CUDA/Ada

An Ada binding to CUDA

Reto Bürki, Adrian-Ken Rüegsegger
University of Applied Sciences Rapperswil (HSR), Switzerland

1/16/2012

Master seminar: Program Analysis and Transformation

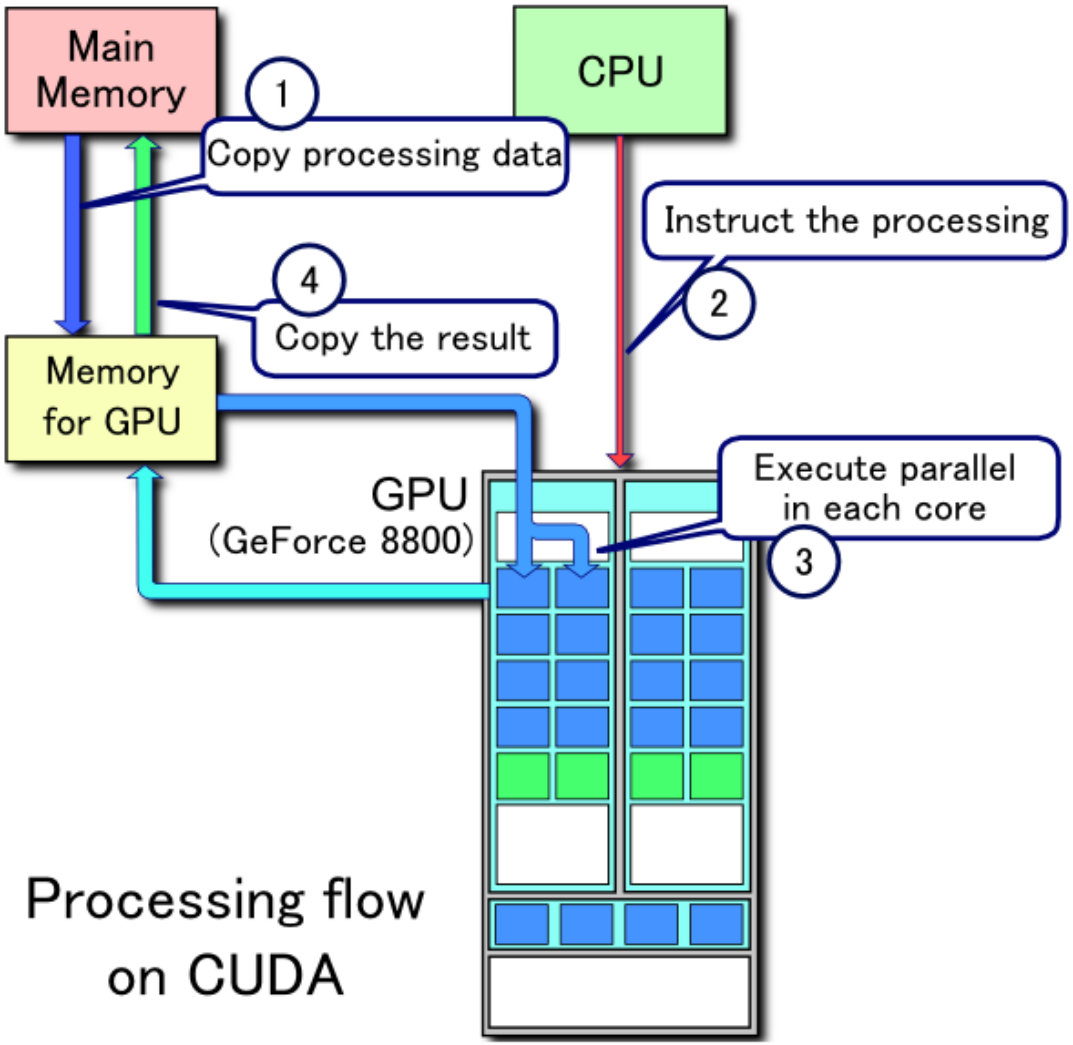
Outline

CUDA

What is CUDA?

- Parallel computing architecture developed by NVIDIA
- "Compute Unified Device Architecture"
- General purpose computation using GPU (GPGPU)
- Use GPU as dedicated massively parallel co-processor
- Tremendous performance improvements possible

CUDA Processing



Ada

The Ada programming language

- Structured, strongly typed programming language
- Initiated by the US Department of Defense (DoD)
- First standardized high-level programming language
- Emphasizes safety and security
- Supports all modern programming paradigms
- Current language standard is Ada 2005, next release 2012
- Mostly used in aviation, railway systems, banking, military and space technology

Ada Compiler

GNAT

- Free-software compiler for Ada
- Part of the GNU Compiler Collection (GCC)
- Supports all versions of Ada (83, 95, 2005)
- Available on most operating systems
- 100% compliant with Ada Conformity Assessment Test Suite (ACATS)

Motivation

Why CUDA/Ada?

- CUDA wrappers exist for many languages but not for Ada
- Make CUDA accessible for Ada developers
- Benefit from the advantages and processing power of a GPU in Ada applications
- Curiosity
  - How well do CUDA and Ada match up?
  - Can it be done in a nice way?
  - Possible perfomance penalties from Ada runtime?

Bindings in Ada

`Import` pragma

- Used to access functionality which is written in another programming language
- Pragmas are directives which control the compiler
- General form: `pragma Name (Parameter_List);`
- Predefined packages that make it easier to interface with C
  - `Interfaces.C`
  - `System`

Importing a C function

C function

```c
int bind(int sockfd, const struct sockaddr *addr
    , socklen_t addrlen);
```

Ada import

```
function C_Bind
 (S       : Interfaces.C.int;
  Name    : System.Address;
  Namelen : Interfaces.C.int)
  return Interfaces.C.int;
pragma Import (C, C_Bind, "bind");
```

Definition

Binding

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".

Thin-Binding

What is a Thin-Binding?

- One-to-one mapping of the foreign library interface to Ada
- Straight forward to create but time consuming and error prone
- Cumbersome to work with (because of direct mapping)
- No protection normally guaranteed by Ada

Thick-Binding

What is a Thick-Binding?

- Provides a more abstract, Ada-like view of foreign library
- Provides proper Ada types and operations
- Ensure safe usage of wrapper library
- Easier to work with but takes more effort and time to create

Thin- and Thick-Bindings

Using both

- Thin- and Thick-Binding layers are often used in conjunction
- Thin-Binding wraps foreign language library (low-level)
- Thick-Binding abstracts Thin-Binding to provide an Ada-like "look and feel"
- Separation improves maintainability because both layers can be adapted when needed

Design Goals

Inspiration

- CUDA/Ada heavily inspired by PyCUDA
- Great binding for Python from Andreas Klöckner

Design Goals II

Goals

- Seamless access to CUDA from Ada
- High abstraction
- Auto-initialization
- JIT-Compilation of CUDA kernels
- Convenient argument handling
- Error-handling using Ada Exceptions
- Speed
- Automatically generated Thin-Binding

Thin-Binding to CUDA

Creation

- Auto-generated using `-fdump-ada-spec` of GNAT
- CUDA/Ada imports CUDA driver and runtime API
  - `cuda.h`
  - `cuda_runtime.h`
- Support for i686 and x86_64
  - e.g. CUdeviceptr:`unsigned` vs.`unsigned_long_long`

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

Architecture (build logic)

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

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

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

Autoinit

Functionality

- Takes care of CUDA initialisation task
  - cuInit
  - cuDeviceGet
  - cuCtxCreate
- Handles release of CUDA context
  - cuCtxDestroy

```
1 with CUDA.Autoinit;
```

Source-Modules

Functionality

- Used to define CUDA kernels “inline”

```
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 += gridDim.x;"
10     & "}}");
```

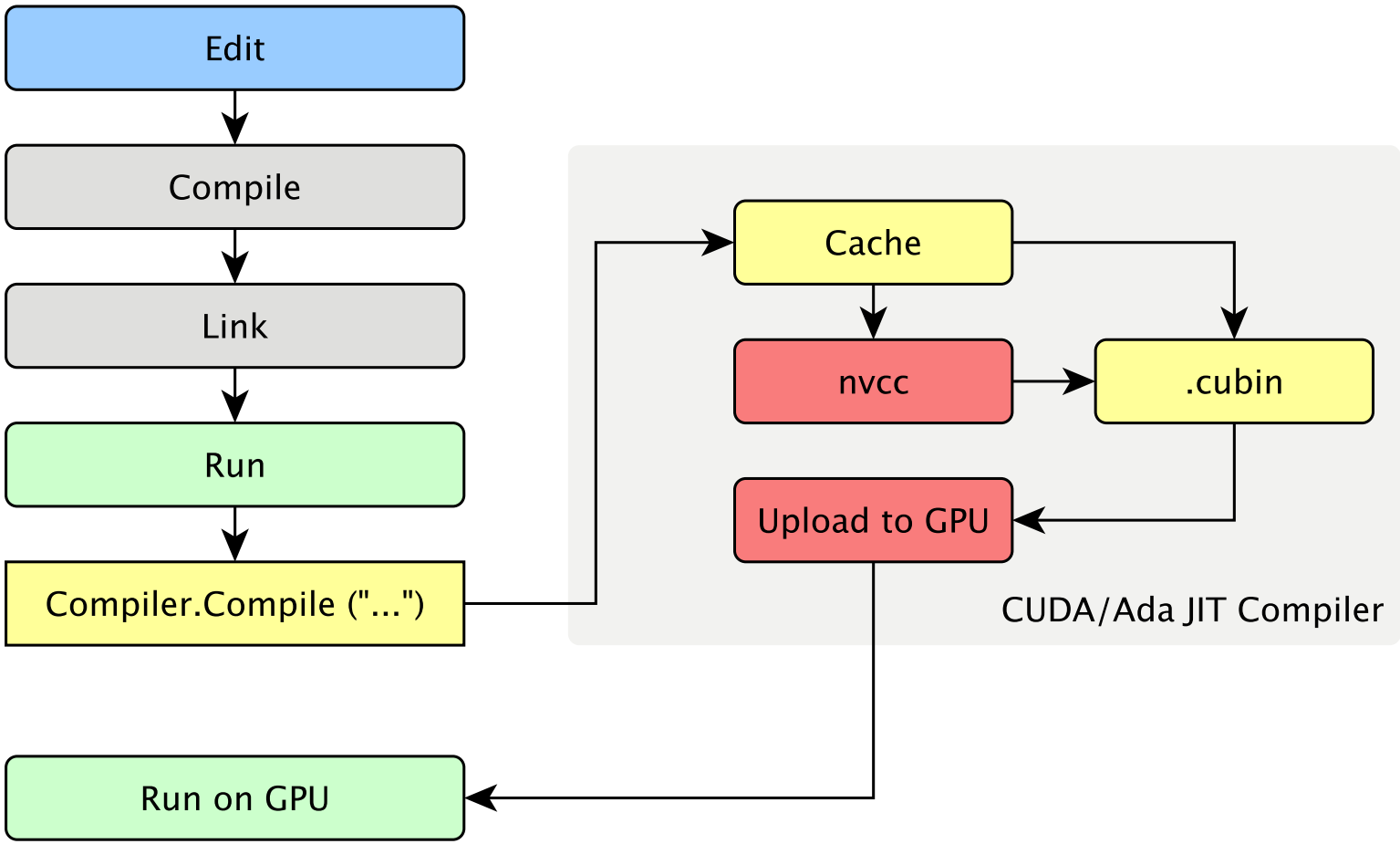
JIT-Compiler

Functionality

- Compile CUDA kernels at runtime
- Uses `nvcc` to generate CUBIN binary code
- Upload modules to GPU
- Caching of compiled modules

```
1     ...
2  is
3     Module : Compiler.Module_Type;
4  begin
5     Module := Compiler.Compile (Source => Src);
6     ...
```

JIT Workflow



Calling a function

```
1   ...
2   Func   : Compiler.Function_Type;
3   Module : Compiler.Module_Type;
4   begin
5   Module := Compiler.Compile (Source => Src);
6   Func   := Compiler.Get_Function
7     (Module => Module,
8      Name   => "add");
9
10  Func.Call
11    (Args =>
12       (1 => In_Arg (Data => A),
13        2 => In_Arg (Data => B),
14        3 => Out_Arg (Data => C'Access)));
```

Kernel arguments

Functionality

- Take care of device memory handling (allocation / freeing)
- Copy data from host to device and from device back to host
- Completely transparent to users of CUDA/Ada
- Implemented using Ada generics
- Three different argument types:
  - *In*
  - *Out*
  - *InOut*
- Similar to Ada's formal parameter modes (in, out, in out)

Kernel arguments II

```
1  generic
2     type Data_Type is private;
3  package Arg_Creators is
4     function In_Arg (Data : Data_Type) return
          Arg_Type;
5
6     function Out_Arg (Data : not null access
          Data_Type) return Arg_Type;
7
8     function In_Out_Arg (Data : not null access
          Data_Type) return Arg_Type;
9  end Arg_Creators;
```

Kernel arguments III

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

Kernel arguments IV

Kernel signature

```
1 void mul(float* A, float* B, float* C)
```

CUDA/Ada call

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

Error handling

Functionality

- Translation of CUDA errors to Ada exceptions
- Automated error checking of all low-level Thin-Binding calls
- Resolution of error code to error message

Requesting a nonexistent kernel 'Matrix_Mul'

```
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
```

Querying CUDA devices

Functionality

- Enumerate all available CUDA devices
- Query device properties like name, compute capability, etc.

```
1  with Ada.Text_IO;
2  with CUDA.Driver; use CUDA.Driver;
3
4  procedure Enum_Devices is
5     procedure Print_Name (Dev : Device_Type) is
6     begin
7        Ada.Text_IO.Put_Line (Name (Dev));
8     end Print_Name;
9  begin
10    Iterate (Process => Print_Name'Access);
11 end Enum_Devices;
```

Performance

Ambition

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

Measurement methodology

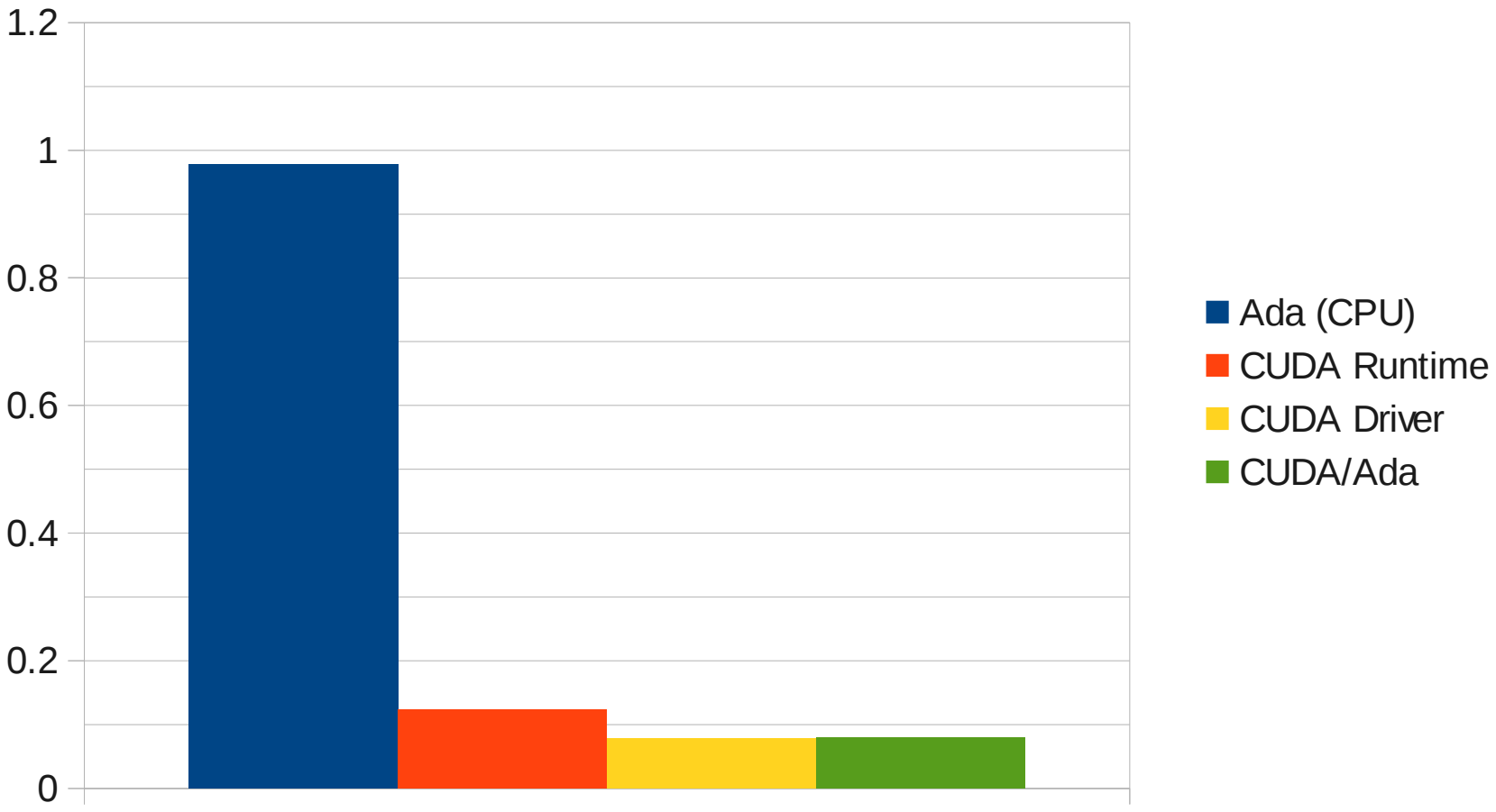
- Cumulated runtime of 20 matrix multiplications (512 x 512)
- Comparison of different implementations:
  - Ada (CPU)
  - CUDA Runtime API
  - CUDA Driver API
  - CUDA/Ada
- Same grid and block size for all CUDA implementations

Performance II

Processor	AMD Phenom II X4 940
Graphics Card	GeForce GTX 560 Ti
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: Test system specs

Benchmarking results



Performance III

Interpretation

- Native Ada (CPU) is slow: CPU vs. GPU/CUDA
- CUDA/Ada is faster than CUDA Runtime API:
  - CUDA Runtime API generates management as well as kernel launch code, etc.
  - CUDA/Ada performs bare minimum of management operations
- CUDA/Ada is negligibly slower than CUDA Driver API
- No visible performance penalty

Review

Assessment of results

- Thick-Binding provides easy usage of CUDA from Ada
- High level abstractions go well with other Ada language constructs
- Simple Autoinitialization of CUDA via `Autoinit` package
- JIT compilation of kernels provides flexibilty
- Speed of CUDA/Ada is excellent
- Automated Thin-Binding generation allows for easy integration of future CUDA API changes
- Paper documents how to write a modern, maintainable language binding for Ada in general

Review II

Deliverables

- Paper
- Slides
- Source code (GPLv3+)
  `http://git.codelabs.ch/?p=cuda-ada.git`
  `git clone http://git.codelabs.ch/git/cuda-ada.git`
- Website with all documents and additional information:
  `http://www.codelabs.ch/cuda-ada/`

Outlook

CUDA/Ada

- Implement additional features:
  - Abstractions for kernel generation (e.g. element-wise)
  - Kernel signature verification on `Call`
  - Multi-Device support
  - ...
- Announce project to Ada community

Ada and CUDA

- Study NVIDIA's LLVM-based CUDA compiler (announced, not yet released)

Questions

Thank you for your attention!