

# CUDA/Ada - An Ada binding to CUDA

Reto Bürki, Adrian-Ken Rügsegger

February 13, 2012

University of Applied Sciences Rapperswil (HSR), Switzerland

## Abstract

This paper outlines an approach for the implementation of an Ada binding to NVIDIA's CUDA parallel computing platform and programming model. While describing the origination process, the paper also documents the required steps to write a modern, maintainable language binding for Ada in general.

## 1 Introduction

CUDA<sup>1</sup> is a parallel computing architecture developed by NVIDIA. CUDA enables developers to access native GPU<sup>2</sup> instruction sets and memory from higher level languages like C for general purpose computing. A GPU has a much larger number of cores compared to a CPU, with each core capable of running thousands of threads. This can improve performance tremendously especially for scientific applications where many expensive calculations can be performed in parallel. It would exceed the scope of this article to give a full introduction to CUDA and the CUDA API. For further information the reader is directed to CUDA-specific literature [3, 4].

CUDA wrappers exist for many languages but until now no such wrapper or binding existed for Ada. In this paper we present the design and implementation of a CUDA binding<sup>3</sup> for the Ada programming language. Having access to CUDA makes it possible to benefit from the advantages and processing power of a GPU in Ada applications.

The paper is organized as follows: We will first give a short introduction of the Ada programming language in section 1.1. Section 2 then outlines the CUDA/Ada binding implementation in detail. The design goals are described as well as the complete design process taken including the automated generation of Thin-binding code (see section 1.1.3 for an explanation of Thick- and Thin-bindings in Ada). The features of the binding are explained in depth in subsection 2.3.

In section 3 we combine most of the implemented CUDA/Ada features and demonstrate the usage of CUDA/Ada with concrete example code.

---

<sup>1</sup>Computer Unified Device Architecture

<sup>2</sup>Graphics Processing Unit

<sup>3</sup>Wrapper = binding

Section 4 compares the performance of code using CUDA/Ada with native Ada. The aim is to illustrate the performance gain which can be achieved by using a GPU instead of a CPU. Furthermore, the speed of CUDA/Ada compared to native CUDA is examined.

## 1.1 The Ada Programming Language

Ada is a structured, strongly typed programming language. The language has initially been designed by Jean Ichbiah from Honeywell Bull in the 1970s. Ada has a very similar structure to Pascal and can be considered a “Wirth”-language<sup>4</sup>.

The development of Ada was initiated by the US Department of Defense (DoD) in order to consolidate and supersede the hundreds of programming languages used in their countless projects. The new language should comply with all identified DoD requirements (dubbed “Steelman Language Requirements” [10]), which focused strongly on security and safety.

Ada was the first standardized high-level programming language [9]. The current version is Ada 2005 [6] which supports all modern programming paradigms. The next major version of the language is planned to be complete in 2012 and will naturally be called Ada 2012.

Ada compilers, before used in practice, have to pass a standardized test suite which guarantees the compliance of the compiler with the Ada standard. Since Ada provides many features which aid in the development of safety and security critical applications, it is nowadays mostly used in such areas where these aspects are important. The primary industries making use of Ada are namely avionics, railway systems, banking, military and space technology.

The language is named after Lady Ada Lovelace (1815-1852), the daughter of Lord Byron<sup>5</sup> who is considered to be the first computer programmer.

### 1.1.1 Compilers

Although many Ada compilers exist this article will focus on the GNAT Ada compiler. GNAT is a free-software compiler for the Ada programming language which is part of the GNU Compiler Collection (gcc). It supports all versions of the language and is available on most operating systems.

There are basically three relevant “variants” of GNAT available, differing mainly in their licensing schemes:

- GNAT Pro, commercially supported and actively maintained by the company AdaCore<sup>6</sup>. New releases are frequent but provided only to paying customers, under the GNAT-modified GPL (GMGPL)<sup>7</sup>.
- The GNAT GPL Edition; also maintained by AdaCore. The licensing terms of the run-time library are pure GPL<sup>8</sup>. Thus, binaries produced with this variant of GNAT can only be distributed under the GPL or any compatible license.

---

<sup>4</sup>Niklaus Wirth was the designer of several programming languages such as Pascal or Modula, see [http://en.wikipedia.org/wiki/Niklaus\\_Wirth](http://en.wikipedia.org/wiki/Niklaus_Wirth)

<sup>5</sup>Ada Lovelace - [http://en.wikipedia.org/wiki/Ada\\_Lovelace](http://en.wikipedia.org/wiki/Ada_Lovelace)

<sup>6</sup><http://www.adacore.com>

<sup>7</sup>[http://en.wikipedia.org/wiki/GNAT\\_Modified\\_General\\_Public\\_License](http://en.wikipedia.org/wiki/GNAT_Modified_General_Public_License)

<sup>8</sup>GPL - <http://www.gnu.org/copyleft/gpl.html>

- The FSF<sup>9</sup> variant of GNAT. This is the Ada front-end which has been distributed as part of GCC since version 3.1. For version 4.4 and later, the license terms are GPL version 3 with GCC Runtime Library Exception which is similar in spirit to the GMGPL.

For more details on GNAT and the different licensing schemes see for example the “Debian Ada policy” [1].

### 1.1.2 Bindings in Ada

Ada provides the `Import pragma` to access functionality which is written in another programming language. Pragas are directives which control the compiler and have the following general form:

```
1 pragma Name (Parameter_List);
```

The `pragma Import` directs the compiler to use code or data objects written in a foreign computer language. The following example code imports the `bind` system call:

```
1 function C_Bind
2 (S      : Interfaces.C.int;
3  Name   : System.Address;
4  Namelen : Interfaces.C.int)
5   return Interfaces.C.int;
6 pragma Import (C, C_Bind, "bind");
```

The function `C_Bind` can then be used in the code like any other Ada function, but when called the call is forwarded to the underlying function in the C library.

Ada provides a set of predefined packages that make it easier to interface with C. The primary package is named `Interfaces.C`, which contains definitions for C types in Ada. The `System` package provides the `Address` type, which is treated like a (void) pointer when passed to an imported function.

Using `pragma Import` enables developers to interface with libraries written in a different programming language, for example the CUDA library from NVIDIA which provides a C API.

A library which wraps another library not written in Ada is called a “binding”. Other programming languages also use the term “wrapper” or “library wrapper”.

### 1.1.3 Thin-/Thick-Binding

Bindings are further divided into Thin- and Thick-bindings. A “thin” binding provides a one-to-one mapping to Ada of whatever interface the foreign library provides. Such a binding is very easy to create. It is often just a matter of diligence. Unfortunately, it is cumbersome to work with a thin binding because of the direct mapping it’s “look and feel” is much like the foreign programming language the actual imported code is written in, e.g. C. Furthermore, a thin binding does not provide the protection normally guaranteed by Ada since calls to imported functions are simply delegated to the underlying library. If for

<sup>9</sup>Free Software Foundation

example a null pointer is provided to an imported function and this function dereferences the given pointer there is nothing Ada can do about the error at this level.

It is very laborious to write a thin binding for an existing library, especially when the library to be wrapped is of considerable size. So having this layer generated automatically by a tool would decrease the work required to create a binding tremendously. Section 2.2 demonstrates how this has been done for the CUDA/Ada project using the GNAT GPL compiler from AdaCore (see section 1.1.1).

A "thick" binding provides a more abstract, Ada-like view of the foreign library or program. It hides all the foreign language constructs behind proper Ada types and operations. Additionally the thick-binding layer usually checks that no dangerous values are passed on to the thin-binding functions (such as the null pointer in the example given in the previous section). Unfortunately, while thick bindings are easier to work with, it takes more work and time to create the proper types and find the right level of abstraction.

Often thin and thick binding layers are used in conjunction. The (automatically generated) thin binding wraps the foreign language library and a thick layer then abstracts the thin binding to provide an Ada-like "look and feel" to the programmer using the binding. This separation also improves maintainability because both layers can be adapted when needed.

## 2 CUDA/Ada

This section describes the implementation of the Ada binding to CUDA in detail.

### 2.1 Design Goals

The design of CUDA/Ada has been heavily inspired by another binding for CUDA, namely PyCUDA<sup>10</sup>. PyCUDA is written by Andreas Klöckner from NYU (New York University) and provides a very comfortable, high-level access to CUDA from Python.

For this reason the design of PyCUDA has been analyzed and studied [12]. It was the goal of the CUDA/Ada project to translate some of the prominent features of PyCUDA into the Ada domain. The intended features and goals derived from PyCUDA are:

- Seamless access to CUDA from Ada

CUDA/Ada should provide a comfortable thick binding (see 1.1.3) to CUDA. The types and operations provided by CUDA/Ada should hide CUDA C semantics and should fit well in the Ada "way of programming".

- High abstraction

The binding should provide a high abstraction level. The user of the binding should not need to care about the low-level, intricate details of the CUDA API.

---

<sup>10</sup><http://mathematician.de/software/pycuda>

- Auto-initialization

A CUDA context must be initialized before other CUDA functions can be used. This is done by calling the `cuInit` and `cuCtxCreate` functions. One goal was to eliminate the need of this explicit initialization in CUDA/Ada. It should be handled automatically by the binding in a manner transparent to the user.

- JIT-Compilation of CUDA kernels<sup>11</sup>

CUDA/Ada should provide the facility to compile CUDA kernels when needed (just in time compilation). It should be possible to have CUDA kernel code “inline” in Ada program code.

- Convenient argument handling

The CUDA/Ada library should take care of device memory management and kernel call argument copying. This functionality should be easy to use and transparently handle any implementation peculiarities.

- Error-handling using Ada Exceptions

Errors in CUDA should lead to an Ada exception with an explanatory and concise exception message.

- Speed

Programs using CUDA/Ada should reach “native” CUDA speed. The overhead imposed by Ada should be kept minimal.

- Automatically generated thin-binding

In order to guarantee good maintainability, it is important to have the thin binding-layer generated in an automated manner. If for example NVIDIA changes some parts of the CUDA API, it is easier to re-create the thin binding than to update it by hand.

## 2.2 Thin-Binding

The thin binding layer of CUDA/Ada has been automatically created using the `-fdump-ada-spec` option of the GNAT GPL compiler (see section 1.1.1). This option is present since GNAT GPL 2010 and allows to create Ada spec files from C/C++ headers. The patch implementing this feature has been merged into FSF GCC and the functionality is available in FSF GNAT 4.6.

The thin binding of CUDA/Ada has been created using the following commands:

```
gcc -c -fdump-ada-spec cuda.h
gcc -c -fdump-ada-spec cuda_runtime.h
```

CUDA/Ada imports both the CUDA driver and runtime API to provide it’s functionality (`cuda.h`, `cuda_runtime.h`).

---

<sup>11</sup>A function compiled for the GPU is called a kernel. The kernel is executed on the device by many different threads in parallel.

To make the thin binding code work for multiple host architectures, the binding must be generated on those hosts as well. CUDA/Ada currently supports i686 and x86\_64 architectures. The build logic automatically detects the correct thin binding to use depending on the host architecture:

```
ARCH ?= $(shell uname -m)
gnatmake -Pcuda -XARCH=$(ARCH)
```

The detected architecture is passed on to the GNAT project files used by gnatmake to build the Ada code. The top level CUDA/Ada project file includes the thin binding specific project file:

```
1 with "thin/binding";
```

The thin binding project determines the source directory with the generated architecture-specific files depending on the ARCH variable:

```
1 type Arch_Type is ("x86_64", "i686");
2 Arch : Arch_Type := external ("ARCH", "x86_64");
3
4 for Source_Dirs use (".", ARCH);
```

It would also be possible to create the thin binding files during the build instead of including the pre-generated files. This functionality is not yet implemented in CUDA/Ada.

## 2.3 Thick-Binding

This section describes the abstraction layer provided by CUDA/Ada, the thick binding (see section 1.1.3).

### 2.3.1 Autoinit package

In order to relieve the user of CUDA/Ada from the need to explicitly initialize CUDA, the Autoinit package takes care of this task automatically. The user only needs to include the package in the source code like so:

```
1 with CUDA.Autoinit;
2 pragma Unreferenced (CUDA.Autoinit);
```

The compiler pragma Unreferenced is used to avoid warnings like:

```
add.adb:23:10: warning: unit "Autoinit" is not referenced
```

This warning is caused by the fact that the package is not referenced in the code, the complete CUDA initialization takes place during program elaboration. Elaboration is the process performed by the Ada Run-Time System (RTS) in order to “elaborate” variable and constant declarations before the actual program is run (see [7, 8]). The following functions are called to initialize CUDA:

```
cuInit
cuDeviceGet
cuCtxCreate
```

See the CUDA reference manual [5] for an explanation of these functions.

The `Autoinit` package also handles the task to release the CUDA context again once the program terminates. This is done using the following CUDA function:

```
cuCtxDestroy
```

### 2.3.2 Source modules

The `Compiler` package introduces the concept of a source module. Source modules are used to define CUDA kernels “inline” directly in Ada code:

```

1 N      : constant                := 32 * 1024;
2 Src    : Compiler.Source_Module_Type :=
3   Compiler.Create
4   (Preamble => "#define N" & N'Img,
5    Operation => "__global__ void add(float *a, float *b) {"
6    & "  int tid = blockIdx.x;"
7    & "  while (tid < N) {"
8    & "    b[tid] = a[tid] + 10;"
9    & "    tid += blockDim.x;"
10   & "  }"});

```

Using a source module, a CUDA kernel preamble can be specified. A preamble is used to pass constants defined in Ada to the CUDA kernel to avoid duplicate declarations.

### 2.3.3 JIT-Compiler

CUDA/Ada provides a just-in-time compiler. This compiler takes care of compiling a source module, as described in the previous section 2.3.2, to CUBIN binary code<sup>12</sup>. Using the `Compile` function of the `Compiler` package a CUDA/Ada source module can be compiled to a module (`Module_Type`), which is loaded onto the GPU and ready for execution.

The module type provides the `Get_Function` operation to access the compiled and loaded module function inside the GPU. A function (`Function_Type`) can be launched on the GPU using the `Call` procedure. The following example code demonstrates how this is done:

```

1 ...
2 Func    : Compiler.Function_Type;
3 Module  : Compiler.Module_Type;
4 begin
5   Module := Compiler.Compile (Source => Src);
6   Func   := Compiler.Get_Function (Module => Module,
7                                   Name   => "add");
8
9   Func.Call
10  (Args =>
11   (1 => In_Arg (Data => A),
12    2 => In_Arg (Data => B),
13    3 => Out_Arg (Data => C'Access)));

```

<sup>12</sup>CUBIN files are CUDA binary files that are compiled for a specific CUDA architecture.

In this example the same source module as in section 2.3.2 (defining the `add` CUDA kernel) is used. The `Call` procedure also passes *In*, *Out* and *InOut* arguments to the CUDA kernel. Arguments are explained in depth in section 2.3.4. The `Call` procedure allows the user to specify the grid and block dimensions of the CUDA kernel launch:

```

1   Func.Call
2   (Args      =>
3     (1 => In_Arg (Data => A),
4     2 => In_Arg (Data => B),
5     3 => Out_Arg (Data => C'Access)),
6   Grid_Dim_X => 128,
7   Grid_Dim_Y => 1,
8   Grid_Dim_Z => 1,
9   Block_Dim_X => 1,
10  Block_Dim_Y => 1,
11  Block_Dim_Z => 1);

```

The compiler uses a hash sum to detect whether compilation of a CUDA kernel is required or if the exact same kernel has already been compiled before. Compiled kernels are stored inside the compiler cache in CUBIN format for faster access.

If a previously compiled kernel is requested, the compiler just loads the pre-compiled CUBIN code onto the GPU and skips the compilation steps speeding up the process.

### 2.3.4 Argument handling

The data on which CUDA kernels typically operate are passed along as function arguments. When executing a kernel in CUDA/Ada (see section 2.3.3) the procedure argument `Args` must match the type signature of the corresponding kernel. `Args` is an arbitrary-length array consisting of `Arg_Type` objects. Such objects can be created by instantiating the generic `Arg_Creators` package and using the provided `In_Arg`, `In_Out_Arg` and `Out_Arg` functions.

The argument creator functions create `Arg_Type` object instances which take care of copying the given data to or from the GPU. `In` arguments copy the data to the device when they are created. `Out` objects copy the data back from the device to the host when they go out of scope and `InOut` perform both operations. Memory allocation and freeing on the GPU is also handled seamlessly. The use of these functions feels natural to the Ada language since it mimics the specification of formal parameter modes for a given argument<sup>13</sup> as they have similar names (`in`, `out` and `in out`).

The argument handling is analogous to PyCUDA's `ArgumentHandler` classes as implemented in the `pycuda.driver` module. Almost equal flexibility is achieved by using Ada generics. The only additional step is the instantiation of the `Arg_Creators` package for all distinct kernel argument types. The following example shows how the generic can be used to create an `In` argument for Ada's `Real_Matrix` type:

```

1   ...
2   package Matrix_Args is new CUDA.Compiler.Arg_Creators

```

<sup>13</sup>Parameter modes specify the data flow of subprogram arguments, see [6], section 6.2

```

3     (Data_Type => Ada.Numerics.Real_Arrays.Real_Matrix);
4     use Matrix_Args;
5
6     Matrix : Ada.Numerics.Real_Arrays.Real_Matrix
7         := (1 .. N => (1 .. N => 0.0));
8     Arg      : CUDA.Compiler.Arg_Type
9         := In_Arg (Data => Matrix);
10    begin
11        ...

```

Generics in Ada are used to implement algorithms and data structures in terms of types which are specified upon instantiation of the generic package. Many container data structures provided by Ada 2005 such as linked lists or maps are implemented as generics. Generic programming was pioneered by Ada as this feature has been part of the language since its first incarnation in 1983. Similar mechanisms exist in other programming languages such as templates in C++ or generics in Java.

The following CUDA kernel invocation example shows how the argument creator functions are used to create anonymous objects which are instantiated just prior to the execution and finalized upon return of the `Call` procedure:

```

1     Func.Call
2         (Args =>
3             (1 => In_Arg (Data => A),
4              2 => In_Arg (Data => B),
5              3 => Out_Arg (Data => C'Access)));

```

### 2.3.5 Ada exceptions

Errors inside the CUDA API are translated to proper Ada exceptions by CUD-A/Ada. When using the functionality provided by the thick-binding all low-level CUDA API calls are checked for errors. The following code snippet shows how this is done inside the `CUDA.Driver.Device_Count` function:

```

1     Check_Result (Code => cuda_runtime_api_h.cudaGetDeviceCount
2                   (arg1 => Dev_Count'Access),
3                   Msg => "Unable to get device count");

```

If for some reason the call to `cudaGetDeviceCount` does not succeed the procedure `Check_Result` will raise an exception with the specified error message and the corresponding error cause as returned by the CUDA API. The following exception message illustrates this.

```

Execution terminated by unhandled exception
Exception name: CUDA.CUDA_ERROR
Message: Could not get function Matrix_Mul (Not found)
Call stack traceback locations:
0x4079b6 0x40bc3a 0x406a4b 0x406299
0x7f159e4afc4b 0x405bd7

```

Analysis of this exception message leads to the conclusion that a CUDA function `Matrix_Mul` has been requested from the GPU, but such a function does not exist.

### 2.3.6 Device enumeration

The `CUDA.Driver` package provides a device type needed to enumerate CUDA devices and query their properties. Using this package it is possible to iterate over all CUDA devices of a system by specifying a procedure which is called for each device found. The following example code displays the total number of CUDA devices and then iterates over all devices to print their names:

```

1  with Ada.Text_IO;
2  with CUDA.Driver; use CUDA.Driver;
3
4  procedure Enum_Devices
5  is
6      -- Print CUDA device name.
7      procedure Print_Name (Dev : Device_Type)
8      is
9          begin
10             Ada.Text_IO.Put_Line (Name (Dev));
11         end Print_Name;
12     begin
13         Ada.Text_IO.Put_Line (Device_Count'Img & " device(s):");
14         Iterate (Process => Print_Name'Access);
15     end Enum_Devices;

```

## 3 Example

The following vector addition example code combines most of the features described in section 2.3.

```

1  with Ada.Text_IO;
2  with Ada.Numerics.Real_Arrays;
3
4  with CUDA.Autoinit;
5  with CUDA.Compiler;
6
7  pragma Unreferenced (CUDA.Autoinit);
8
9  procedure Add
10 is
11     use CUDA;
12     use Ada.Numerics.Real_Arrays;
13
14     package Real_Vector_Args is new Compiler.Arg_Creators
15         (Data_Type => Ada.Numerics.Real_Arrays.Real_Vector);
16     use Real_Vector_Args;
17
18     N : constant := 32 * 1024;
19
20     A      : Real_Vector      := (1 .. N => 2.0);
21     B      : Real_Vector      := (1 .. N => 2.0);
22     C      : aliased Real_Vector := (1 .. N => 0.0);
23     Src    : Compiler.Source_Module_Type;
24     Func   : Compiler.Function_Type;

```

```

25   Module : Compiler.Module_Type ;
26   begin
27     Src := Compiler.Create
28       (Preamble => "#define N" & N'Img,
29        Operation =>
30         "__global__ void add(float *a, float *b, float *c) {"
31         & "    int tid = blockIdx.x;"
32         & "    while (tid < N) {"
33         & "        c[tid] = a[tid] + b[tid];"
34         & "        tid += blockDim.x;"
35         & "    }"}");
36
37     Module := Compiler.Compile (Source => Src);
38     Func    := Compiler.Get_Function (Module => Module,
39                                     Name    => "add");
40
41     Func.Call
42       (Args =>
43        (1 => In_Arg (Data => A),
44         2 => In_Arg (Data => B),
45         3 => Out_Arg (Data => C'Access)));
46   end Add;

```

- Line 4  
A CUDA context is automatically initialized by the inclusion of the `Autoinit` package (see section 2.3.1). The package is marked as unreferenced on Line 7 because it is not needed any further.
- Line 14-16  
An argument creator package is instantiated which is used on lines 43-45 to pass the `Ada.Numerics.Real_Arrays.Real_Vector` arrays to the CUDA kernel and copy back the result (section 2.3.4).
- Line 27-35  
A source module is created from the inline CUDA kernel. The `N` constant is passed to the kernel using the preamble feature of the `source module type` (section 2.3.2).
- Line 37  
The source module is compiled and loaded onto the GPU.
- Line 38  
A handle to the kernel implementing the vector addition is obtained by calling the `Get_Function` operation of the `Module_Type`.
- Line 41-45  
The kernel is launched on the GPU. Arguments are passed using in/out/i-nout argument creators which handle device memory allocation and data initialization automatically. The result of the vector addition is stored in variable `C`.

The last three items in the list are all explained in section 2.3.3.

|                        |                         |
|------------------------|-------------------------|
| Processor              | AMD Phenom II X4 940    |
| Operating System       | Debian Linux 6.0        |
| Kernel                 | 2.6.32-5-amd64          |
| Ada Compiler           | FSF GNAT 4.4.5          |
| NVIDIA Graphics Driver | 270.41.19, Linux 64-bit |
| NVIDIA CUDA Toolkit    | 4.0.17, Linux 64-bit    |

Table 1: Measurement parameters

## 4 Performance Analysis

One of the project’s goals was to bring the potential speed benefits of CUDA to Ada programs. An additional design goal (see 2.1) was to have comparable performance to native CUDA C code. The impact of Ada and its runtime environment on the execution speed should be as small as possible.

To determine the degree of target achievement the runtime of the matrix multiplication operation was measured. A native Ada, a CUDA/Ada and two native CUDA implementations were benchmarked.

The “\*”-operator in the `Ada.Numerics.Real_Arrays` package was used for the Ada code executing the multiplication on the CPU. The CUDA/Ada version used an adapted kernel from NVIDIA’s SDK matrix multiplication code sample<sup>14</sup>. The identical kernel was used for the native CUDA Runtime and Driver API implementations.

The benchmarking data for the analysis in this section was gathered by executing the performance measurement code on an AMD Phenom II X4 940 Quadcore Processor with a NVIDIA GeForce GTX 560 Ti graphics card. Table 1 lists the relevant system information:

The following listing shows detailed information about the GPU used for benchmarking. The data was produced using CUDA/Ada’s `enum_devices` tool.

```
$ obj/enum_devices
Found 1 CUDA device(s):
Name           : GeForce GTX 560 Ti
Compute capability : 2.1
Clock rate      : 1660000
Device copy overlap : TRUE
Kernel exec timeout : TRUE
Total global mem : 1072889856
Total constant mem : 65536
Max mem pitch    : 2147483647
Texture alignment : 512
Multiprocessor count : 8
Shared mem per mp : 49152
Registers per mp  : 32768
Threads in warp   : 32
Max threads per block : 1024
Max thread dimensions : ( 1024, 1024, 64 )
Max grid size     : ( 65535, 65535, 65535 )
```

<sup>14</sup><http://developer.nvidia.com/cuda-cc-sdk-code-samples#matrixMul>

| Implementation   | Total time [s] |
|------------------|----------------|
| Ada (CPU)        | 0.977781       |
| CUDA Runtime API | 0.123061       |
| CUDA Driver API  | 0.077797       |
| CUDA/Ada         | 0.078863       |

Table 2: Measurement results

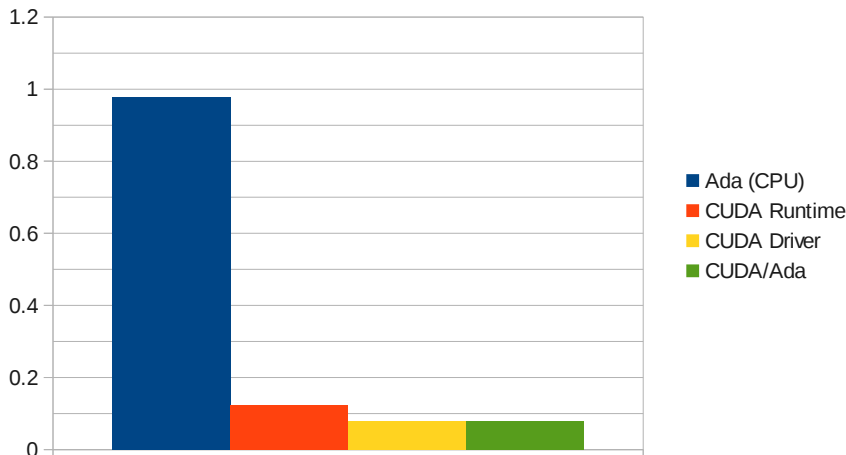


Figure 1: Performance measurement chart

Table 2 lists the cumulated execution times of performing the matrix multiplication operation on a 512 by 512 matrix 20 times. All CUDA implementations used the same kernel, a grid size of 32 and a block size of 16.

As expected the Ada operation executed on the CPU is much slower than the other implementations, which leverage the power of CUDA. While the CPU has to perform each calculation sequentially, the CUDA kernel computes many intermediate results in parallel. Since the multiplication operation of the `Ada.Numerics.Real_Arrays` package is single-threaded the operation is performed on a single CPU core and does not fully utilize the Quadcore CPU. An implementation of the “\*” operator, which would use all available CPU cores would decrease the overall runtime, but since GPUs have orders of magnitude more threads it will still be significantly slower than the CUDA implementations.

Somewhat surprisingly CUDA/Ada is faster than the CUDA runtime API benchmarking code. The reason for this is that the higher-level runtime API (see [4], section 3.2) of CUDA provides implicit initialization, context and module management. The necessary code is automatically generated by `nvcc`<sup>15</sup>. Since CUDA/Ada uses the driver API (see [4], section 3.3), initialization and all other steps have to be done explicitly. This is achieved by using CUDA/Ada’s thick-binding packages such as `Autoinit` or `Compiler` as described in chapter 2.3.

<sup>15</sup>NVIDIA CUDA compiler

CUDA/Ada is careful to only perform the bare minimum of CUDA management operations and thus adds less overhead than the CUDA Runtime API.

The fact that measurements for the CUDA Driver API code are close to the ones for CUDA/Ada show that this is indeed the case.

As figure 1 shows CUDA/Ada's performance is excellent and on par with native CUDA code. No visible performance penalty could be measured, which might have been incurred by the Ada runtime.

## 5 Conclusions

This paper described in detail an approach to implement a modern and maintainable Ada language binding to NVIDIA's CUDA parallel computing platform and programming model. The terms thin- and thick-binding were introduced and their differences explained.

The rest of the paper presented and examined the CUDA/Ada binding by enumerating the design goals, discussing various specific features and showing the usage of the code by means of a commented example.

The example code presented in section 3 and the discussion of performance measurements show that the design goals listed in section 2.1 were all achieved:

- The thick-binding provides easy usage of CUDA from Ada.
- A high level of abstraction was achieved by providing packages like argument- or exception-handling, which go well with other Ada language constructs.
- Autoinitialization of CUDA context via the `Autoinit` package.
- Just-in-Time compilation and caching of compiled CUDA kernels provides flexibility as well as agility with regards to handling and execution of CUDA kernels.
- Speed of CUDA/Ada is very good, even in comparison to native CUDA code.
- The Thin-Binding was generated automatically, which makes it easy to adapt CUDA/Ada to future CUDA API changes.

Since we achieved the goals we set out to do, we are rather satisfied with CUDA/Ada. Obviously the supported features are limited but nevertheless does the current codebase enable Ada developers to access the potential of GPU's and their massively parallel architecture.

NVIDIA's recent announcement [11] to release the source code to their LLVM-based CUDA compiler might provide a different route to use CUDA from Ada. Support for the Ada programming language would need to be implemented, which in our opinion is a non-trivial task.

We hope that this paper might spark some interest for the Ada language and CUDA. It was also written in a manner to be useful as a guideline, for somebody looking to write an Ada binding for a different library. CUDA/Ada is opensource and all the code, documentation and further information is available on the project's website at <http://www.codelabs.ch/cuda-ada/>.

## References

- [1] Debian Policy for Ada, 2006-2009, Ludovic Brenta, Stephen Leake
- [2] [GNAT User's Guide](#), GNAT, The GNU Ada Compiler, GCC version 4.7.0
- [3] CUDA by Example, An Introduction to General-Purpose GPU Programming, Jason Sanders, Edward Kandrot
- [4] NVIDIA CUDA C Programming Guide, Version 4.0, 2011, NVIDIA Corporation
- [5] NVIDIA CUDA Library Documentation, Version 4.0, NVIDIA Corporation
- [6] [Ada Reference Manual](#), ISO/IEC 8652:1995(E) with Technical Corrigendum 1 and Amendment 1
- [7] Programming in Ada 2005, John Barnes, International Computer Science Series, Addison Wesley, 2006
- [8] A Detailed Description of the GNU Ada Run Time, Javier Miranda, University of Las Palmas de Gran Canaria, Spain, 2002
- [9] Ada Programming Language, ANSI/MIL-STD-1815A-1983, American National Standards Institute, Inc, 22 January 1983
- [10] Requirements for High Order Computer Programming Languages "STEEL-MAN", Department of Defense, June 1978
- [11] [NVIDIA Opens Up CUDA Platform by Releasing Compiler Source Code](#), NVIDIA Corporation, 13 December 2011
- [12] [PyCUDA documentation](#), Andreas Kloeckner, 2011