

The rather naive CUDA implementation makes use of the on-board GPU texture cache, resulting in the best performance for small grid dimensions. From a certain grid size on, HMPP Workbench gives the best results with a speedup of almost 22 compared to the CPU implementation.

4.3 Framework Evaluation

Now we evaluate the frameworks regarding implementational aspects. This includes mapping effort, maintenance, integration, data management and applicability. Again, we emphasize that this evaluation is based on our experience on the examples evaluated only.

Mapping effort CUDA requires a significant mapping effort, as all tasks, such as parallelizing loops have to be coded by hand. For PGI Accelerator and HMPP Workbench a significantly decreased mapping effort can be expected for small portions of code. For larger codes, HMPP Workbench is more convenient. The callsite/codelet approach is better applicable.

Maintenance Code maintenance within PGI Accelerator and HMPP Workbench is convenient. For changes in the implementation of an algorithm, only one code has to be modified and only the corresponding accelerator directives need to be adapted. In case of CUDA, changes made, might require to manually optimize the CUDA implementation again to get satisfying performance results. If an additional CPU implementation is required, two codes need to be modified.

Integration The integration time for all frameworks is convenient. For CUDA code, the CUDA compiler nvcc has to be invoked manually. Linking from Fortran or other programming languages than C/C++ might be time consuming, but not difficult. PGI Accelerator and HMPP Workbench are distributed with all necessary software and libraries and can perform this automatically.

Data management CUDA allows the most flexible data management. A CPU version and an accelerator version of one set of data has to be declared separately. The mirroring concept of HMPP Workbench almost allows the same flexibility. The data management within PGI Accelerator currently is rather limited.
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^4$</td>
<td>8.00</td>
<td>3.99</td>
<td>2.99</td>
<td>96.80</td>
<td>114.55</td>
<td>21.23</td>
<td></td>
</tr>
<tr>
<td>$10^5$</td>
<td>81.00</td>
<td>54.60</td>
<td>35.19</td>
<td>141.86</td>
<td>211.90</td>
<td>24.53</td>
<td></td>
</tr>
<tr>
<td>$10^6$</td>
<td>851.18</td>
<td>321.12</td>
<td>407.68</td>
<td>443.71</td>
<td>370.47</td>
<td>72.17</td>
<td></td>
</tr>
</tbody>
</table>

Table 3: CPU and GPU timings for a single precision scalar product computation excluding GPU data transfer.

<table>
<thead>
<tr>
<th>Vector size</th>
<th>CPU naive gcc ÷</th>
<th>CPU naive ifort AVX ÷</th>
<th>CPU BLAS ÷</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PGI</td>
<td>HMPP</td>
<td>cuBLAS</td>
</tr>
<tr>
<td>$10^4$</td>
<td>0.08</td>
<td>0.07</td>
<td>0.38</td>
</tr>
<tr>
<td></td>
<td>0.04</td>
<td>0.03</td>
<td>0.18</td>
</tr>
<tr>
<td>$10^6$</td>
<td>0.57</td>
<td>0.38</td>
<td>3.30</td>
</tr>
<tr>
<td></td>
<td>0.38</td>
<td>0.25</td>
<td>2.20</td>
</tr>
<tr>
<td>$10^6$</td>
<td>1.92</td>
<td>2.27</td>
<td>11.80</td>
</tr>
<tr>
<td></td>
<td>0.72</td>
<td>0.86</td>
<td>4.45</td>
</tr>
</tbody>
</table>

Table 4: CPU and GPU speedup for a single precision scalar product computation excluding GPU data transfer.

<table>
<thead>
<tr>
<th>Matrix size</th>
<th>CPU naive gcc [ms]</th>
<th>CPU naive ifort AVX [ms]</th>
<th>CPU BLAS [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PGI</td>
<td>HMPP</td>
<td>cuBLAS</td>
</tr>
<tr>
<td></td>
<td>0.82</td>
<td>0.20</td>
<td>0.31</td>
</tr>
<tr>
<td></td>
<td>0.31</td>
<td>0.20</td>
<td>0.25</td>
</tr>
<tr>
<td></td>
<td>0.60</td>
<td>0.38</td>
<td>0.72</td>
</tr>
<tr>
<td></td>
<td>0.11</td>
<td>0.03</td>
<td>0.92</td>
</tr>
<tr>
<td></td>
<td>0.02</td>
<td>0.02</td>
<td></td>
</tr>
</tbody>
</table>

Table 5: CPU and GPU timings for a single precision matrix multiplication excluding GPU data transfer.

<table>
<thead>
<tr>
<th>Matrix size</th>
<th>CPU naive gcc ÷</th>
<th>CPU ifort AVX ÷</th>
<th>CPU BLAS ÷</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PGI</td>
<td>HMPP</td>
<td>cuBLAS</td>
</tr>
<tr>
<td></td>
<td>24.05</td>
<td>7.45</td>
<td>41.00</td>
</tr>
<tr>
<td></td>
<td>5.57</td>
<td>1.82</td>
<td>10.00</td>
</tr>
<tr>
<td></td>
<td>9.09</td>
<td>2.82</td>
<td>15.5</td>
</tr>
<tr>
<td></td>
<td>340.48</td>
<td>261.79</td>
<td>2028.68</td>
</tr>
<tr>
<td></td>
<td>4.45</td>
<td>3.43</td>
<td>26.54</td>
</tr>
<tr>
<td></td>
<td>15.72</td>
<td>12.09</td>
<td>93.68</td>
</tr>
<tr>
<td></td>
<td>470.42</td>
<td>368.26</td>
<td>3464.79</td>
</tr>
<tr>
<td></td>
<td>4.90</td>
<td>3.84</td>
<td>36.09</td>
</tr>
<tr>
<td></td>
<td>16.64</td>
<td>13.03</td>
<td>122.56</td>
</tr>
</tbody>
</table>

Table 6: CPU and GPU speedup for a single precision matrix multiplication excluding GPU data transfer.
<table>
<thead>
<tr>
<th>Grid dimension</th>
<th>CPU [ms]</th>
<th>PGI [ms]</th>
<th>HMPP3 [ms]</th>
<th>CUDA [ms]</th>
<th>PGI speedup</th>
<th>HMPP3 speedup</th>
<th>CUDA speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>38x29x56</td>
<td>6.7</td>
<td>38.8</td>
<td>4.6</td>
<td>n/a</td>
<td>0.2</td>
<td>1.5</td>
<td>n/a</td>
</tr>
<tr>
<td>75x57x111</td>
<td>29.2</td>
<td>53.6</td>
<td>22.3</td>
<td>n/a</td>
<td>0.5</td>
<td>1.3</td>
<td>n/a</td>
</tr>
<tr>
<td>149x113x111</td>
<td>115.2</td>
<td>96.4</td>
<td>72.9</td>
<td>n/a</td>
<td>1.2</td>
<td>1.6</td>
<td>n/a</td>
</tr>
</tbody>
</table>

Table 7: CPU and GPU timings and speedups for the coupled 7-point stencil including GPU data transfer (single precision).

<table>
<thead>
<tr>
<th>Grid dimension</th>
<th>CPU [ms]</th>
<th>PGI [ms]</th>
<th>HMPP3 [ms]</th>
<th>CUDA [ms]</th>
<th>PGI speedup</th>
<th>HMPP3 speedup</th>
<th>CUDA speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>38x29x56</td>
<td>6.7</td>
<td>0.7</td>
<td>0.7</td>
<td>0.2</td>
<td>9.6</td>
<td>9.6</td>
<td>27.9</td>
</tr>
<tr>
<td>75x57x111</td>
<td>29.2</td>
<td>2.4</td>
<td>1.8</td>
<td>1.5</td>
<td>6.2</td>
<td>16.2</td>
<td>19.5</td>
</tr>
<tr>
<td>149x113x111</td>
<td>115.2</td>
<td>6.7</td>
<td>5.3</td>
<td>6.1</td>
<td>17.2</td>
<td>21.7</td>
<td>18.9</td>
</tr>
</tbody>
</table>

Table 8: CPU and GPU timings and speedups for the coupled 7-point stencil excluding GPU data transfer (single precision).

## 5 Conclusion

In this technical report we have evaluated two different compiler–based hardware accelerator frameworks for multiple criteria and compared them to CUDA.

Implementations with CUDA will give the best performance, but only if they are coded carefully. This usually is time consuming and requires a good understanding of the hardware accelerator. Currently CUDA is limited to NVidia GPUs. PGI Accelerator and HMPP Workbench are both easy to learn and convenient to use. Depending on the application both frameworks do have their limitations but the programming model of HMPP Workbench is more evolved and gives more control over the accelerator code generation. HMPP Workbench now supports CUDA and OpenCL. On the other hand, for small portions of isolated code, PGI Accelerator has a sophisticated CUDA code generator and can apply several optimization strategies automatically. A disadvantage of both frameworks is the lack of a vital user community. Their code generators produce satisfying accelerator code but it is not comparable to specialized hardware accelerator libraries. If such libraries are available, they should be used to speed up computations.

Both frameworks are evolving rapidly and it is hard to foresee their future capabilities. Even if their application does not seem practical for a certain application now, checking their features from time to time is a good idea. A general decision about which framework is the best is not possible, as the requirements strongly depend on the application. Nonetheless, we hope that we could give a satisfying introduction to PGI Accelerator and HMPP Workbench.

OpenACC (and OpenHMPP as well) are genuine efforts for a standardization of directive–based accelerator frameworks. Multiple companies are involved in
Table 9: Some rough and category based estimates on the frameworks. They are based on our experience gathered during their evaluation.

<table>
<thead>
<tr>
<th></th>
<th>CUDA</th>
<th>PGI</th>
<th>HMPP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mapping effort</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Maintenance</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Integration</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Data management</td>
<td>+</td>
<td>0</td>
<td>+</td>
</tr>
</tbody>
</table>

The corresponding committees. By an increasing number of hardware accelerators the need for hardware independent programming paradigms will become stronger and directive–based frameworks will become more popular and widespread. As a result, the success of compiler vendors will be strongly based on the number of hardware architectures supported and by the quality of the code generators.

Let us now briefly summarize the conclusion from above:

- For the acceleration of small portions of code, PGI Accelerator may be an excellent choice. Its compiler performs many optimizations automatically and the programming model is very to learn.
- For more complicated algorithms and applications, consisting of several procedure levels and complex data structures, HMPP Workbench clearly is the better choice.
- Currently PGI Accelerator is limited to CUDA devices. HMPP covers the most hardware accelerator due to support for CUDA, OpenCL and a future support for Pthreads.
- For time critical applications, optimized libraries should be used if available, instead of code generated by hardware accelerator frameworks.
- The user community for PGI Accelerator and HMPP Workbench is small. Questions regarding the frameworks will have to be directed directly to the vendor’s support.

**References**


