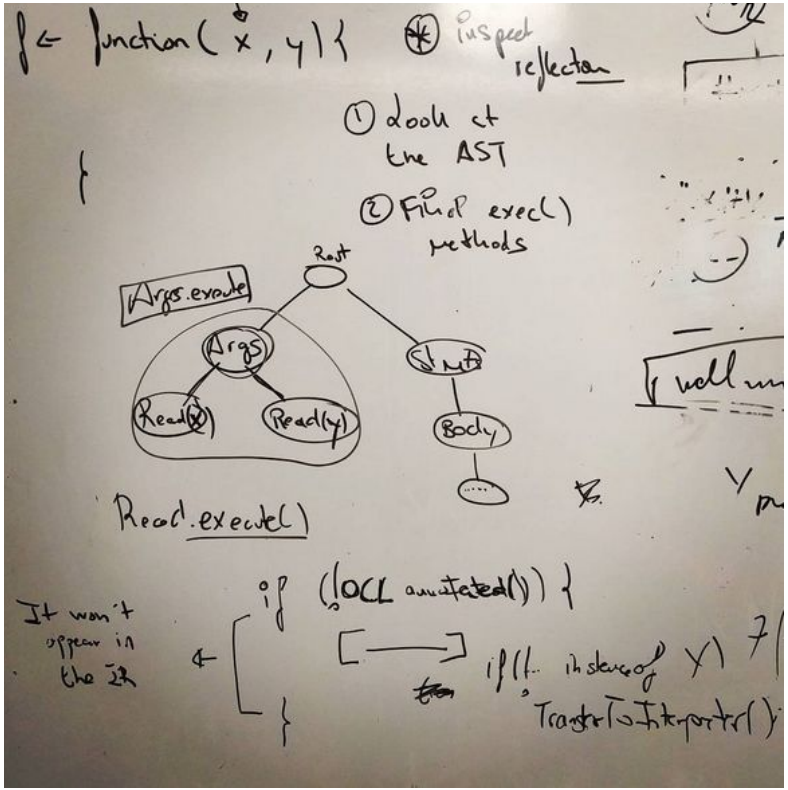
Automatic Acceleration of R programs with GPUs

15 February 2017

Edinburgh R Users Group

Juan Fumero <juan.fumero@ed.ac.uk>

Disclaimer: this is a research prototype

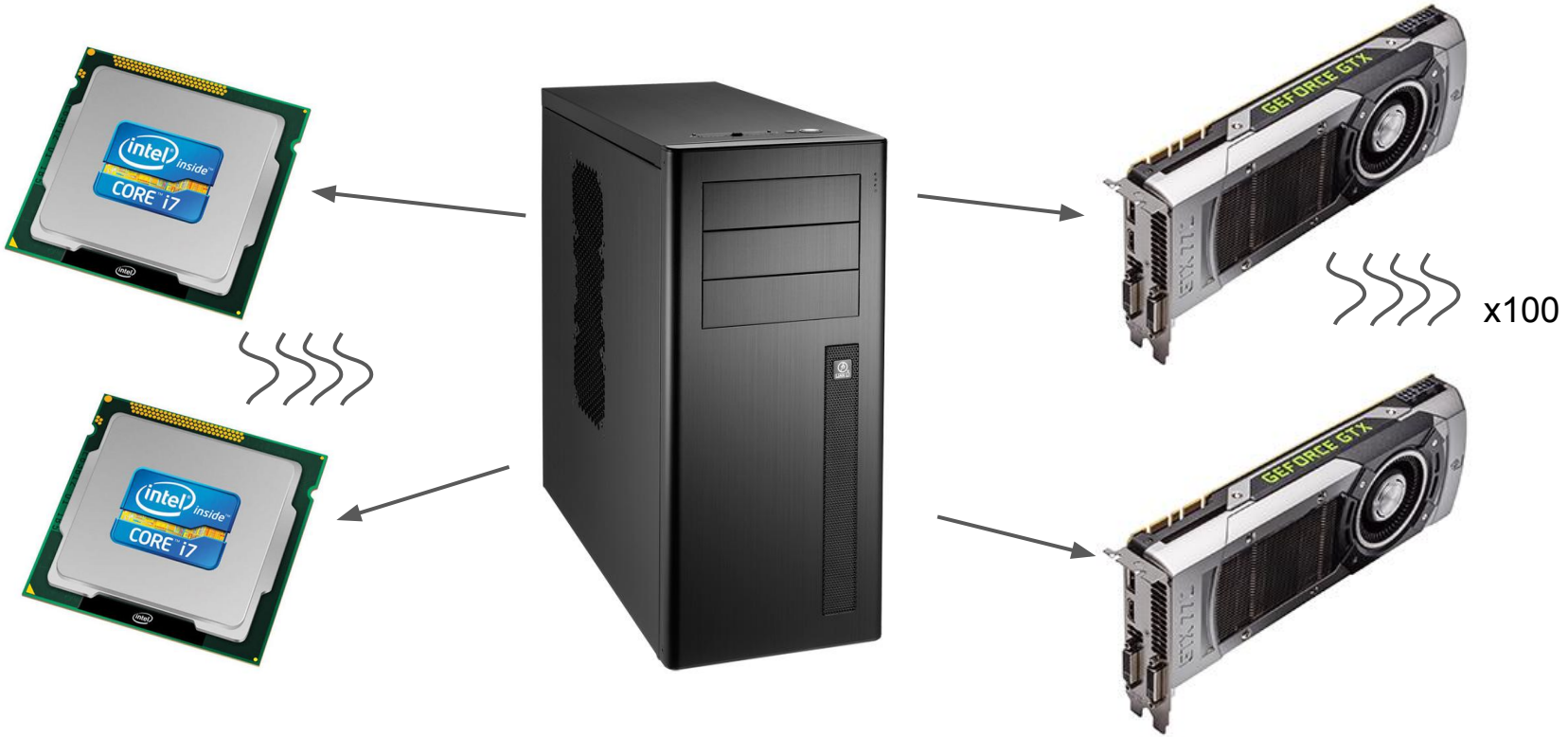


Outline

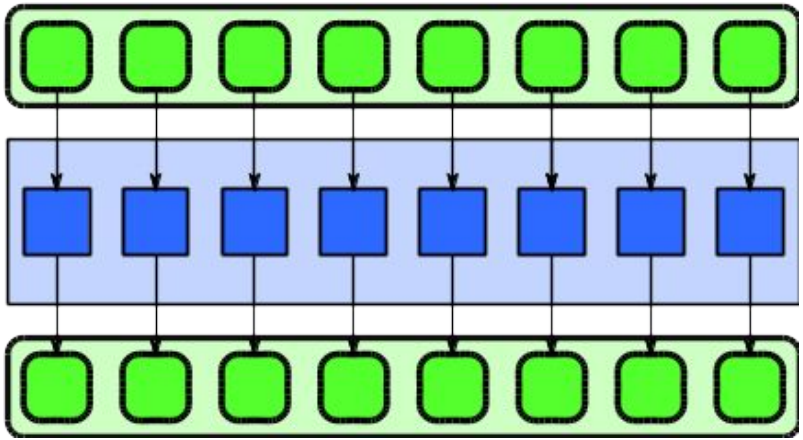
1. Introduction to GPUs and Motivation
2. R and GPU Programming
3. Our Compiler Approach: Automatic GPU programming with R
4. Demo
5. Some results

Introduction to GPUs and motivation

Heterogeneous Systems



GPUs: What are they good for? → Map/Apply computation



- No data dependency between iterations
- Best case for parallelization

```
for (int i = 0; i < n; i++) {

 output[i] = function(input[i]);

}
```

- Examples:
  - NBody,
  - Montecarlo
  - Black-Scholes
  - KMeans
  - Ray-Tracing, Image Processing, …

Example - NBody Simulation

$$\mathbf{a}_i \approx G \cdot \sum_{1 \le j \le N} \frac{m_j \mathbf{r}_{ij}}{\left(\|\mathbf{r}_{ij}\|^2 + \varepsilon^2 \right)^{3/2}}.$$

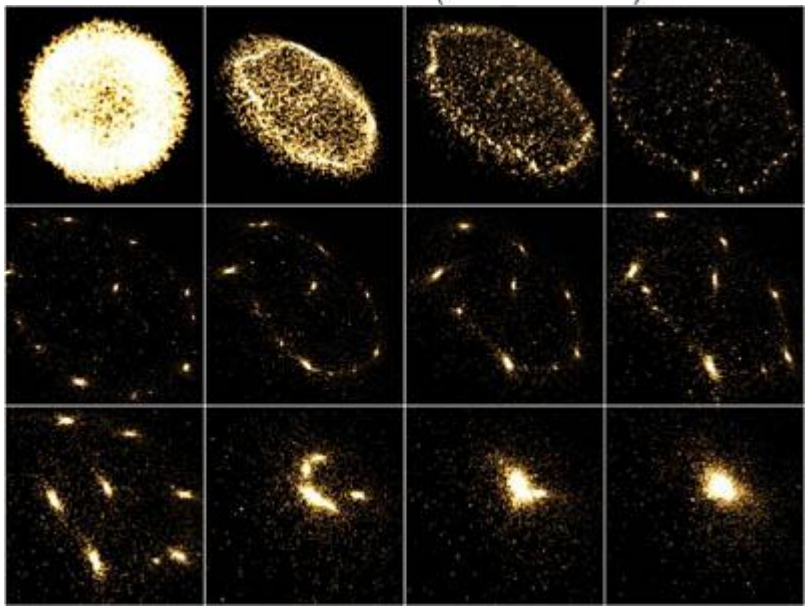
- NVIDIA GPU Tesla C1060 (240 cores)
- Sequential code (Intel Core i7)
- 400k particles

Seq: 41 hours

OpenMP: 11 hours

GPU code: 23 minutes

105x



// Source: Nvidia SDK

Cool, but how to program?

OpenCL

C extensions

R and GPU Programming

How to program GPUs from R? Ruby? Python?

- Via wrappers to C code. → You have to change your program and mix langs.

```
library(OpenCL)
p = oclPlatforms()
d = oclDevices(p[[1]])

code <- c("
  #pragma OPENCL EXTENSION cl_khr_fp64 : enable
  __kernel void gpu_sum(
    // first two args are mandatory
    __global double* output,
    const unsigned int n,
    // user args
    __global const double* input1,
    __global const double* input2
  ) {
     int i = get_global_id(0);
     if (i<n) output[i] = input1[i] + input2[i];
  };
")
k.gpu.sum <- oclSimpleKernel(d[[1]], "gpu_sum", code, "double")
vector2 <- as.double(2:11)
result <- oclRun(k.gpu.sum, length(vector1), vector1, vector2)
print(result)
```

- Via external modules or libraries → You have to change your program and limited

```
# R example

library(gpuR)

ORDER = 1024

A = matrix(rnorm(ORDER^2), nrow=ORDER)
B = matrix(rnorm(ORDER^2), nrow=ORDER)
gpuA = gpuMatrix(A, type="double")
gpuB = gpuMatrix(B, type="double")

C = A %*% B
gpuC = gpuA %*% gpuB

all.equal(C == gpuC[])
```

R is an Interpreted Language. What does it mean?

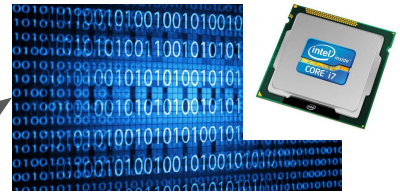
The interpreter translates each line of the source code into binary code

```
# R program

workshop <- c(1, 2, 1, 2, 1, 2)

par(mfrow = c(2,1))  # set up the graphics
plot(soi, ylab="", xlab="", main="Southern Oscillation Index")
plot(rec, ylab="", xlab="", main="Recruitment")
```

R Interpreter



They are very slow!

How to Speed-up Interpreters?

- Jut-In-Time (JIT) Compilation: When the application is running, it compiles the most expensive parts of the input program

- If a function is executed multiple times, the whole function is compiled to binary code.

What we do is JIT compilation for GPUs instead of CPU!

Our solution: Automatic GPU programming with R

My proposal

Offload any arbitrary R code to the GPU automatically for apply functions

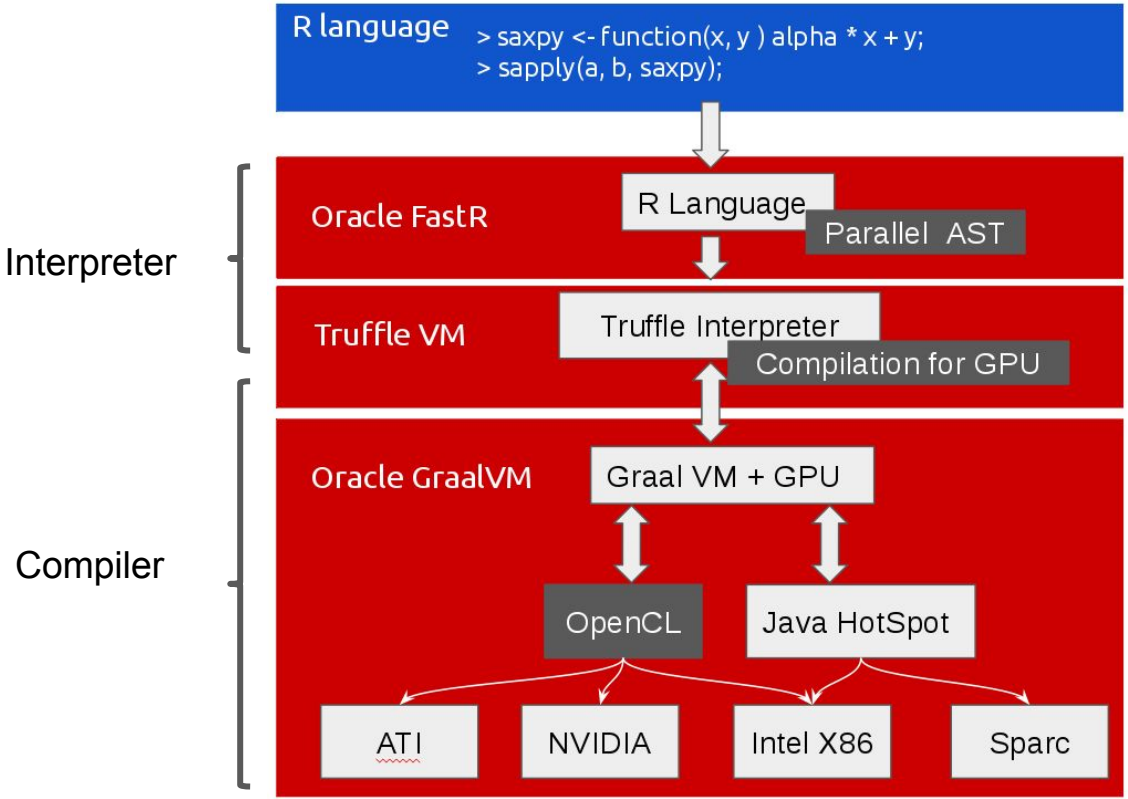
- No changes in the source code
- Achieve as good performance as C

Our idea: get the existing functions in the language that can be easily parallelize:

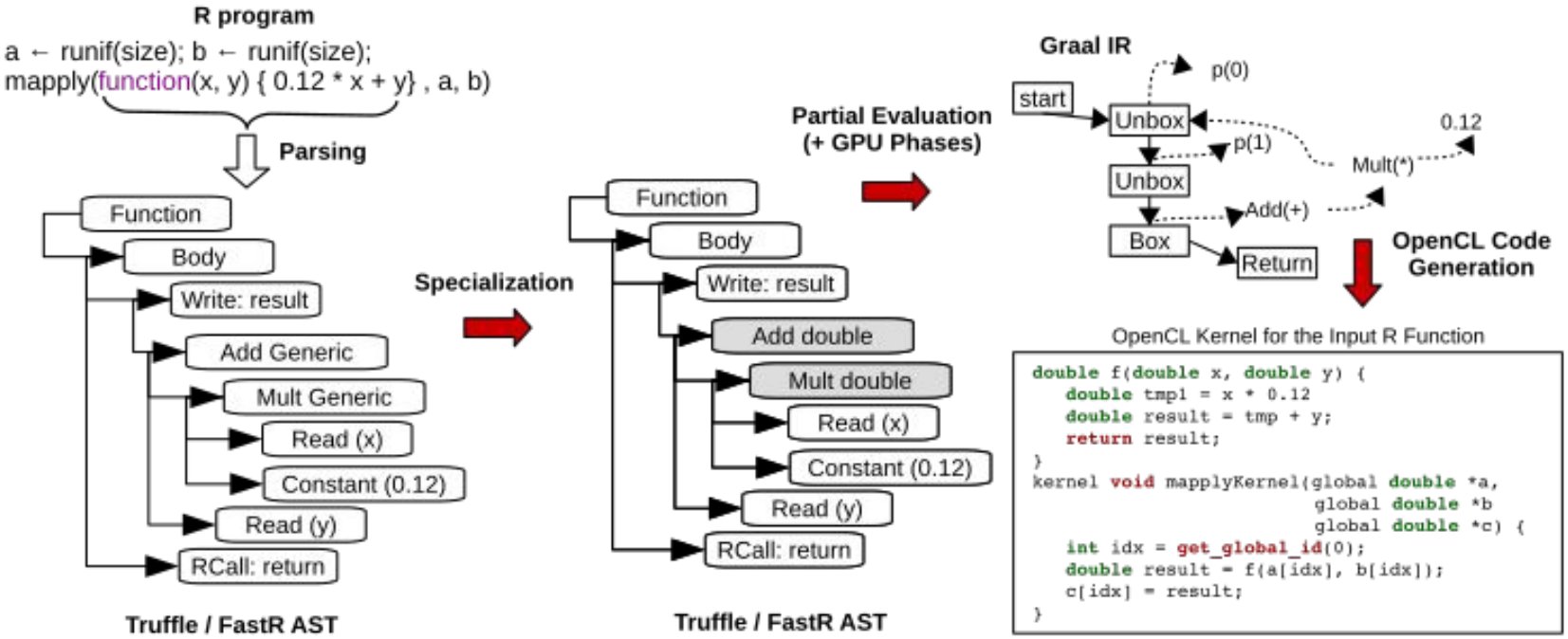
apply(<userFunction>, data)

The <userFunction> will be executed on GPU

Compiler Infrastructure



A little bit more in detail



Example

Vector operations (artificial to show how it works)

```
saxpyFunction <- function(x, y) {
  result <- (0.12 * x) + y;
        return (result);
}

x <- runif(size)
y <- runif(size)

result <- mapply(saxpyFunction, x, y);
```

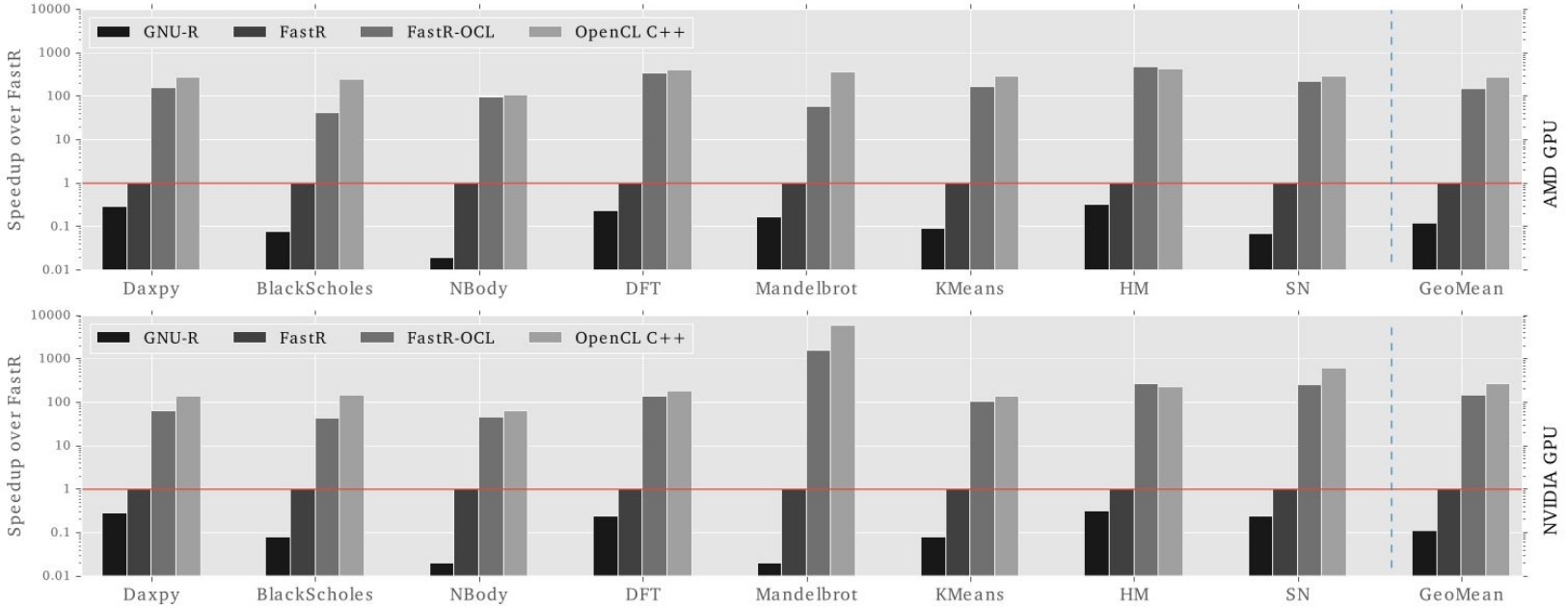
Demo

Demo

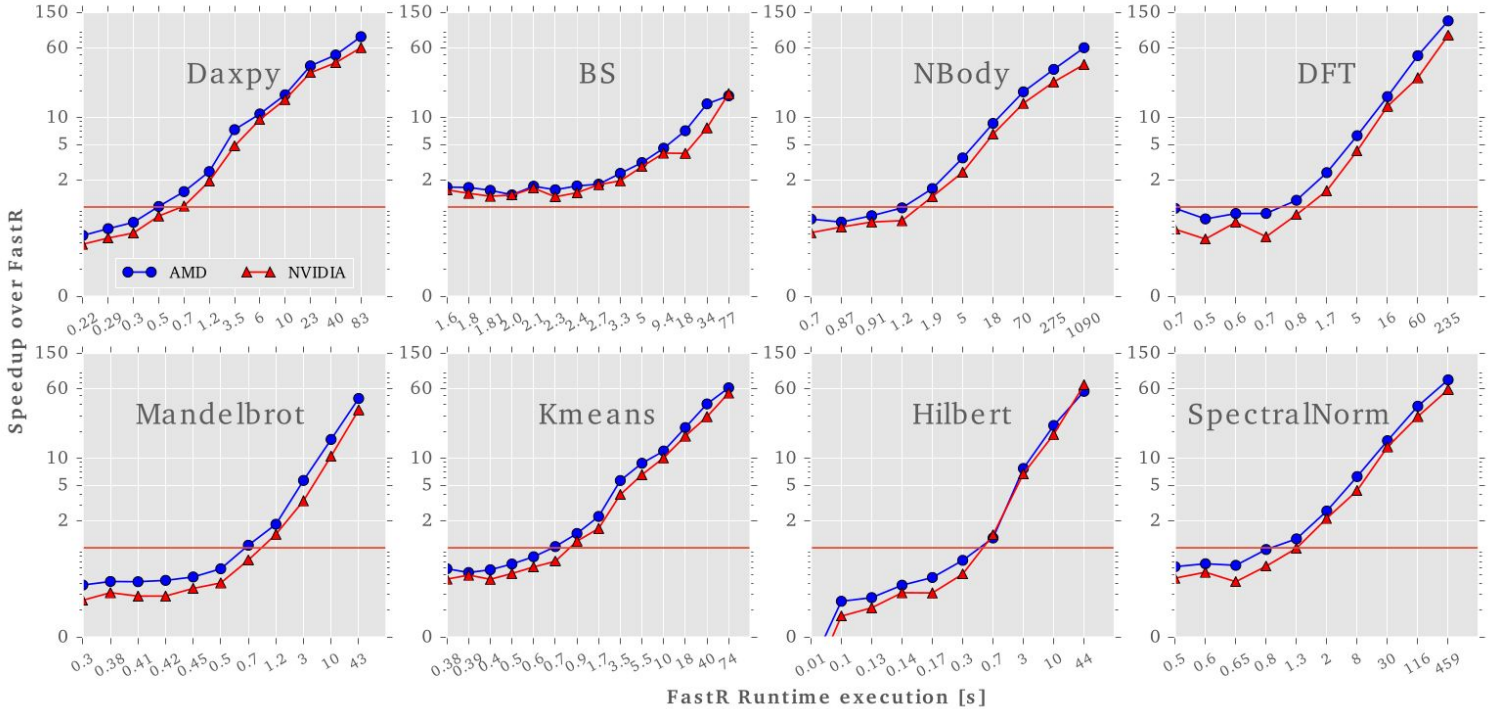
1. Vector operations: how the GPU code is generated
2. Data optimizations
3. Bigger experient- nBody!!!! Comparison with R!

More serious results

Results


GNU-R
FastR
FastR-OCL
OpenCL C++
Speedup over FastR
10000
1000
100
10
1
0.1
0.01
Daxpy
BlackScholes
NBody
DFT
Mandelbrot
KMeans
HM
SN
GeoMean
AMD GPU
NVIDIA GPU

Results: fresh execution



More Info

For more details we have a paper

in VEE 2017!

VEE'17 preprint

Just-In-Time GPU Compilation for Interpreted Languages with Partial Evaluation

Juan Fumero[†] Michel Steuwer[†] Lukas Stadler[*] Christophe Dubach[†]

[†]The University of Edinburgh, [*]Oracle Labs

juan.fumero@ed.ac.uk michel.steuwer@ed.ac.uk lukas.stadler@oracle.com christophe.dubach@ed.ac.uk

Abstract

Computer systems are increasingly featuring powerful parallel devices with the advent of many-core CPUs, GPUs and FPGAs. This offers the opportunity to solve large computationally-intensive problems at a fraction of the time traditional CPUs need. However, exploiting this heterogeneous hardware requires the use of low-level programming languages such as OpenCL, which is incredibly challenging, even for advanced programmers.

On the application side, interpreted dynamic languages are increasingly becoming popular in many emerging domains due to their simplicity, expressiveness and flexibility. However, this creates a wide gap between the nice high-level abstractions offered to non-expert programmers and the low-level hardware-specific interface. Currently, programmers have to rely on specialized high performance libraries or are forced to write parts of their application in a low-level language like OpenCL. Ideally, non-expert programmers should be able to exploit heterogeneous hardware directly from their interpreted dynamic languages.

In this paper, we present a technique to transparently and automatically offload computations from interpreted dynamic languages to heterogeneous devices. Using just-in-time compilation, we automatically generate OpenCL code at runtime which is specialized to the actual observed data types using profiling information. We demonstrate our tech- account start-up time, large speedups are achievable, even when the applications run for as little as a few seconds.

1. Introduction

Nowadays, most computer systems are equipped with powerful parallel devices such as Graphics Processing Units (GPUs). Many applications domains can benefit from these devices, often achieving orders of magnitude speedup over parallel CPU code. However, exploiting this hardware requires a deep knowledge of the architectures and low-level languages such as OpenCL, CUDA or C. This is a very challenging task for non-expert programmers.

Many non-computer scientists prefer using interpreted languages such as Ruby, Python or R which are hugely popular despite their poor performance. They offer high-level functionality and simplicity of use, and the interpreter enables fast iterative software development. However, exploiting a GPU from these languages is far from trivial since programmers either have to write the GPU kernels themselves or rely on third-party GPU accelerated libraries.

Ideally, an interpreter for a dynamic programming language would be able to exploit the GPU automatically and transparently. A possible solution is to port the interpreter to the GPU and directly interpret the input program on the GPU. Unfortunately, this naïve solution is not practical since many parts of the interpreter are hard to port to a GPU such

Thanks so much for your attention

This work is supported by a grant from:

Oracle Labs

Juan Fumero <juan.fumero@ed.ac.uk>