Evaluation of CUDA Fortran for the CFD code “Strukti”

Practical term report from Stephan Soller
High performance computing center Stuttgart[1]
Stuttgart Media University[2]

1. http://www.hlrs.de/
2. http://www.hdm-stuttgart.de/

High performance computing center Stuttgart

The high performance computing center Stuttgart[3] (HLRS) was founded in 1986 as part of the university of Stuttgart. Since then it enables scientists and engineers to use up to date super computing capabilities. In cooperation with researchers and the industry the HLRS helps to solve scientific and engineering problems using their super computers.

The HLRS is part of the Gaus center for super computing[4], an alliance of three German super computing centers (Jülich supercomputing center, Leibnitz-Rechenzentrum Garching and the HLRS). This alliance allows to use the advantages of the different cluster and super computing systems and provides scientific education. Throughout the whole year different courses are held at these computing centers, covering a wide range of topics: physics, numerics, parallel programming and new computing technologies.

Currently the HLRS hosts 5 super computers with a 6th one currently under construction:

- NEC SX-8: 1.2 TFlops with 80 processors
- Cray XT5m: 8.5 TFlops with 112 Dual Sockel Quad-Core AMD Opteron nodes, 16 GByte memory per node
- NEC SX-9: 19.2 TFlops with 192 processors
- HLRS Grid-Cluster: about 46 TFlops with 545 nodes
- NEC Nehalem Cluster: 62 TFlops with 700 Dual Sockel Quad Core Intel Xeon nodex, 12 GByte memory per node

3. http://www.hlrs.de/

4. http://www.gauss-centre.eu/

Project goals and motivation

The goal of the projects was to evaluate the usage of graphics processors (GPUs) to accelerate a computational fluid dynamics (CFD) simulation. The simulation code (called “Strukti”) was provided by the institute of aerodynamics and gas dynamics (IAG) of the university Stuttgart. At the start of the project the code was optimized for single thread performance and did not use any explicit parallelization.

The project was motivated by the recent developments in the area of GPUs. During the last decade these graphics processing units became more and more programmable. This made them an attractive platform not only for fast realtime graphics but also for general purpose calculations. They are designed for very high calculation throughput which is currently at about 0.7 to 2.7 TFlops *per GPU*[5].

However this performance comes at a cost: The architecture is centered around the needs of realtime computer graphics. While this pipeline contains large parts that can be used for general purpose computations not every problem can be programmed in a way that GPUs can efficiently execute it.

Nevertheless GPUs became popular in high performance computing over the past years. The Nehalem cluster is no exception to this. 32 of its nodes are equipped with nVidia Tesla S1070 GPUs. Additionally some nodes contain the more recent Tesla C2070 GPUs.

5. AMD Radeon™ HD 6970 GPU Feature Summary as of May 27th, 2011. http://www.amd.com/US/PRODUCTS/DESKTOP/GRAPHICS/AMD-RADEON-HD-6000/HD-6970/Pages/amd-radeon-hd-6970-overview.aspx

Used programming languages

Several programming languages were used during the project. Therefore a short overview of the most important aspects of these languages follows.

nVidia CUDA C

The direct way to harness the computational power of the nVidia Tesla GPUs is to use the programming interface nVidia provides: CUDA, “Compute Unified Device Architecture”. The name “CUDA” describes a whole range of tools used to write software for nVidia GPUs. The most basic part is an compiler for CUDA C, an extended and customized C++ like language. It adds constructs to execute code on the GPU and to access other GPU functions.

Basic architecture

The most important aspect when programming in CUDA or on GPUs in general is parallelism. GPUs achieve their high performance by massively parallel execution of code, often doing several hundred calculations at the same time. In contrast modern quad core CPUs can do 16 single precision floating point calculations at the same time using special SSE instructions on all four cores.

This massive parallelism is needed for GPUs to be efficient. With hundreds of execution units the GPU really needs at least an equal number of threads to keep those units busy. To keep the execution units simple they do not contain much of the complex logic to hide the latencies of memory access. Instead this is done by keeping a large number of threads ready and execute them as soon as the needed data arrives from the memory. This effectively hides memory latency and keeps the units simple as long as there are enough threads available. This approach allows GPU manufactures to build very space efficient calculation chips.

However this architecture requires the programmer to feed a GPU with an even larger number of threads than it can handle. Otherwise memory latency will reduce performance as the execution units wait for data and cannot execute other threads in the mean time.

In computer graphics this usually isn't a large problem since this area provides some intrinsic ways to parallelize the calculations. For example a display at an resolution of 1280·1024 consists of 1.310.720 pixels. The color values of each of these pixels can usually be calculated in parallel. Additionally GPUs often perform calculations on all vertices of a model. In computer games or

visualization applications such models can easily consist of several thousand or more vertices. This potential parallelism has lead to the massively parallel architecture of GPUs. To use them efficiently we need to program in such a massively parallel way.

Programming model

This directly reflects in the programming model used by CUDA C. While normal code written with CUDA C is interpreted in the same way as C++ code, special function definitions are used to declare code that is meant to be run on the GPU. Such functions are called “kernels”. They contain the instructions of a single thread and when a kernel is executed the programmer needs to define how many times it is to be executed in parallel. A kernel usually reads it's current thread index (e.g. 42 of 320.000) and uses it to determine the piece of memory it should work on. Basically all threads are doing the same but on different data. This principle is called SIMT (Single Instruction Multiple Threads) or more general SIMD (Single Instruction Multiple Data). It is also the basic idea of the SSE special instruction set of x86 CPUs, however on a much smaller scale.

For a kernel to work with data that data must be accessible by the GPU. Traditional programs store their data on the main memory of the computer, the so called RAM. However this memory is not fast enough to be used for computer graphics and therefore a GPU provides it's own high speed memory to operate on. In CUDA this is called the ”global memory” of the compute device. A kernel can only work on data that is stored in there, much like a CPU can only work on data that is stored in the RAM.

Therefore we have to move all data we want to process to the GPUs global memory before we execute the kernel. After the kernel finished we copy the result back to the main memory and use the CPU for further processing (e.g. writing the data to a file). This results in a very common workflow in CUDA:

1. Prepare data with the CPU in the main memory
2. Move data to the GPUs global memory
3. Perform calculations or other operations on the GPU by executing a kernel
4. Move the results of the kernel back to the main memory
5. Continue processing on the CPU

In CUDA the GPU cannot do useful calculations on its own. It's rather a very powerful co-processor that needs to be controlled by a CPU program. Basically what the DMA controllers are for I/O the GPU has become for massively parallel calculations. However while the DMA

controllers, that handle input and output data transfers on the CPUs behalf, don't require explicit management a GPU cannot function without explicit control by the programmer.

Background

Since a program written with CUDA C contains code for both, the CPU (or “host” in CUDA terminology) as well as the GPU (or “device”), some additional processing is done in the background while compiling and executing CUDA C source code.

The CUDA C compiler nvcc separates host and device code. The host code is compiled with an usual C++ compiler and x86 machine code is generated for it. The device code however is compiled into the assembly like PTX (Parallel Thread Execution) code. These two are then combined into a single executable.

When a kernel is called during the runtime of a program the PTX assembly code is send to the CUDA runtime that manages the GPU device. The CUDA runtime then compiles the PTX code into instructions the currently used GPU supports and executes the kernel.

The PTX code is a good way to verify that the GPU actually does what the CUDA C code is meant do to, just like Assembler code is for the CPU.

Fortran

The computational fluid dynamics code “Strukti” is written in Fortran, a programming language very popular in the high performance computing sector. Fortran has been available since 1957 and is still actively maintained and developed. It's being used by many scientists and engineers for heavy computing tasks like numerical computation. Despite it's somewhat historic image in other industry branches it's well suited for that task.

- Performance: Fortran compilers can usually produce very fast executables for Fortran code. This is due to the optimization friendly language and the advanced optimization techniques used by the compilers.
- Backward compatibility: It's still possible to compile very old Fortran code like Fortran 77 which was still written in a format suitable for punch cards. This code can than be linked and used together with code written in a more current Fortran version.
- High level: Fortran code can be written without focusing to much on the lower levels. While it's still necessary to do memory allocation manually it's not necessary to use pointers for

this as it is in C. Most tasks can be handled without thinking about the underlying mechanism. Array manipulation does not require pointer arithmetic for example.

Because of its popularity in high performance computing (HPC) many super computers have one or more Fortran compilers installed. Thanks to that it's possible to write Fortran code that runs on different super computers.

Despite these advantages some parts in Fortran also show its age and long history of compromises and compatibility:

- Backward compatibility: While it's an advantage to reuse old code, much Fortran code tends to use a mixture of old and new language constructs. This doesn't always results in clean code and sometimes old habits are continued even if there is no need to.
- I/O handling: While Fortran provides build in ways to do I/O these are rather limited when compared to the I/O facilities of e.g. the C standard library. Some features are also compiler dependent.
- Precision handling: In Fortran there are different ways to handle how precise floating point variables are. It's possible to use normal types (`real` and `double precision`), set compiler specific type attributes (`real (kind=8)`) or use compiler switches to set the precision of build in types.

CUDA Fortran

There are several ways to write code for nVidia CUDA enabled GPUs. In a C environment CUDA C and OpenCL can be well combined with other C or C++ code. However since Strukti is written in Fortran interoperability with Fortran is a key aspect. Therefore we take a closer look at CUDA Fortran, an extension to the PGI Fortran compiler that allows to write CUDA kernels directly in Fortran.

However CUDA Fortran adds another preprocessing step to the workflow of the GPU code and therefore potentially decreases performance of the resulting kernel. Because of that we take a closer look at CUDA C and CUDA Fortran with the aim to determine the cost of using CUDA Fortran instead of CUDA C. Since Strukti is a large Fortran code base the ability to write CUDA code directly in Fortran can avoid the work to port code to CUDA C.

Test environment

All test code is run on one node of the HLRS Nehalem cluster. The GPGPU enabled nodes are equipped with two Intel Xeon Quad-Core X5560 CPUs and two nVidia Tesla T10 GPUs. For the sake of simplicity in this comparison only one CPU thread and one GPU is used. Therefore the results can *not* be used to compare CPU vs. GPU speed.

Test structure

To test the performance of both languages a matrix multiplication algorithm is implemented in both of them (CUDA C and CUDA Fortran). A simple CPU version is used to calculate the expected result which is then used to check the product calculated on the GPU. The GPU algorithm will be optimized in several stages to see how much one particular optimization can speed up the algorithm.

CUDA C and CUDA Fortran do not share the exact same feature set. Therefore the tests for both languages are not exactly equal:

- CUDA C provides access to the GPUs texturing subsystem which can be used to cache global memory access to some degree but it only supports single precision floating point data. At the time of writing CUDA Fortran did not allow usage of textures and therefore this optimization is only used in the CUDA C test when compiled with single precision.
- CUDA C allows loop unrolling using the `unroll` pragma. This is used to unroll the loop in the unaligned shared memory test with an optimal block size of 16. In CUDA Fortran however this does not work for GPU code as all unrolling instructions and options are ignored for it. The optimizer however unrolls loops in blocks of 4 when compiled with `-O3` as was done in this test. Therefore the unaligned shared memory test is less efficient in CUDA Fortran.
- In CUDA Fortran the `matmul()` intrinsic is used to calculate the reference matrix product. For sake of comparison however a naive CPU based matrix multiplication is also implemented in CUDA Fortran (same algorithm as in CUDA C).

With that in mind this is the list of matrix multiplication versions the test contains:

- Simple CPU implementation.
- CUDA Fortran only: intrinsic `matmul()` function.
- Naive GPU implementation without any optimizations. This is the worst case scenario with two global reads and one global write per multiplication.

- Naive GPU implementation which accumulates the sum in a local variable (register) and safes it to global memory when done. This reduces the number of global writes to one per element of the result matrix.
- CUDA C with single precision only: Like the write optimized version but global reads are done through textures.
- Like the write optimized version but each block uses shared memory to avoid global reads. This is the algorithm found in the CUDA C and CUDA Fortran programming guides with some additions to allow arbitrary matrix sizes.
- CUDA C only: The shared memory version above but the shared memory loop is unrolled with a block size of 16 (the size of the shared memory block).
- A shared memory version which pads incoming matrices to fit the block size of the shared memory. Except for the padding this is the algorithm of the CUDA manuals. Because of the constant loop count the CUDA Fortran compiler automatically unrolls the shared memory loop.
- CUDA C only: The above algorithm but with an `unroll` pragma to unroll the shared memory loop.

The CUDA C as well as the CUDA Fortran test are both compiled and run for single and double precision. The calculation is repeated 1000 times without moving the data to or from the GPU (all algorithms are idempotent).

The results

The result of the different tests. All values are in GFlops/s.

Test	Single precision			Double precision		
	CUDA C	CUDA Fortran	Speed	CUDA C	CUDA Fortran	Speed
cpu naive	1.34	1.89	141%	1.11	1.73	156%
cpu intrinsic		10.62	-		4.13	-
gpu naive	14.58	10.13	69%	8.64	6.90	80%
gpu naive write opt	25.54	17.40	68%	16.31	12.82	79%
gpu texture mem	45.91		-			-
gpu shared mem	73.00	70.11	96%	33.94	35.30	104%
gpu shared mem unroll	91.98		-	43.51		-
gpu shared mem aligned	154.54	169.64	110%	53.38	55.70	104%

The results show some interesting points:

- Even the naive GPU implementation can perform as well as the obviously optimized Fortran intrinsic `matmul()`.
- Write optimization (keeping the sum in a local variable) and manual unrolling in CUDA C are very cheap optimizations code vise with good performance gain. In CUDA Fortran using `-O3` comes nearest to unrolling.
- For CUDA Fortran using shared memory and using aligned memory are the two optimizations with the biggest performance gain. However these are the two most time consuming and expensive optimizations to do.
- That CUDA Fortran outperforms CUDA C in the `gpu shared mem aligned` might be due to not equally well optimized memory access in the CUDA C version.

Remarks

- When arrays are specified as kernel parameters CUDA Fortran moves the array descriptors to the global memory, too. This is a data structure containing information about the array (range of the dimensions, type, kind, ...). When intrinsics like `size()`, `shape()`, `lbound()`, etc. are used in GPU code these information are read out of the descriptor in the global memory. Depending on the kernel code this can impact performance: increased memory access but also more instructions used to calculate the pointer offsets to read the descriptor. This is best visible in the generated CUDA C and PTX code.

  To avoid this behavior it's best to not use intrinsics that read data from the descriptors but pass this information to the kernel as extra value parameters (`value` type attribute).

- The PGI CUDA Fortran compiler ignores unroll directives and unroll compiler options in GPU code. These rules are simply not applied to the generated CUDA C code. This basically makes manual unrolling impossible and can have an significant impact on performance. Therefore loops with constant loop count should be used whenever possible.

Thanks to this performance test several performance guidelines for CUDA Fortran have been developed:

- Use padded memory and loops with constant loop count whenever possible. This greatly improves memory access as well as calculation performance.
- Don't use intrinsics to read data from array descriptors within the kernel. Pass this information as extra kernel parameters.
- Don't trust CUDA Fortran and it's automatic operations. While nice when they work there are still hard to find bugs.

Using CUDA Fortran in Strukti

CUDA Fortran in itself worked reliable in the performance test above. However several problems arose when it was used within Strukti. The problems found in the performance test only decreased performance, but many of the problems found while using it with Strukti lead to wrong calculation results or to major architecture drawbacks. Therefore these bugs had to be taken care of before continuing the parallelization of Strukti.

Broken array descriptors

As CUDA Fortran kernels are almost like normal subroutines in Fortran they accept Fortran data types as parameters. Since kernels usually work on arrays it's straight forward to pass Fortran arrays as parameters to kernels. This is supported by CUDA Fortran and worked in the performance test above. However in Strukti passing arrays to kernels resulted in corrupted data.

This behavior was first spotted because all kernels in Strukti returned values very close to zero (e.g. `0.7e-307`). This data did not in any way resemble the output of the same algorithm run on the CPU.

After extensive debugging the source of this data corruption was found in the handling of the array descriptors. These descriptors are compiler specific runtime data structures that contain information about a variable. Specific intrinsic functions then read information from these descriptors like the lower or upper bounds of an array dimension. This is somewhat similar to runtime reflection information used in other languages.

When an array is passed to a kernel its data as well as its descriptor is copied to the global memory of the GPU device. The compiler then inserts special code into the kernel to read the required information from there. This inserted code does extensive pointer arithmetic as well as several reads on the global memory. The performance impact of that was visible in the performance test (see remarks section above).

As most arrays in Strukti use 6 or 7 dimensions the size of the array descriptor increases to store the lower and upper bounds of all dimensions. However the compiler generates kernel code seems to not understand the new descriptor layout and reads the wrong information. This of course breaks all kernel code that uses intrinsics like `lbound()` and `ubound()` to loop over all elements in an array. Additionally the compiler generated code to store an array value at a specific index writes the requested data to the wrong position since it relies on the descriptor data of the array to calculate the offsets.

To work around this bug we need to pass all relevant dimensions (e.g. the number of elements in the x direction) directly as kernel parameters. These parameters can then be used to define the dimensions of the array parameter. This way we can still use the built in array handling of Fortran but don't rely on the correct handling of the array descriptor. As a further positive side effect this also eliminates the various global reads performed by the generated array descriptor handling code. Note that scalar values like these dimensions are best passes by value which can be done by adding the `value` attribute to the argument type definition. Otherwise each value is stored in the global memory and then read from there.

In the performance test of CUDA Fortran this approach was only used to increase performance. In Strukti however it's necessary for the PGI compiler to produce correct code.

Corrupt memory transfer

On several occasions data transfer from and to automatically allocated device arrays (`device` attribute set but not `allocatable`) corrupted the data in question (e.g `0.2...e-266` or `Inf` instead of `0.081...`). We were not able to find a reliable way to reproduce the bug since it only appears in Strukti and even there only sporadically. On such situations marking the array as `allocatable` and allocating it with `allocate` worked around the issue.

Fortunately this bug appears to be fixed with version 11.1 of the PGI compiler. It was not encountered in Strukti with this version.

Small deviations from CPU results

Towards the end of the project small differences between the CPU and GPU results were encountered. These differences were in the size of 0.5e-14 for an individual kernel execution but resulted in noticeable differences in the end results. After extensive debugging the source of these deviations was found in the "fused multiply-add" (FMA) instruction of the Tesla T10 GPUs.

Fused multiply-add is a special instruction for calculating `a·b + c`. Since this term is quite common in computer graphics many GPUs support such instructions. However these calculations are usually performed without rounding the intermediate result of `a·b`. On CPUs such terms are calculated by an multiply instruction followed by an add instruction and after each the result is rounded.

While the calculation remains the same different rounding behavior occurs. On the GPU only one rounding operation is performed[6], on the CPU two rounding operations occur. Since the IEEE 754

6. nVidia PTX ISA 2.2, fma Floating-Point Instruction, page 87

floating point standard requires two rounding operations the FMA instruction of the Tesla T10 GPU is not IEEE 754 compliant. The calculations on the GPU might be more accurate but when validating the correctness with CPU results these differences are undesired.

To suppress FMA instructions in GPU code the undocumented PGI compiler switch `-Mcuda=nofma` can be used. This option is passed on to the `nvcc` compiler which in turn does not generate code that contains FMA instructions.

Please note that this is only necessary when validating GPU computations by comparing them with CPU computations. FMA instructions should actually increase the accuracy of the whole simulation. Therefore it would be worthwhile to compare the GPU results with an exact result if that's possible.

Porting workflow to isolate bugs

In the first naive approach to parallelize Strukti several kernels were written for well parallelizable CPU subroutines. Compiler bugs were not regarded as an possible error source at that time. However the mentioned bugs prevented these kernels from calculating correct results. Because of that the kernels were rewritten several times in the hopes to find the error within the kernel code.

Since debugging support for GPUs is currently very limited another way to isolate the error source was applied:

1. Replace the original CPU subroutine with a verification subroutine that will execute both, the CPU code as well as the GPU code, and verifies that the results are equal. Normal floating point inaccuracy has to be taken into account here so the verification is not an exact match of the bit patterns but rather a check if the difference between two values is still within a certain tolerance.
2. Use a duplicate of the CPU code for the GPU code (both are normal subroutines at that stage).
3. Transform the code of the GPU version to match the GPU execution model
    - Use nested loops to simulate the parallelism of a kernel call (e.g. a loop for every dimension of the grid and block configuration).
    - Perform some necessary optimizations for the GPU, like using sum variables.
4. Test the code on the CPU. Since both versions are pure CPU subroutines right now the code meant for the GPU can be handled and debugged like any other CPU function.

5. Wrap the GPU code into an actual kernel and test it.

Everything happens on the CPU except for the last step and the program is compiled in the usual way without using CUDA. This way the logic in the GPU code can be verified to be correct. Only the last step actually involves the GPU and requires the PGI compiler to generate real GPU code. If the GPU code was not changed when wrapping it into a kernel but produces wrong results it's an indicator for an compiler bug.

While finding the actual source of the bug is still difficult the search area can be reduced to the tools involved. If all calculations in step 4 are correct errors in the algorithms can be ruled out safely.

No external access to module device variables

The code of Strukti is divided into several Fortran modules. All modules are compiled separately and then linked together to generate the final executable. This provides some degree of encapsulation. However many interdependencies between the modules exist. These interdependencies are not only in the form of subroutine or function calls. Many subroutines directly access global variables of other modules and work with them.

Unfortunately the same architecture cannot be used for variables stored on the GPU device or kernels. While device variables and kernels are declared in the same way as normal variables and subroutines they are not symbols a linker can link against. A “symbol” in this context is something a linker can search and import from other compiled modules or object files to build the final executable. The device variable and kernel declarations are in fact removed before the object file is created and replaced with GPU specific code. Therefore code of other modules cannot work with device variables, thus breaking the code organization currently used in Strukti.

Another consequence of this limitation is that kernels can only use device variables declared within the same module. Since Strukti relies heavily on global variables that are used by many different modules every kernel and device variable would need to be in the same module. This would lead to one module that basically contains all Strukti code, breaking not only encapsulation but also the possibility to do incremental builds.

The C preprocessor can be used as a possible workaround. With the `#include` directive this one large module could at least be split into multiple source files matching the current source organization of Strukti. However this does not allow incremental builds as this large module would still need to be compiled in one large block.

Project results and further directions

The project has shown that CUDA Fortran can be used within Strukti to utilize the computational power of GPUs. But the compiler bugs and design drawbacks of CUDA Fortran require some compromises when used in Strukti:

- Pass array dimensions as explicit parameters to kernels.
- Do not use the FMA instruction when verifying code against the CPU results.
- Keep all device variables and kernels in one large module.
- Use the porting workflow described above to isolate compiler bugs if necessary.

Performance

Due to the large amount of time spent on various compiler bugs it was not possible to parallelize large parts of Strukti and evaluate the actual performance. The experience gathered with the matrix multiplication algorithm however allows us to do a rough estimation of the performance increase depending on how much we optimize a kernel.

Several of the optimizations applied in the performance test are easier to implement than others. The sum variable optimization for example requires only a new variable and some syntactic changes. There is no need to touch or change the logic of the code in question. Using the text memory or cache also does not require a change in the logic, however it can only be used with single precision and only in CUDA C. Additionally the code to manage texture memory in CUDA C can get rather complicated but still the logic of the code we want to execute requires no modifications. Shared memory and padded data structure on the other hand require complex changes in the logic of the existing code. Therefore these optimizations are more time consuming.

The computing power of these easy optimizations is well accessible. In the process of a first parallelization of Strukti most kernels would probably be optimized in that way. If we take a look at the matrix multiplication test this would be about 13 GFlops. If we compare this with an easy OpenMP parallelization that utilizes 4 cores we're at about 7 GFlops. When all 8 cores are used 14 GFlops are possible. However since each GPU enabled node of the Nehalem cluser contains two quad-core CPUs as well as two Tesla GPUs it's fair to compare one CPU with on GPU. But it should also be noted that it's way easier to extend the OpenMP parallelization from 4 to 8 threads than managing the CUDA Fortran code to use two GPUs.

If the question would be to decide between an CPU based parallelization (e.g. with OpenMP) and GPU based parallelization with CUDA Fortran the mid term performance gain of GPUs is not higher than that of CPUs. However the GPU code contains much more potential for manual optimizations and therefore might be a good strategic choice.

Please note that this projection is based on the performance stats of the matrix multiplication and therefore is not a reliable base. Algorithms in Strukti vary in their complexity and memory access patterns. However this comparison should provide a fair view on the advantages and disadvantages of a GPU based parallelization.

Further directions

To continue the development and optimization of Strukti several possibilities are available:

- Continue to use CUDA Fortran to port more subroutines of Strukti to the GPU
- Take a look at the PGI Accelerator[7]
- Use libraries that utilize the GPU instead of handling it directly

Using CUDA Fortran to port more code to the GPU was already discussed above. PGI Accelerator is an compiler extension like CUDA Fortran. But instead of writing kernel code directly only OpenMP like directives are used to annotate loops. This should result in a much more maintainable code base than using CUDA Fortran but needs further testing. Not using the GPU directly but instead relying on libraries to utilize the GPU may sound wasteful at first. This however allows to reduce the complexity and might fit very well when combined with an OpenMP or MPI based parallelization.

To utilize the full potential of GPUs however all these approaches have drawbacks. CUDA Fortran kernels require extensive and difficult optimizations as well as hardware specific tuning for each kernel. With the high abstraction level of PGI Accelerator or libraries it's doubtful that the GPU will be used to it's full extend. To really achieve full utilization a different overall programming model like GRAMPS[8] that is partially implemented e.g. in OpenCL might yield better overall results. However these technologies are not quite ready yet and would require much more time to evaluate than currently available tools like CUDA.

7. http://www.pgroup.com/resources/accel.htm
8. Sugerman, J., Fatahalian, K., Boulos, S., Akeley, K., and Hanrahan, P. 2009. GRAMPS: A programming model for graphics pipelines. ACM Trans. Graph. 28, 1, Article 4 (January 2009), 11 pages. DOI = 10.1145/1477926.1477930. http://doi.acm.org/10.1145/1477926.1477930