A GPU Approach to Fortran Legacy Systems

Mariano Méndez, Fernando G. Tinetti[*]
III-LIDI, Facultad de Informática, UNLP
50 y 120, 1900, La Plata
Argentina

Technical Report PLA-001-2012[†]
October 2012

Abstract

A large number of Fortran legacy programs are still running in production environments, and most of these applications are running sequentially. Multi- and Many- core architectures are established as (almost) the only processing hardware available, and new programming techniques that take advantage of these architectures are necessary. In this report, we will explore the impact of applying some of these techniques into legacy Fortran source code. Furthermore, we have measured the impact in the program performance introduced by each technique. The OpenACC standard has resulted in one of the most interesting techniques to be used on Fortran Legacy source code that brings speed up while requiring minimum source code changes.

1 Introduction

Even though the concept of legacy software systems is widely known among developers, there is not a unique definition about of what a legacy software system is. There are different viewpoints on how to describe a legacy system. Different definitions make different levels of emphasis on different characteristics, e.g.: a) "*The main thing that distinguishes legacy code from non-legacy code is tests, or rather a lack of tests*" [15], b) "*Legacy Software is critical software that cannot be modified efficiently*" [10], c) "*Any information system that significantly resists modification and evolution to meet new and constantly changing business requirements.*" [7], d) "*large software systems that we don't know how to cope with but that are vital to our organization.* "[4]. Thus, some legacy systems' characteristics are [13]

1. The software resistance to change.
2. The inherent complexity.
3. The crucial task performed by the software in the organization.
4. The system size, generally medium or large.

In the specific case of Fortran, there is software that has been running in production environments for 20 or 30 years. Throughout this period, software gradually deteriorates and it may need different types of changes, such as: enhancements, corrections, adaptations and preventions. All of these tasks require knowledge and comprehension of the system. In the multi-core and many-core era, software changes become more and more complex. In this article we will explore one side of the complexity that involves the parallelization of a legacy software by using GPU (Graphic Processing Units) devices. In order to achieve that, we will focus on advantages and disadvantages of these programming techniques to be applied on a Fortran legacy system.

[*]Investigador Comisión de Investigaciones Científicas de la Prov. de Bs. As.
[†]PLA: Parallel Linear Algebra

2 Fortran Legacy Systems

Fortran is one of the most long-lived programming languages used by scientists all around the world. It is considered the first high level language, it was born in 1954 [1]. This fact makes Fortran a programming language which has millions of source code lines already written. It became very popular specially in two fields, scientific programming and High Performance Computing (HPC), both closely related. In addition, due to its long-life, Fortran has gone through a particular standardization process in which each previous version of the standard is compliant with the current one [14]. This standardization process makes a FORTRAN 77 program still compilable with modern Fortran 2008 compilers. All these features help Fortran language to be, even today and contrary to the popular belief, in a solid and well defined position. Nowadays, there is a big set of Fortran programs running in the production environment of Universities, Companies and Governmental institutions. Some good examples can be seen in programs like climate models, earthquake simulations, magnetohidrodynamics simulation, astronomical software, quantum physics software, quantum chemistry software, and so forth. Most of these programs have been programmed years (or decades) ago and their users need them to be modernized, improved and/or updated. Modernizing this kind of programs also implies making them capable of taking advantage of the modern processors's architecture and, specifically numerical processing facilities. The modernizing and parallelization process/es is/are still far away from being performed automatically, due to the complexity associated to it/them. One task that is complex to achieve regarding the automation lays in the fact that today there is not tool capable of automatically parallelizing a sequential program.

3 Multi-core Architecture

Hardware manufacturers were able to improve microprocessors' performance almost continuously by combining technological and architectural design enhancements. The multi-core era was *officially acknowledged* by 2004/5, even when the trend could be noted some years before [9] and, in fact, the first multi-core design was released by IBM in 2001: the POWER4 processor [2]. Nevertheless, the architectural improvement implied by multi-core building has it own set of bottlenecks. One of the most important problems that chip-makers found was power consumption, or the *power wall*. The Intel Pentium 4 processor almost reached 100 watt per cm^2, this fact made Intel interrupt its Pentium 4 production. Even while Moore's Law was still valid, reality brought the heat level processors close to that of a nuclear reactor. Another drawback is the fact that while transistors ran faster, wires got slower this is known as "The wire delay problem". Other problems are not specifically hardware/design related but strongly affect performance, such as the one referred to as the "Memory Wall" problem. More specifically, if one core is several orders of magnitude faster than memory, the difference will be proportionally worse when several cores share (access) the same memory.

Finally, one relevant aspect to include is the fact that in these days most of prior existent programs do not use the multi-core parallel capabilities of the multi-core chips. Even when compilers have been improved, they are not able to rewrite or transform sequential programs into the corresponding parallel versions of them. They are capable of improving programs by applying, for example, instruction level parallelism, but introduction of automated multi-core parallelism is at a very early stage of development.

4 NVIDIA/CUDA Many-core Architecture

Many-core architecture is used for describing the environment/processor design where several hundreds or thousands of small cores compute in parallel, as a different approach of that taken by multi-core designs involving a small set of complex cores (2, 4, 6, 8) [5] [23]. As a result of this approach, an extremely good performance has been achieved within *reasonable* power consumption rates. The idea that underlies many-core architecture is that even when simple processors are slower (e.g. 3 times slower than a complex processor) they simple processors are more than ten times smaller than complex ones so the total throughput of many-core processors outperforms that of the more complex processors. These days, one of the most illustrative examples of many-core architectures is the so called GPGPU (General-Purpose Graphics Processing Units) which aims to utilize graphic processing units, commonly focused on computer graphics, to perform computation on programs managed by a central processor unit(CPU) [6, 18]. In 1999, the NVIDIA Gforce 256 was released, and it was "a single-chip processor with integrated transform, lighting, triangle set up/clipping, and rendering engines that is capable of processing a minimum of 10 million polygons per second". NVIDIA 256 had a graphic pipeline architecture, configured as follows: Pixel shader, Vertex shader, a Texture mapping unit, and Render Output unit. Its precise core configuration was 4,0,4,4 respectively [17, 18, 11]. The next big advance in GPU architecture arrives on the hands of G80 based on the unified shader architecture. This architecture supports the unified shader model which consists of the same instruction set used for any kind of shader, as shaders can read from data buffers, textures and perform the same arithmetic operation. Basically, a GPU is composed of a set of Streaming Processors (SP) organized in Streaming Multiprocessors (SM). Each SM include several Special Function Units (SFUs) as shown in Fig. 1. Several SMs, in turn, are grouped in independent processing units called Texture/Processors Clusters (TPC) [11], as shown in as shown in Fig. 2. The streaming processor array (SPA) is located at the higher level of the GPU grouping a set of TPC, as shown in Fig. 3.

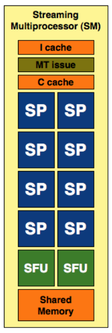

Figure 1: Streaming Processor

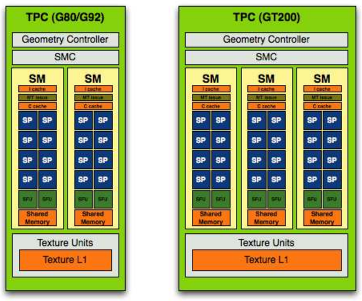

Figure 2: Texture/Processors Clusters from a G80 GPU and GT200

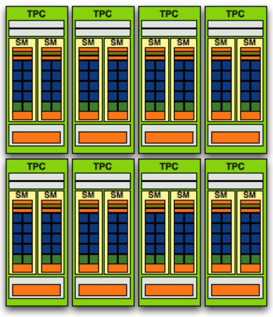

Figure 3: A Streaming Processor Array from a G80 architecture

Due to the fact that the world of real-time and high definition 3D programming has traditionally required advanced computational power, the Graphic Processor Unit acquired highly parallel, multi-threaded, manycore processor capabilities. Thanks to the graphic rendering nature and throughput time, GPU has specialized in compute-intensive and highly parallel computation rather than CPU, which has evolved to control flow and caching data [18]. Chip area devoted to processing as compared to control and data cache/s in each processor show this trend.

5 CUDA and Using CUDA in a Fortran Program

The first implementation of the *initially* defined as Compute Unified Device Architecture (CUDA) [8] was introduced by NVIDIA in 2006. CUDA was developed based on a new parallel programming model and on a new instruction set architecture [18]. It has represented a completely new concept in GPU programming. In the very early stages of GPU programming, programmers should develop the programs in assembler, requiring a very specific knowledge of the hardware they used [3]. Since 2006, with the advent of CUDA, GPU programming has become less and less complex. Since 2006, CUDA programming can be implemented, for example, using the C programming language. In 2009, the first version of CUDA Fortran was released by the Portland Group (PGI) Fortran compiler [12]. One of the most relevant aspects about CUDA is the parallel granularity found in the GPU computing model, basically organized into three levels [17]:

- Thread Level: a single sequence of (instructions') execution. A single SP is used for thread execution. GPUs are not meant for this kind of processing but, instead, used the thread as the underlying *building block* for parallel execution.
- Block Level: a set of threads is grouped in blocks, i.e. executed concurrently. Threads in a block are able to use a shared memory (intra SM as shown in Fig. 1) and can be synchronized. Blocks are identified with a block ID.
- Grid Level: an array of blocks is identified as a grid which executes the same code (or kernel). Blocks in a grid share global memory, which is shared by every TPC (such as those shown in Fig. 2 and Fig. 3)

Different organizations or definitions of code to be executed in threads (i.e. kernel code), number and organization of threads in a block and number and organization of blocks in a grid will provide a wide range of performance results. As a matter of fact, defining the processing architecture independent of the graphic pipeline as explained in the previous section has been (maybe the first) part of the decision in order to provide CUDA-enabled GPUs. On the software side, the general purpose programming model and API (Application Programming Interface) to provide access to processing capabilities (along with the *corresponding* runtime) is also defined by CUDA specifically designed for programmers and general purpose parallel processing.

NVIDIA and The Portland Group (PGI) have developed a CUDA Fortran Compiler [19] which, in fact, includes three levels of CUDA programming:

- CUDA Fortran as implemented by the PGI compiler, which is almost identical to that of CUDA C, but taking advantage of the Fortran type (and, in general, entity) attribute [30] which is defined by the Fortran standard since Fortran 90.
- CUDA Kernel Loop Directives, also defined by the PGI Fortran compiler to be applied directly on Fortran `do` loops, so the compiler automatically generates CUDA kernels. Kernel Loop Directives are very similar to OpenMP worksharing loop construct [22].
- OpenACC directives [21], which are not directly part of CUDA programming but provides access to CUDA-enabled GPUs via a complete set of directives similar to those defined by the OpenMP specification.

The following subsections briefly describe each type of access to CUDA-enabled GPUs and related specifications/definitions.

5.1 CUDA Fortran

To port a legacy program to CUDA processing much of (the compute-intensive) source code has to be modified. Usually, the necessary changes that need to be made to an original program to be run in a GPU can be summarized as:

- GPU memory allocation: as a first step those data structures that will be used to compute in the GPU must be allocated from the host, e.g.:

```
...
integer, allocatable, device :: a_d(:,:), b_d(:,:), c_d(:,:)
...
allocate(a_d(row,col))
allocate(b_d(row,col))
allocate(c_d(row,col))
...
```

the first line indicates that these data structures will be allocated in the GPU by using the device keyword.

- Host $\leftrightarrow$ GPU data transfers are handled directly by assignments. The compiler *figures out* actual transfer (either from host memory to GPU memory or from GPU memory to host memory) according data declarations with or without the device keyword.

```
...
integer, allocatable, device :: a_d(:,:), c_d(:,:)
integer, allocatable,        :: c(:,:)
...
allocate(a_d(row,col))
allocate(c_d(row,col))
allocate(c(row,col))
...
a_d = a
...
c   = c_d
...
```

where, `a_d = a` implies a host memory to GPU memory data transfer and `c = c_d` implies a GPU memory to host memory data transfer.

- The CUDA Kernel: the most important CUDA structure, the CUDA Kernel, must be programmed as a subroutine by using CUDA internal structures/data such as blockIdx, blockdim, threadIdx, and so forth. One CUDA Kernel example is listed bellow:

```
attributes(global) subroutine matrix_mul(a, b, c, row, col)
  implicit none
  double precision, intent(in)  :: a(row,col), b(row,col)
  double precision, intent(out) :: c(row,col)
  integer, value                :: row, col

  integer, device               :: cij
  integer, device               :: k, ix, iy

  ix = (blockIdx%x-1) * blockdim%x + threadIdx%x
  iy = (blockIdx%y-1) * blockdim%y + threadIdx%y

  cij = 0.0d0

  do k = 1, row
    cij = cij + a(k,ix)*b(k,iy)
  end do
  c(ix,iy) = cij
end subroutine matrix_mul
```

It is worth noting that a kernel function directly defines what a single thread will compute in a single SP (of those shown in Fig. 1). Also, note the use of the attribute `value` for arguments `row` and `col` so that each thread has its own copy of the values for those arguments.

- GPU Block Grid Size and organization: The GPU parallel model requires the specification of the number and organization of threads grouped in a block that can cooperate by using shared memory (intra SM, see Fig. 1) or by synchronizing their execution. The number of threads in a block is referred to as the Block Size, and the threads can be organized in 1, 2, or 3 dimensions [24]. The number and organization of blocks has to be given each time a kernel is launched, and the number of blocks of threads is referred to as Grid Size. Blocks can be arranged in 1, or two dimensions [24]. At runtime, each CUDA Kernel launches a grid of thread blocks, i.e. the grid of thread blocks is launched by invoking the corresponding kernel function with the values

  <<<Grid Specification, Block Specification>>>

  such as in

  ```
  ...
  GridSpec  = dim3(row/10,col/10,1)
  BlockSpec = dim3(10,10,1)
  ...
  call matrix_mul<<<GridSpec,BlockSpec>>>(a_d,b_d,c_d,row,col)
  ...
  ```

One of the most clear disadvantages of Fortran CUDA Programming from the legacy source code point of view lays in the number of changes that need to be introduced. Most (if not all) of these changes can be considered critical, since they have to be made to the compute intensive fraction of code which is executed most of the runtime.

5.2 CUDA Kernel Loop Directives

PGI CUDA Fortran supports automatic kernel generation and invocation by using directives associated to one or more host tightly nested loops [30] [24]. These directives must be used before the loop/loops so that the code will be processed (in parallel) in the GPU. In the simplest form, the programmer just indicates that the following `do`(s) (the ourtermost, by default, or the outermost `n` loops if `n` is given in the directive) should be a kernel call, and the compiler generates the kernel, and kernel call including grid and block definitions. The PGI Fortran CUDA Kernel Loop Directives (KLD) syntax is given as

```
   !$cuf kernel do[(n)] <<< grid, block >>>
```

which could be used directly as

```
   !$cuf kernel do <<<*, *>>>
   do ...
```

so that the compiler will define the kernel, grid, and block details/values of the `do` loop immediately following the directive. Grid and block specifications can be given as integer scalar values or as parenthesized list/s. KLD avoid explicit definition of kernel subroutines in the source code. Thus, KLD imply less legacy source changes/adaptations than the previous method (using *plain* CUDA Fortran), but loops on which this directive can be applied have a number of restrictions [24] such as

- If the directive specifies `n`, it must be followed by at least that many tightly-nested loops.
- The (tightly-nested) loops in the scope of the directive must have invariant loop limits: the lower limit, upper limit, and increment must be invariant with respect to any other loop in the kernel `do`.
- There can be no GOTO or EXIT statements within or between any loops that have been mapped onto the grid and block configuration values.
- The body of the loops may contain assignment statements, IF statements, loops, and GOTO statements.
- Only CUDA Fortran dataypes are allowed within the loops.
- CUDA Fortran intrinsic functions are allowed, if they are allowed in device code, but the device-specific intrinsics such as syncthreads, atomic functions, etc. are not.
- Subroutine and function calls to attributes(device) subprograms are allowed if they are in the same module as the code containing the directive.
- Arrays used or assigned in the loop must have the device attribute.

- Implicit loops and F90 array syntax are not allowed within the directive loops.
- Scalars used or assigned in the loop must either have the device attribute, or the compiler will make a device copy of that variable live for the duration of the loops, one for each thread. Except in the case of reductions; when a reduction has a scalar target, the compiler generates a correct sequence of synchronized operations to produce one copy either in device global memory or on the host.

Even when the source code will not include kernel subroutines like that in the previous subsection, specified with `attributes(global)`, a source legacy code should clearly *adapted* in order to fulfil the (rather large/strong) number of requirements of KLD. Summarizing, using KLD imply:

- Changes on legacy source code are made in loops, which are automatically compiled as kernels by the PGI Fortran compiler.
- Changes on legacy source code are, in some way, identified/defined by the restrictions under which KLD can be used.
- Grid and block specifications can be handled by the compiler/runtime or can be explicitly defined by the programmer.
- Loop kernel code is relatively constrained to that required by the KLD, i.e. not all the CUDA routines/facilities are allowed to be used in KLD-related code.

Thus, KLD can be considered a step further in making simpler and less aggressive legacy source code changes/adaptations at the cost of constraining the code to be executed in CUDA kernels (generated by the PGI compiler).

5.3 OpenACC Directives

OpenACC [21] is a standard developed by Cray, CAPS, NVIDIA, and PGI [16]. This programming standard was created for parallel computing development to make the parallel programming on heterogeneous CPU-GPU systems. OpenACC focuses in the concept of "directives" or code annotations just like KLD, but OpenACC directives are intended to be used directly on existing (including legacy) source code. While KLD restrictions imply, for example, to define device data for usage in `!$cuf ...` directives, OpenACC directives are hints provided by the programmer to the compiler, in which regions of source code are identified to be parallelized, without requiring the programmer to modify or adapt the underlying code itself. The use of the directives allows the compiler to perform the tasks to map the computations that will be executed in the host and those that will be performed by the GPU. These code annotations in the case of OpenACC are very similar to those adopted by OpenMP. The compiler is notified via the annotations or directives that a region of source code should be accelerated/parallelized by using a GPU. Once the compiler finds a directive, it is in charge of mapping data structures to the device, generating the kernel section, specifying thread block the block grid, transferring data between host and device/s, and finally generating the compiled version of the code that will run partially in the host and partially in the device. The programmer only is requierd to write the directive into the legacy source code.

It seems to be the ideal programming model to be used on legacy programs. In Fortran, OpenACC directives are included as comments. This offers two advantages, (i) the program can be compiled with a compiler that does not conform to OpenACC and it will still work, and (ii) changes introduced in the source code will not impact directly on the current program functionality granting that no new bugs will be added to the software. There are several types of OpenACC directives, referred to as "PGI Accelerator Directives" in [25], such as

- Accelerator Compute Region Directive, which is syntactically defined as

  ```
  !$acc region [clause [, clause]]
    structured block
  !$acc end region
  ```

  which could be directly related to kernel code as defined above: code to be executed in a GPU device.
- Accelerator Data Region Directive,which is syntactically defined as

```
!$acc data region [clause [, clause]]
  structured block
!$acc end data region
```

which is used for handling data in the device memory. Here, *handling* refers to device memory allocation as well as data movement between device and host memory.

The relationship/similarity with OpenMP directives is clear starting at the syntax. In general, every type of accelerator directive is related to compute and/or memory handling in a device (GPU).

6 Evaluation on a Matrix Multiplication Example

In order to study how CUDA programming and PGI KLD and accelerator directives impact into Fortran Legacy Code, an initial and very simple program has been selected. The selected example is a simple matrix multiplication program, as that shown in Fig. 4, where we have *hardcoded* the matrix size, `n`, and

```
program matmul
  use modmm
  implicit none

  integer                         :: n
  double precision, allocatable  :: a(:,:)
  double precision, allocatable  :: b(:,:)
  double precision, allocatable  :: c(:,:)

  n = 4000

  allocate(a(n,n))
  allocate(b(n,n))
  allocate(c(n,n))

  ! initialize matrices
  call initmat(a, n, 1d0)
  call initmat(b, n, 1d0)
  call initmat(c, n, 0d0)

  ! c = a x b
  call mm(a, b, c, n)

  ! check result
  call chkval(c, n, dble(n))

  deallocate(a)
  deallocate(b)
  deallocate(c)
end program  matmul
```

Figure 4: Example Matrix Multiplication Main Program

we have used a Fortran module in order to reduce the number of lines of code necessary for the example program. Module `modd` includes subroutines `initmat`, `mm`, and `chkval` as shown in Fig. 5, where:

- `initmat` and `chkval` are not expected to be very useful routines, since they are specifically implemented for the tests to be carried out in this report.
- Clearly, there are better matrix multiplication algorithms and optimizations to be applied, but legacy code does not necessarily includes such enhancements.
- Specifically, the routine `mm` computes `c = axb + c` where `a`, `b`, and `c` are `nxn` element matrices.

```
module modmm
contains
  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  subroutine initmat(m, n, val)
    implicit none
    double precision :: m(n,n), val
    integer          :: n, i, j

    do i = 1, n
      do j = 1, n
        m(i,j) = val
      end do
    end do
  end subroutine initmat
  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  subroutine mm(a, b, c, n)
    implicit none
    double precision :: a(n,n), b(n,n), c(n,n)
    integer          :: n, i, j, k

    ! making the product
    do i = 1, n
      do j = 1, n
        do k = 1, n
          c(i,j) = c(i,j) + a(i,k)*b(k,j)
        end do
      end do
    end do
  end subroutine mm
  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  subroutine chkval(m, n, val)
    implicit none
    double precision :: m(n,n), val
    integer          :: n, i, j

    do i = 1, n
      do j = 1, n
        if (m(i,j) /= val) print *, "diff. at", i, j
      end do
    end do
  end subroutine chkval
end module modmm
```

Figure 5: Module Including a Simple Matrix Multiplication Routine

We will use the code of Fig. 4 and Fig. 5 as a departure point in order to evaluate:

- The number/scope of necessary changes necessary in source code to take advantage of PGI CUDA Fortran, KLD, and accelerator directives.
- The performance gains obtained in a specific platform (PC + NVIVIA GPU).

The computer used for the experiments is shown in Table 1, which can be considered a standard Linux installation in current PC hardware, adding the PGI Fortran compiler. The PC has two Tesla 2075 GPU

Processor	i5-2310 CPU @ 2.90GHz
RAM	16 GB
OS	Linux 3.2 64 bits
Compiler	pgfortran 12.5-0 64-bit

Table 1: GPU Host Computer Characteristics

cards, with the main characteristics shown in Table 2 as obtained via the command `pgaccelinfo` (also provided by the PGI compiler installation)

Device Name	Tesla C2075
Device Revision Number	2.0
Global Memory Size	5636554752
Number of Multiprocessors	14
Number of Cores	448
Total Constant Memory	65536
Total Shared Memory per Block	49152
Registers per Block	32768
Warp Size	32
Maximum Threads per Block	1024
Maximum Block Dimensions	1024, 1024, 64
Maximum Grid Dimensions	65535 x 65535 x 65535
Clock Rate	1147 MHz
...	
Initialization time	5137572 microsec

Table 2: NVIDIA Tesla C2075 Characteristics

The matrix multiplication program of Fig. 4, using the specific code for matrix multiplication given in Fig. 5 was compiled with `pgfortran -fast`. Fig. 6 shows the runtime as measured and reported by the `time` command. Having defined the most important details of CUDA Fortran, Kernel Loop Directives,

```
real    0m37.769s
user    0m37.490s
sys     0m0.144s
```

Figure 6: Runtime of the First Matrix Multiplication Program

and OpenACC Directives, we will use those facilities in order to analyze necessary changes as well as performance obtained with each one of them.

6.1 CUDA Fortran

Clearly, only the routine `mm` (Fig. 5) should be affected by including CUDA Fortran processing in the source code. We can describe changes in terms of data and processing. New (device) data is necessary in order to hold the initial values of matrices `a`, `b`, and `c`, as well as to hold the new values of `c`, computed in the GPU. The new matrices, their allocation and initialization can be almost directly written in the original source code of subroutine `mm` as

```
! Data to be allocated in the GPU
 double precision, allocatable, device  :: a_d(:,:)
 double precision, allocatable, device  :: b_d(:,:)
 double precision, allocatable, device  :: c_d(:,:)

 ...

 ! Allocate and initialize matrix data in GPU
 allocate(a_d(n,n))
 allocate(b_d(n,n))
 allocate(c_d(n,n))
 a_d = a
 b_d = b
 c_d = c
```

Given that there is no more to be computed in the host, the only remaining task is launching a grid of blocks (and threads) so that the result is computed in the GPU. Assuming there is a kernel routine `cudamm` implemented, the following code implies:

1. Create a grid of `n/10` x `n/10` thread blocks. This kind of expressions for defining grids is relatively usual, since *directly implies* that blocks are defined as being of `10` x `/10` threads.

2. Threads in a block are arranged in a bidimensional array of size `10 x /10`, as previously explained.
3. The result is copied in host memory (the last assignment).

```
! Set grid and block specification
gridspec  = dim3(n/10, n/10, 1)
blockspec = dim3(10, 10, 1)

! Kernel call
call cudamm<<<gridspec, blockspec>>>(a_d, b_d, c_d, n)

! Get the result from GPU memory
c = c_d
```

Thus, the complete `mm` subroutine can be given as in Fig. 7. Note the line with `use cudafor` which is

```
use cudafor
...
! This subroutine does not compute, just calls the kernel
! and handles memory transfers
subroutine mm(a, b, c, n)
  implicit none
  double precision :: a(n,n), b(n,n), c(n,n)
  integer          :: n

  ! Data to be allocated in the GPU
  double precision, allocatable, device  :: a_d(:,:)
  double precision, allocatable, device  :: b_d(:,:)
  double precision, allocatable, device  :: c_d(:,:)

  ! Grid and Block definitions
  Type(dim3)                             :: gridspec, blockspec

  ! Allocate and initialize matrix data in GPU
  allocate(a_d(n,n))
  allocate(b_d(n,n))
  allocate(c_d(n,n))
  a_d = a
  b_d = b
  c_d = c

  ! Set grid and block specification
  gridspec  = dim3(n/10, n/10, 1)
  blockspec = dim3(10, 10, 1)

  ! Kernel call
  call cudamm<<<gridspec, blockspec>>>(a_d, b_d, c_d, n)

  ! Get the result from GPU memory
  c = c_d
end subroutine mm
```

Figure 7: Subroutine Including a Call to a CUDA Kernel Matrix Multiplication Routine

specifically needed in this example for the definition of `dim3` which, in turn, is necessary in the `gridspec` and `blockspec` declarations. In the kernel routine, `cudamm`, the two outermost loops of the *original* `mm` subroutine of Fig. 5 are *replaced* by references to thread location in the grid of threads, i.e. each thread will compute a single element of matrix `c`, and threads take into account its location in the grid and block. The kernel routine is given in Fig. 8.

```
attributes(global) subroutine cudamm(a, b, c, n)
 implicit none
 double precision, device :: a(n,n), b(n,n), c(n,n)
 integer, value           :: n
 integer                  :: k, ix, iy

 ix = (blockIdx%x-1) * blockdim%x + threadIdx%x
 iy = (blockIdx%y-1) * blockdim%y + threadIdx%y
 do k = 1, n
   c(ix, iy) = c(ix, iy) + a(k,ix)*b(k,iy)
 end do
end subroutine cudamm
```

Figure 8: CUDA Fortran Kernel Matrix Multiplication Routine

At this point, we are able to build a *complete* CUDA Fortran matrix multiplication program including a CUDA kernel matrix multiplication routine, as shown in Fig. 9, where:

```
!==============================================================
module modmm
  use cudafor
contains
  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  subroutine initmat(m, n, val)
    ...
  end subroutine initmat

  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  ! This subroutine does not compute, just calls the kernel
  ! and handles memory transfers
  subroutine mm(a, b, c, n)
    ...
  end subroutine mm

  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  attributes(global) subroutine cudamm(a, b, c, n)
    ...
  end subroutine cudamm

  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  subroutine chkval(m, n, val)
  ...
  end subroutine chkval
end module modmmcuda

!==============================================================
program matmul
  ...
end program matmul
```

Figure 9: Complete Program Including a CUDA Kernel Matrix Multiplication Routine

- Subroutines `initmat` and `chkval` are those given in Fig. 5.
- Subroutines `mm` and `cudamm` are those given above in this section.
- The main program remains unchanged (from that of Fig. 4) since changes are completely contained in `modmm`.

The complete source code is given in Appendix A. It could be argued that it is unlikely that functions to be changed/transformed to use CUDA are already contained in modules, but we think there are some previous tasks that need to be performed before dealing with several optimization and parallelization work [27]. Several tasks, such as including functions in modules is not easy, since those functions could use Fortran `common` (global) data, but such work will provide better and more manageable software. And this is the kind of work we are engaging in with legacy source code.

The program has to be compiled (at least) with flag `-Mcuda` so that the compiler recognizes CUDA-related source code [28]. Actually, we used `pgfortran -fast -Mcuda` so that host code is also optimized. The `time` command reported

```
real    0m23.163s
user    0m14.505s
sys     0m4.192s
```

on runtime/s, which could be compared with those in Fig. 6, and

- Processing in the GPU reduces the runtime: `0m37.769s` vs. `0m23.163s`.
- When using CUDA, the `real` time is no longer approximately equal to `user + sys`.
- `sys` time seems to be too high, even in the case the CPU is strongly involved in memory transfers to/from the GPU.

Looking for the possible causes of the high `sys` time when using CUDA, we have taken into account

- The last line in Table 2, which is reported by `pgacclinfo` which is near the `sys` time shown above
- The advice given in [26], section "Running an Accelerator Program", about using `pgcudainit` in order to avoid the GPU startup time.

Fig. 10 shows the runtime as measured and reported by the `time` command and using the command `pgcudainit` running in the background. Reported `sys` time in Fig. 10 is greater than that reported

```
real    0m18.135s
user    0m17.849s
sys     0m0.212s
```

Figure 10: Runtime of the CUDA Matrix Multiplication Program

in Fig. 6, but it is in some way expected, because of the data transfers to/from the GPU. Clearly, the runtime has been reduced by using Fortran CUDA, from `0m37.769s` as shown in Fig. 6 to 0m18.135s as shown in Fig. 10, but there is room to a lot of enhancements, such as testing/looking for the best values of block dimensions or using other ways of computing in the kernel routines, as that shown in [28]. Actually, we could look for further enhancements given that we have two Tesla 2075 cards attached to the computer, which could be used in parallel.

6.2 CUDA Kernel Loop Directives

As explained before, Kernel Loop Directives (KLD) are applied directly on loops (as its name suggests). Clearly, matrix multiplication has three nested loops and, thus, there is no problem to identify KLD "target" code. Furthermore, every Fortran application will have its most time-consuming code in loops. However, not every loop or group of nested loops will be as simple as that of the matrix multiplication algorithm as given in Fig. 5.

As explained before, applying KLD to a legacy source code is not immediate, and we could define changes in terms of computation and data

- Computation: the most time consuming loop/s should be identified as those on which `!$ cuf` directive should be used.
- Data: Every data involved in the loops under the `!$ cuf` directive/s should be analyzed in order to define `device` data.

In the specific case of the code given in Fig. 4 and Fig. 5 (the initial simple program taken as legacy code), the most time consumig code is contained in the loops:

```
do i = 1, n
  do j = 1, n
    do k = 1, n
      c(i,j) = c(i,j) + a(i,k)*b(k,j)
    end do
  end do
end do
```

which involves computation on three two-dimensional arrays: `a`, `b`, and `c`. Thus, we should a) declare device data for those arrays, b) allocate data in GPU, and c) initialize GPU data with the corresponding matrices, i.e.

```
...
! Data to be allocated in the GPU
double precision, allocatable, device  :: a_d(:,:)
double precision, allocatable, device  :: b_d(:,:)
double precision, allocatable, device  :: c_d(:,:)
...
! Allocate and initialize matrix data in GPU
allocate(a_d(n,n))
allocate(b_d(n,n))
allocate(c_d(n,n))
a_d = a
b_d = b
c_d = c
```

Now, using a KLD is almost straigthforward

```
!$cuf kernel do(2) <<< *,* >>>
do i = 1, n
  do j = 1, n
    do k = 1, n
      c_d(i,j) = c_d(i,j) + a_d(i,k)*b_d(k,j)
    end do
  end do
end do
```

Or, for a grid of 10x10 threads blocks, we should use

```
!$cuf kernel do(2) <<< (n/10, n/10), (10, 10) >>>
...
```

And all these source code transformations should be made to the *original* `mm` subroutine shown in Fig. 5. The *new* `mm` subroutine using KLD is shown in Fig. 11, where each matrix to be allocated in the GPU is directly allocated by the lines

```
double precision, device :: a_d(n,n)
double precision, device :: b_d(n,n)
double precision, device :: c_d(n,n)
```

instead of using the sequence

```
...
! Data to be allocated in the GPU
double precision, allocatable, device  :: a_d(:,:)
double precision, allocatable, device  :: b_d(:,:)
double precision, allocatable, device  :: c_d(:,:)
...
allocate(a_d(n,n))
allocate(b_d(n,n))
allocate(c_d(n,n))
```

and this reduces the number of source code lines while not affecting the runtime (the runtime for the `mm` routine with allocatable matrices is the same as that of the routine given in Fig. 11). Another (minor) source code simplification is direct usage of parenthesized notation for grid and block definition instead of using `Type(dim3)` data, which also implies that module `cudafor` is no longer needed in the `modmm` module. Note that the code in Fig. 11 not only avoids the usage of a CUDA kernel routine but, also, the source code is really similar the original one in Fig. 4, with the exception of computing on `device` data instead of computing on the corresponding subroutine's dummy arguments. However, `device` data declaration, GPU data initialization and and getting the result computed in the GPU via an assignment are necessary source code modifications to be made to legacy code in order to take advantage of GPU processing. The complete matrix multiplication program using KLD in subroutine mm is given in Appendix B, and can be compiled with `pgfortran -fast -Mcuda` as the previous version (the one with an explicit CUDA kernel routine).

```
! This subroutine has been changed to use a cuf directive
subroutine mm(a, b, c, n)
  implicit none
  double precision :: a(n,n), b(n,n), c(n,n)
  integer          :: n

  ! Data to be allocated in the GPU
  double precision, device :: a_d(n,n)
  double precision, device :: b_d(n,n)
  double precision, device :: c_d(n,n)

  ! matrix indexes
  integer                  :: i, j, k

  ! Initialize matrix data in GPU
  a_d = a
  b_d = b
  c_d = c

  !$cuf kernel do(2) <<< (n/10,n/10), (10,10) >>>
  do i = 1, n
    do j = 1, n
      do k = 1, n
        c_d(i,j) = c_d(i,j) + a_d(i,k)*b_d(k,j)
      end do
    end do
  end do

  ! Get the result from GPU memory
  c = c_d
end subroutine mm
```

Figure 11: Fortran Matrix Multiplication Routine Using KLD

Fig. 12 shows the runtime as measured and reported by the `time` command and using the command `pgcudainit` running in the background. Even when source code of Fig. 11 and that in Fig. 9 are not completely different, runtime of code Fig. 11 as given in Fig. 12 is less than a half the runtime of code in Fig. 9, as given in Fig. 10 (matrix size, grid and blocks are the same). A possible cause for the

```
real    0m8.291s
user    0m8.041s
sys     0m0.208s
```

Figure 12: Runtime of the Matrix Multiplication Program Using KLD

performance improvement of KLD code (Fig. 11) as compared to that of using the kernel subroutine (Fig. 9) could be the way in which data accesses are made. In the case of KLD code, matrix accesses are handled by the compiler from the source code which directly uses the loop indexes, while in the kernel subroutine each access is computed from thread ids (thread *location* in the grid and block). However, there is not *full* evidence that accessing matrices via data local to a subroutine could impose more than 100% performance penalty.

6.3 OpenACC Directives

The matrix multiplication algorithm is particularly well suited for OpenACC directives since, from the point of view of computing, the `!$acc region` should be used on the three tightly nested loops, just as explained in the previous subsection for KLD. Thus, the code adding the corresponding OpenACC *accelerator computing directive* would be

```
!$acc region
do i = 1, n
  do j = 1, n
    do k = 1, n
      c(i,j) = c(i,j) + a(i,k)*b(k,j)
    end do
  end do
end do
!$acc end region
```

And there is no need for additional accelerator directives such as those referred to as *accelerator data directives*, since

- variables `i`, `j`, and `k` are automatically handled as local to each thread by the compiler, such as when using a `!$cuf` directive.
- matrix storage (for matrices `a`, `b`, and `c`) is automatically allocated in the device.
- matrix data transfers are automatically handled by the compiler: initialization (transferring data from host to device memory) previous to the computation in the GPU, and obtaining the result (transferring data from device to host memory) after computation in the GPU.

It is very interesting compiler's definitions for grid and blocks and the corresponding ways of changing compiler's default/s. The first step, however, is that the compiler to handle the OpenACC directives (referred to as accelerator directives in [25]). The `pgfortran` option `-Mcuda` does not include handling OpenACC directives, so we should use at least the option `-ta=nvidia`. Actually, we use `pgfortran -ta=nvidia,time -Minfo=accel` since

- Option `-ta=nvidia,time` (including *nvidia_suboption* "`time`") implies:
  - Accelerator regions are compiled to the NVIDIA accelerator/GPU.
  - Accelerator regions are profiled (via a so-called "limited-profiling library").
- Option `-Minfo=accel` provides information about the way in which accelerator regions are compiled.

The complete code is given in Appendix C just for the sake of completeness, but it is the same as that in Fig. 4 and Fig. 5 with the three loops inside `!$acc region ... !$acc end region` as shown above. Compiling the code with the aforementioned options (`pgfortran -ta=nvidia,time -Minfo=accel`), provides the compiler information shown in Fig. 13, where

```
mm:
     29, Generating copyin(b(:n,:n))
         Generating copyin(a(:n,:n))
         Generating copy(c(:n,:n))
         Generating compute capability 1.3 binary
         Generating compute capability 2.0 binary
     30, Loop is parallelizable
     31, Loop is parallelizable
     32, Complex loop carried dependence of 'c' prevents parallelization
         Loop carried dependence of 'c' prevents parallelization
         Loop carried backward dependence of 'c' prevents vectorization
         Inner sequential loop scheduled on accelerator
         Accelerator kernel generated
         30, !$acc do parallel, vector(16) ! blockidx%x threadidx%x
         31, !$acc do parallel, vector(16) ! blockidx%y threadidx%y
         32, !$acc do seq(16)
             Cached references to size [(x)x16] block of 'a'
             Cached references to size [16x(y)] block of 'b'
             CC 1.3 : 27 registers; 80 shared, 20 constant, 0 local memory bytes
             CC 2.0 : 22 registers; 16 shared, 76 constant, 0 local memory bytes
```

Figure 13: Compiler Report on OpenACC Region

- Information is related to subroutine `mm`, which is the only one with OpenACC directives.
- Data copy is reported for matrices `a`, `b`, and `c`.
- The two outer loops are identified as parallelizable and the inner loop is identified as having a data dependence, hence it is not parallelized.
- The grid is constructed with blocks of 16x16 threads, using two `!$acc do parallel, vector(16)` OpenACC directives.

In order to make a *fair* comparison with previous timing measurements, we have explicitly given the block of threads via the corresponding `!$acc do ...` directives, i.e.

```
!$acc region
!$acc do parallel, vector(10)
do i = 1, n
  !$acc do parallel, vector(10)
  do j = 1, n
    do k = 1, n
      c(i,j) = c(i,j) + a(i,k)*b(k,j)
    end do
  end do
end do
!$acc end region
```

Now, besides the output of the `time` command, the program generated by the compiler shows its own profiling data on accelerator regions, as specifically instructed by the *nvidia_suboption* "`time`" of the `-ta=nvidia` option (recall that `-ta=nvidia,time` was used). The complete output of the program is shown in Fig. 14 (the command `pgcudainit` is running in the background), where

```
...
  mm
    30: region entered 1 time
        time(us): total=3156709 init=55486 region=3101223
                  kernels=2939557 data=155903
        w/o init: total=3101223 max=3101223 min=3101223 avg=3101223
        35: kernel launched 1 times
            grid: [400x400]  block: [10x10]
            time(us): total=2939557 max=2939557 min=2939557 avg=2939557

real    0m3.517s
user    0m2.120s
sys     0m1.368s
```

Figure 14: Runtime of the Matrix Multiplication Program Using OpenACC, 10x10 Threads in a Block

- The last three lines are generated by the `time` command as in the previous examples, and the previous lines are generated by the program, using its own profiling data.
- It is reported that the kernel is launched 1 time, as expected.
- We now have available some interesting and potentially very useful timing data (given in microseconds), specifically about
  - kernel runtime, given as 2939557 (about 2.94s)
  - initialization, given as 55486 (about 0.55s)
  - data transfer/s runtime, given as 155903 (about 0.16s)
- The total runtime as measured by the operating system is 3.517s, and specifically the GPU (accelerator region) runtime is about 3.1s.

Is this runtime the optimum one? A priori, it is better than that of using KLD directives, given in Fig. 12. However, threads are scheduled in the GPU in groups of 16 threads (*so-called* warps) inside Streaming Processors (SPs) where all of the threads are able to share a common memory and synchronize. Thus, the compiler's *default* for 16x16 threads blocks (shown in Fig. 13) seems to be a good choice for launching a kernel in a GPU. Fig. 15 shows the runtime as measured and reported by the `time` command and using the command `pgcudainit` running in the background.

```
real    0m2.117s
user    0m1.244s
sys     0m0.848s
```

Figure 15: Runtime of the Matrix Multiplication Program Using OpenACC, 16x16 Threads in a Block

OpenACC turns to be the best option for this simple problem, because: a) it involves just a few source changes and b) it provides the best performance. Thus, OpenACC directives seem to be the first choice when a legacy Fortran program has to be adapted/transformed in order to take advantage of GPU capabilities. Also, OpenACC directives seem to be the best supported approach by the PGI Fortran compiler, since:

- The compiler is able to report static information about accelerator (GPU) code being generated, as explained above for the usage of the `-Minfo=accel`. It is clear that kernel code is explicitly given in Fortran CUDA and using KLD, but some of the information given under `-Minfo=accel` and using OpenACC directives is hard to obtain, e.g. information about binary compute capability as shown in Fig. 13.
- We consider the *automatic* profiling code and report produced by `ta=nvidia,time` highly useful and at not cost, and we would like to have a similar option for Fortran CUDA and KLD. Even when the code can be profiled via instrumentation (e.g. using `acc_enable_time` and related subroutines/functions) the source code has to be changed. The compiler option available for accelerator directives reduces the necessary source code changes and provides useful runtime-collected data.

7 Conclusions and Further Work

In this article we have studied some different ways of taking advantage of GPU devices in legacy Fortran code, such as Fortran CUDA programming, Fortran CUDA Programming using Kernel Loop Directives, and OpenACC programming. A source code example has been selected in order to apply these different techniques and to measure the benefits of their usage. All of these techniques have (highly) improved processing performance. Table 3 show the performance obtained in the host computer ("Legacy" column) and for each of the alternatives in a single Tesla GPU ("CUDA", "CUDA KLD", "OpenACC 10x10", and "OpenACC 16x16" columns). We are not able to extrapolate these results neither in terms of Fortran legacy source code transformations nor expected performance gains. The matrix multiplication algorithm is extremely simple, contained in a single subroutine, and has a well known data-access and data-processing patterns. Most of the Fortran legacy source code usually is, at least, hard to read and cannot be approached in a single and straight way. Instead, we propose an incremental process of transformations as explained in [27]. Source code transformations for GPU processing is possible and can be included as part of the transformation incremental process. From the point of view of performance, we have obtained great improvements for the matrix multiplication program, but we are still far from the peak double precision floating point performance for the Tesla 2075, as given in [20]: 515 Gflop/s. Also, the performance obtained in the host computer as well as in the GPU is highly biased by the non-optimized source code. However, we think it is possible to include GPU processing in Fortran legacy optimized and unoptimized source code. The amount of work will have to be evaluated in a case-by-case analysis, as usual. We also should take into account that specifically optimized host algorithms are not necessarily well suited directly for GPU processing [29].

	Legacy	CUDA	CUDA KLD	OpenACC 10x10	OpenACC 16x16
(Elapsed) Runtime	37.769s	18.135s	8.291s	3.517s	2.117s
Gflop/s	3,4	7,1	15,4	36,4	60,5

Table 3: Host and GPU Performance for Matrix Multiplication

We will continue our work following several guidelines, taking into account the work in this report as well as previous work on Fortran legacy source code transformations and GPU processing:

- Approach real Fortran legacy source code, such as climate models and Magnetogasdynamics (MGD) flows model/s.
- Measurements of source code transformations, specifically related to performance improvements obtained in GPUs.
- Inclusion of specific tools in a development tool-chain or IDE (integrated Development Environment) in order to take advantage of OpenACC directives in legacy Fortran source code.

We also think that much of the work for using more than one GPU card in parallel is already done, so we will explore this line of research in the short-term research work.

Acknowledgements

We would like to thank NVIDIA Corporation, which donated several NVIDIA cards (including the two NVIDIA Tesla 2075 mentioned in this report) to IEEE/NSF Fall-11 Early Adopter awardees. We would also like to thank The Portland Group (PGI) for providing an extended evaluation copy of PGI Accelerator Workstation for this research.

References

[1] J. Backus. The History of Fortran I, II, and III. *ACM SIGPLAN Notices*, 13(8):165–180, 1978.

[2] S. Behling, R. Bell, P. Farrell, H. Holthoff, F. OConnell, and W. Weir. *The POWER4 Processor Introduction and Tuning Guide.* International Business Machines Corporation, Nov. 2001.

[3] R. G. Belleman, J. Bédorf, and S. F. Portegies Zwart. High performance direct gravitational n-body simulations on graphics processing units ii: An implementation in cuda. *New Astronomy*, 13(2):103–112, 2008.

[4] K. Bennett. Legacy systems: Coping with success. *IEEE Software*, 12(1):19–23, 1995.

[5] S. Borkar. Thousand core chips: a technology perspective. In *Proceedings of the 44th annual Design Automation Conference*, pages 746–749. ACM, 2007.

[6] S. Borkar. Thousand core chips: a technology perspective. In *Proceedings of the 44th annual Design Automation Conference*, pages 746–749. ACM, 2007.

[7] M. L. Brodie and M. Stonebraker. *Migrating legacy systems.* Morgan Kaufmann Publishers, 1995.

[8] R. Farber. Cuda, supercomputing for the masses: Part 1. *Doctor Dobb's Journal*, April 15, 2008.

[9] D. Geer. Chip makers turn to multicore processors. *Computer*, 38(5):11–13, 2005.

[10] N. E. Gold. *The meaning of legacy systems.* Univ. of Durham, Dept. of Computer Science, 1998.

[11] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. Nvidia tesla: A unified graphics and computing architecture. *Micro, IEEE*, 28(2):39–55, 2008.

[12] M. Markowitz. Pgi cuda fortran now available from the portland group. http://www.pgroup.com/about/news.htm#37, Nov. 2009.

[13] M. Méndez. Fortran refactorings for legacy systems. *Universidad Naciona de La Plata*, 2011.

[14] M. Metcalf. The seven ages of fortran. *Journal of Computer Science and Technology*, 11(1):1–8, 2011.

[15] C. Michael. Feathers. working effectively with legacy code, 2005.

[16] NVIDIA Corporation. Openacc | nvidia developer zone. http://developer.nvidia.com/cuda/openacc.

[17] NVIDIA Corporation. Nvidia whitepaper, nvidia's next generation cuda compute architecture: Fermi. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_ Architecture_Whitepaper.pdf, 2009.

[18] NVIDIA Corporation. Nvidia cuda programming guide, 2011.

[19] NVIDIA Corporation. Cuda fortran. http://developer.nvidia.com/cuda/cuda-fortran, 2012.

[20] NVIDIA Corporation. NVIDIA Tesla C2075 Companion Processor. Calculate Results Exponentially Faster. http://www.nvidia.com/content/PDF/datasheet/NV_DS_Tesla_C2075_Sept11_US_HR.pdf, Sept. 2011.

[21] OpenACC-Standard.org. The openacc application programming interface. version 1.0. http://openacc.org/download-area, Nov. 2011.

[22] OpenMP Architecture Review Board. Openmp application program interface - version 3.1. http://openmp.org/wp/, July 2011.

[23] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore gpus.

[24] The Portland Group. CUDA Fortran. Programming Guide and Reference. http://www.pgroup.com/doc/pgicudaforug.pdf, Release 2012.

[25] The Portland Group. PGI Compiler Reference Manual. Parallel Fortran, C and C++ for Scientists and Engineers. http://www.pgroup.com/doc/pgiref.pdf, Release 2012.

[26] The Portland Group. PGI Compiler User's Guide. Parallel Fortran, C and C++ for Scientists and Engineers. http://www.pgroup.com/doc/pgiug.pdf, Release 2012.

[27] F. G. Tinetti and M. Méndez. Fortran legacy software: source code update and possible parallelisation issues. *ACM SIGPLAN Fortran Forum*, 31(1):5–22, April 2012.

[28] M. Wolfe. GPU Programming with CUDA (C and PGI CUDA Fortran). Part 2: CUDA. http://www.pgroup.com/lit/presentations/pgi-gpu-tutorial-cuda.pdf, March 2011.

[29] M. Wolfe. GPU Programming withthe PGI Accelerator Programming Model. Part 3: The PGI Accelerator Programming Model. http://www.pgroup.com/lit/presentations/pgi-gpu-tutorial-accelerator.pdf, March 2011.

[30] M. Wolfe. Cuda fortran: The next level. PGI Insider, Technical Information from The Portland Group, http://www.pgroup.com/lit/articles/insider/v2n3a1.htm, Sep. 2010.

Apendix A: CUDA Matrix Multiplication

```
!============================================================
! CUDAmm.f90
!============================================================
module modmm
  use cudafor
contains
  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  subroutine initmat(m, n, val)
    implicit none
    double precision :: m(n,n)
    integer          :: n
    double precision :: val

    integer          :: i, j

    do i = 1, n
      do j = 1, n
        m(i,j) = val
      end do
    end do
  end subroutine initmat

  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  ! This subroutine does not compute, just calls the kernel
  !and handles memory transfers)
  subroutine mm(a, b, c, n)
    implicit none
    double precision :: a(n,n), b(n,n), c(n,n)
    integer          :: n

    ! Data to be allocated in the GPU
    double precision, allocatable, device  :: a_d(:,:)
    double precision, allocatable, device  :: b_d(:,:)
    double precision, allocatable, device  :: c_d(:,:)

    ! Grid and Block definitions
    Type(dim3)                             :: gridspec, blockspec

    ! Allocate and initialize matrix data in GPU
    allocate(a_d(n,n))
    allocate(b_d(n,n))
    allocate(c_d(n,n))
    a_d = a
    b_d = b
    c_d = c

    ! Set grid and block specification
    gridspec  = dim3(n/10, n/10, 1)
    blockspec = dim3(10, 10, 1)

    ! Kernel call
    call cudamm<<<gridspec, blockspec>>>(a_d, b_d, c_d, n)

    ! Get the result from GPU memory
    c = c_d
  end subroutine mm



  ! continued...
  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
   attributes(global) subroutine cudamm(a, b, c, n)
    implicit none
    double precision, device :: a(n,n), b(n,n), c(n,n)
    integer, value           :: n

    integer                  :: k, ix, iy

    ix = (blockIdx%x-1) * blockdim%x + threadIdx%x
    iy = (blockIdx%y-1) * blockdim%y + threadIdx%y

    do k = 1, n
      c(ix, iy) = c(ix, iy) + a(k,ix)*b(k,iy)
    end do
  end subroutine cudamm

  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  subroutine chkval(m, n, val)
    implicit none
    double precision :: m(n,n)
    integer          :: n
    double precision :: val

    integer          :: i, j

    do i = 1, n
      do j = 1, n
        if (m(i,j) /= val) print *, "diff. at", i, j
      end do
    end do
  end subroutine chkval
end module modmm

!==============================================================

program matmul
  use modmm
  implicit none

  integer                         :: n
  double precision, allocatable  :: a(:,:)
  double precision, allocatable  :: b(:,:)
  double precision, allocatable  :: c(:,:)

  n = 4000

  allocate(a(n,n))
  allocate(b(n,n))
  allocate(c(n,n))

  ! initialize matrices
  call initmat(a, n, 1d0)
  call initmat(b, n, 1d0)
  call initmat(c, n, 0d0)

  ! c = a x b
  call mm(a, b, c, n)

  ! check result
  call chkval(c, n, dble(n))

  deallocate(a, b, c)
end program  matmul
```

Compiled with `pgfortran -fast -Mcuda -o CUDAmm CUDAmm.f90`

Apendix B: Matrix Multiplication Using a Kernel Loop Directive

```fortran
!============================================================
! KLDmm.f90
!============================================================
module modmm
contains
  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  subroutine initmat(m, n, val)
    implicit none
    double precision :: m(n,n)
    integer          :: n
    double precision :: val

    integer          :: i, j

    do i = 1, n
      do j = 1, n
        m(i,j) = val
      end do
    end do
  end subroutine initmat


  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  ! This subroutine has been changed to use a cuf directive
  subroutine mm(a, b, c, n)
    implicit none
    double precision :: a(n,n), b(n,n), c(n,n)
    integer          :: n


    ! Data to be allocated in the GPU
    double precision, device  :: a_d(n,n)
    double precision, device  :: b_d(n,n)
    double precision, device  :: c_d(n,n)

    ! matrix indexes
    integer                   :: i, j, k

    ! Initialize matrix data in GPU
    a_d = a
    b_d = b
    c_d = c

    !$cuf kernel do(2) <<< (n/10,n/10), (10,10) >>>
    do i = 1, n
      do j = 1, n
        do k = 1, n
          c_d(i,j) = c_d(i,j) + a_d(i,k)*b_d(k,j)
        end do
      end do
    end do

    ! Get the result from GPU memory
    c = c_d
  end subroutine mm


  ! continued...
  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  subroutine chkval(m, n, val)
    implicit none
    double precision :: m(n,n)
    integer          :: n
    double precision :: val

    integer          :: i, j

    do i = 1, n
      do j = 1, n
        if (m(i,j) /= val) print *, "diff. at", i, j
      end do
    end do
  end subroutine chkval
end module modmm

!==============================================================

program matmul
  use modmm
  implicit none

  integer                         :: n
  double precision, allocatable  :: a(:,:)
  double precision, allocatable  :: b(:,:)
  double precision, allocatable  :: c(:,:)

  n = 4000

  allocate(a(n,n))
  allocate(b(n,n))
  allocate(c(n,n))

  ! initialize matrices
  call initmat(a, n, 1d0)
  call initmat(b, n, 1d0)
  call initmat(c, n, 0d0)

  ! c = a x b
  call mm(a, b, c, n)

  ! check result
  call chkval(c, n, dble(n))

  print *, "Last value...", c(n,n)
  deallocate(a)
  deallocate(b)
  deallocate(c)
end program  matmul

!==============================================================
```

Compiled with `pgfortran -fast -Mcuda -o KLDmm KLDmm.f90`

Apendix C: Matrix Multiplication Using OpenACC Directives

```
!==============================================================
! ACCmm.f90
!==============================================================
module modmm
contains
  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  subroutine initmat(m, n, val)
    implicit none
    double precision :: m(n,n)
    integer          :: n
    double precision :: val

    integer          :: i, j

    do i = 1, n
      do j = 1, n
        m(i,j) = val
      end do
    end do
  end subroutine initmat

  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  subroutine mm(a, b, c, n)
    implicit none
    double precision :: a(n,n), b(n,n), c(n,n)
    integer          :: n

    integer          :: i, j, k

    ! making the product
    !$acc region
    do i = 1, n
      do j = 1, n
        do k = 1, n
          c(i,j) = c(i,j) + a(i,k)*b(k,j)
        end do
      end do
    end do
    !$acc end region
  end subroutine mm

  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  subroutine chkval(m, n, val)
    implicit none
    double precision :: m(n,n)
    integer          :: n
    double precision :: val

    integer          :: i, j

    do i = 1, n
      do j = 1, n
        if (m(i,j) /= val) print *, "diff. at", i, j
      end do
    end do
  end subroutine chkval
end module modmm
!==============================================================


! continued...
```

```
program matmul
  use modmm
  implicit none

  integer                         :: n
  double precision, allocatable  :: a(:,:)
  double precision, allocatable  :: b(:,:)
  double precision, allocatable  :: c(:,:)

  n = 4000

  allocate(a(n,n))
  allocate(b(n,n))
  allocate(c(n,n))

  ! initialize matrices
  call initmat(a, n, 1d0)
  call initmat(b, n, 1d0)
  call initmat(c, n, 0d0)

  ! c = a x b
  call mm(a, b, c, n)

  ! check result
  call chkval(c, n, dble(n))

  print *, "Last value...", c(n,n)
  deallocate(a)
  deallocate(b)
  deallocate(c)
end program  matmul

!==============================================================
```

Compiled with `pgfortran -fast -ta=nvidia,time -Minfo=accel -o ACCmm ACCmm.f90`