# GPU Linear algebra extensions for GNU/Octave

Attilio Santocchia INFN and Physics Department Perugia

Leone Bosi Physics Department Camerino

Mirko Mariotti Physics Department Perugia

ACAT 2011 – September 5-9, 2011, Uxbridge, London

14th International Workshop on Advanced Computing and Analysis Techniques in Physics Research

## GNU Octave

GNU Octave is a high-level interpreted language similar to Matlab and intended for numerical computations.

It provides powerful matrix manipulation, numerical problems solver and simulation tools.

It also provides extensive graphics capabilities for data visualization and manipulation.

## OpenCL

OpenCL is a framework for writing programs for heterogeneous platforms.

In OpenCL there are:
- kernels: function that executes on a device.
- control program: a standard program that supervise the kernels execution.

Modern GPU have several processors capable of running OpenCL kernels very fast.

OpenCL is a open standard from Khronos group.

## Octave + OpenCL

Our goal is to bring the power of OpenCL within GNU Octave.

When operating on big matrices a GPU device may have much better performance than a traditional CPU.

In order to do this we have to learn first how to use Octave and GPU together

## Extending Octave: OCT files

- It's the standard way to extend octave adding new commands
- These are essentially runtime loadable shared object
- The utility 'mkoctfile' is used to create custom OCT files.
- The octave function entry point is declared with the DEFUN_DLD macro

## OCT example file

**helloworld.cc**

```
#include <octave/oct.h>

DEFUN_DLD (helloworld, args, nargout, "Hello World Help String")

{

int nargin = args.length ();

octave_stdout << "Hello World has " << nargin << " input arguments
    and " << nargout << " output arguments.\n";

return octave_value_list ();

}
```

```
test# mkoctfile helloworld.cc
test# octave
octave:1> helloworld
Hello World has 0 input arguments and 0 output arguments
```

## First implementation: *vector_add* on GPU

- To use the GPU computing on an octave *vector_add* command we need to build an OCT file that:
  - Copy vectors data from octave objects to GPU memory
  - Create the OpenCL *vector_add* kernel and lauch it upon the previous data
  - Copy the result back to the octave result object

# Copying vector data

```
DEFUN_DLD(vector_add, args, , "GPU sum of 2 vectors")
{
  octave_value_list retval;
  int nargin = args.length ();

...

  NDArray input_A = args(0).array_value ();
  NDArray input_B = args(1).array_value ();

...

  const int LIST_SIZE = input_A.nelem () ;

  // Create the two input vectors
  octave_idx_type i;
  int *A = (int*)malloc(sizeof(int)*LIST_SIZE);
  int *B = (int*)malloc(sizeof(int)*LIST_SIZE);

  for(i = 0; i < LIST_SIZE; i++)
  {
    A[i] = input_A.elem (i);
    B[i] = input_B.elem (i);
  }
```

•Vector data is extracted from the octave function arguments  and stored within the host memory to the array A and B

Note: This procedure implies a double bufferization, that introduce a not negligeble overhead.

•After the GPU has made its work, the same is done the get back the results from the array C

```
// Creating the c[illegible]e array and fill it with results
NDArray output[illegible]put_A);

for(i = 0; i < LI[illegible]E; i++)
{
  output_C.elem (i)=C[i];
}
```

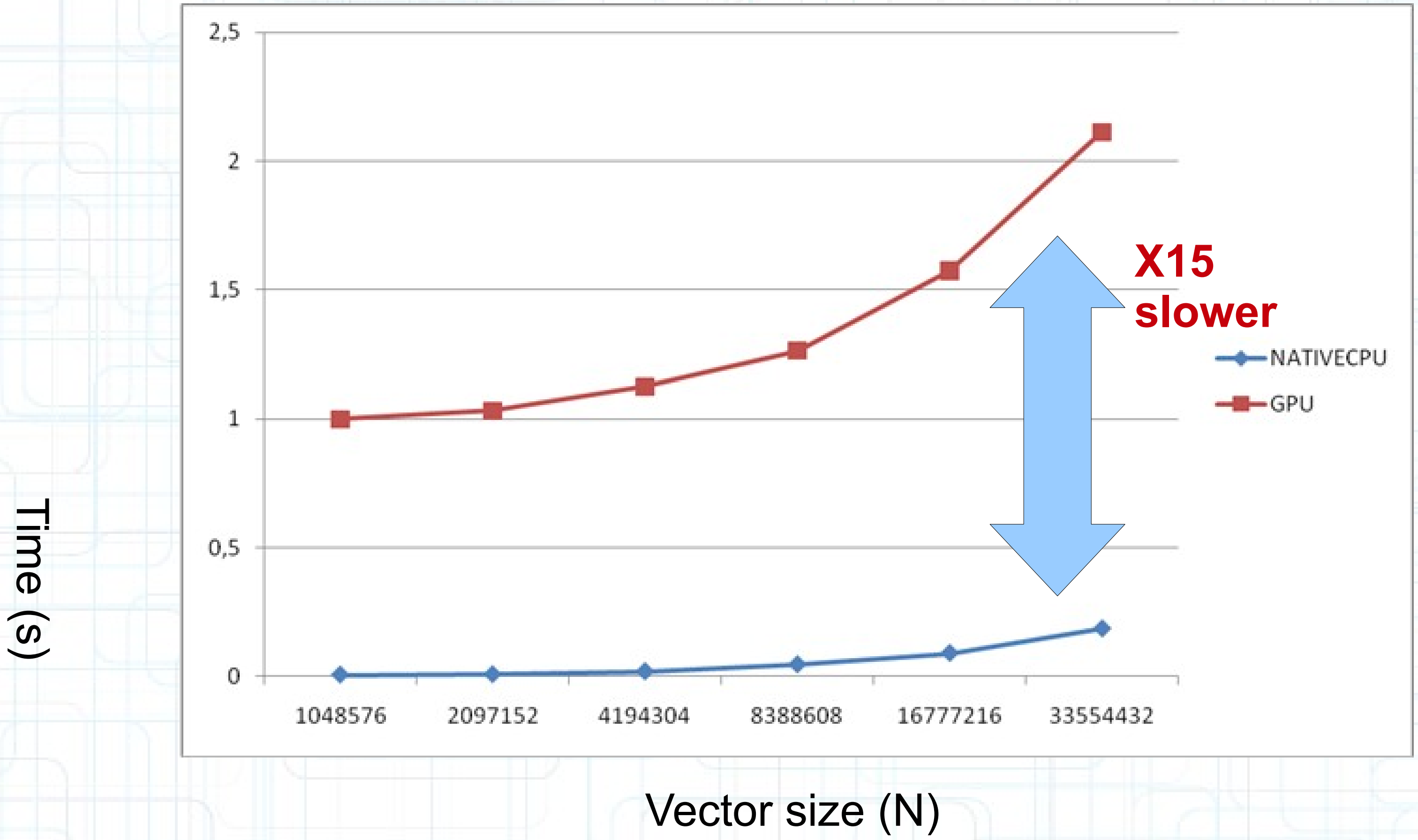
## *vector_add*
## The first benchmarks

We wrote a simple benchmark under Octave.

The code loops several times the *vector_add* command over different length vectors.
- The length is changed in order to test the code increasing computational complexity. The GPU is expected to work best under high computing density.

We started comparing:
- <u>Octave native vector sum</u>: this use the native octave "+"
- *<u>vector_add</u>*: this is the first GPU implementation

## Native vs GPU benchmark



Hardware:
- CPU - Intel Xeon E5520 2.27GHz
- GPU - Nvidia Tesla C1060

## First analysis

With the zero level implementation we show the octave-GPU link feasibility but also a first evident criticality.

The benchmark reports that the our first GPU code runs slower then the native one

**It is clear that this is not the solution we are looking for...**

Several overhead sources are present. At first order:
•The OpenCL initialization and kernel compilation occur every time an operation is performed.
•The Vector data are copied every time from/to host/host and from/to the GPU/host memory (double buffering).

## Code profiling

In order to find out bottlenecks and sources of systematic overheads we profiled our code.

profile OCT files is not straightforward but it needs some trick. We follow two methods:

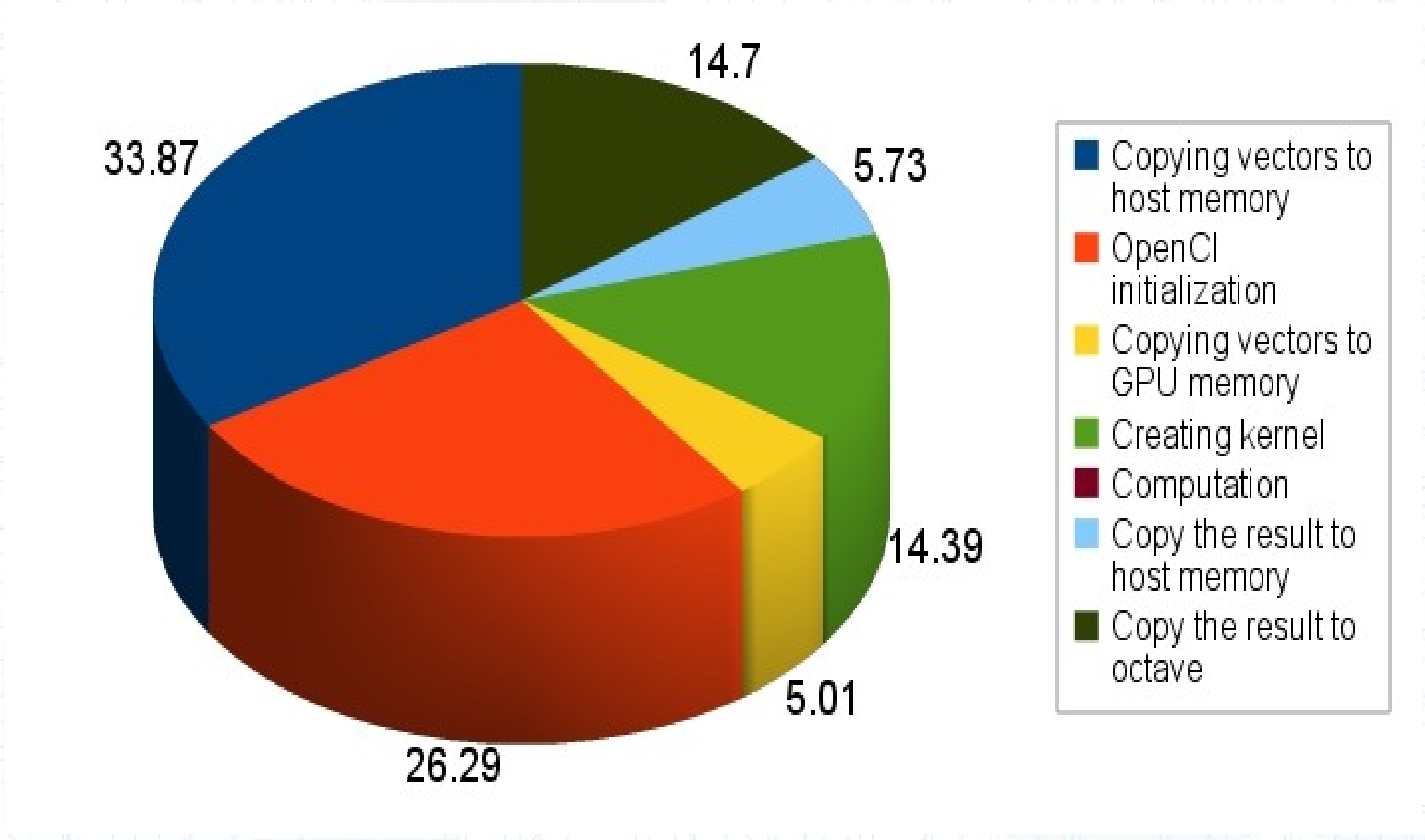
Method 1

-Instead of creating an OCT file we use the octave interpreter within our program.
-We build statically our programs with both native octave computation and GPU computation.
-So we may profile and benchmarking the code with more efficiency

Method 2

-Using inline gettimeofday calls

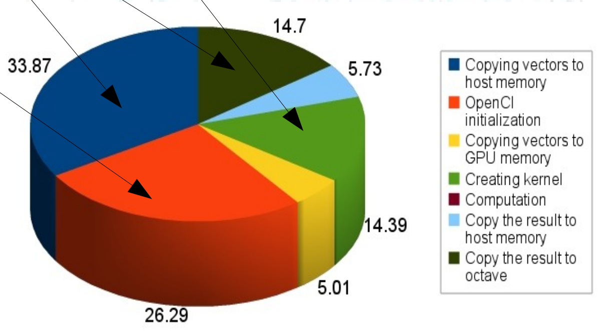
## Profiling Vector add



## Code optimization

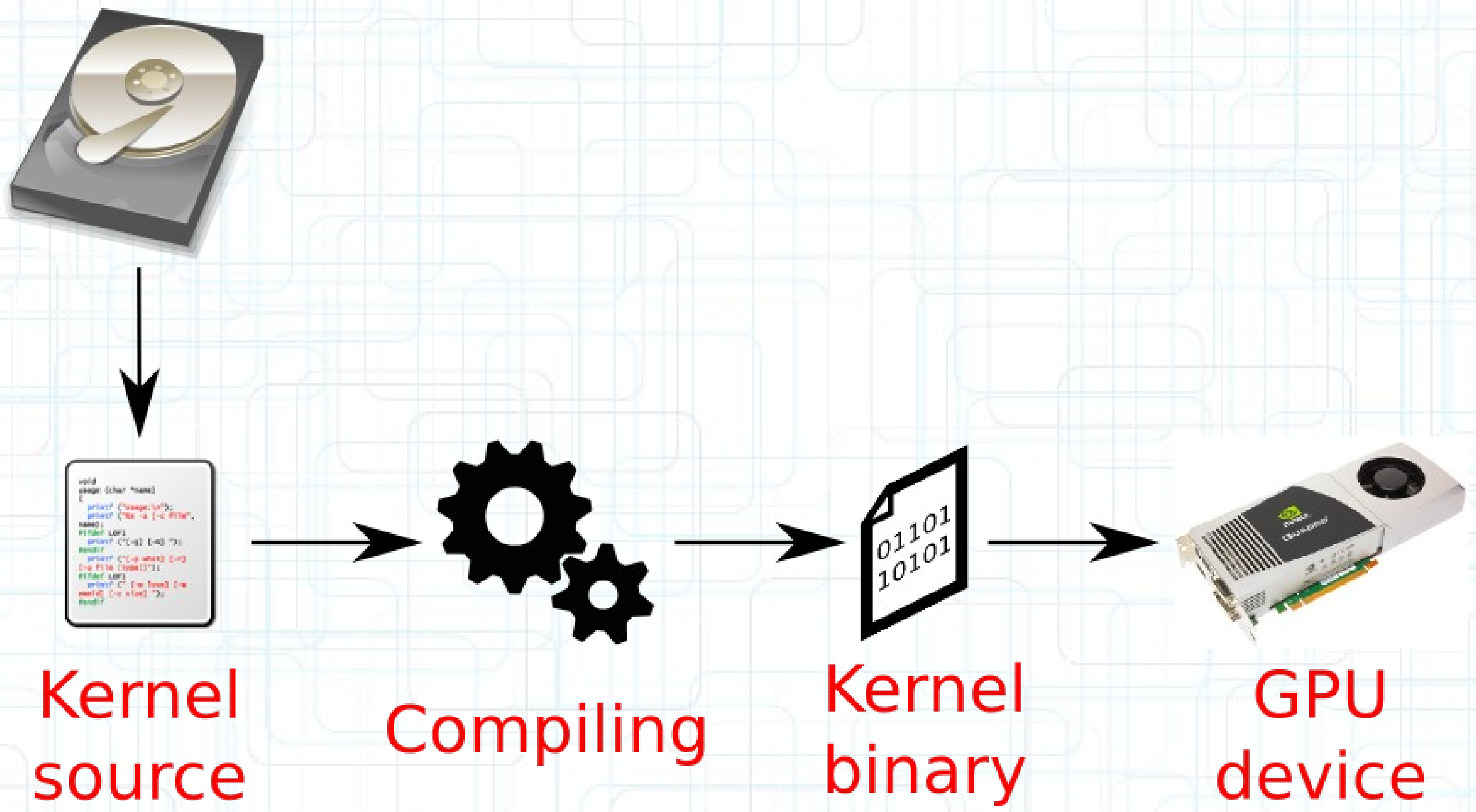
The profiling reports the IO overheads.

The code section to be optimized are:
- Kernel just in time compilation
- Octave memory objects to host buffers RW
- OpenCL initialization



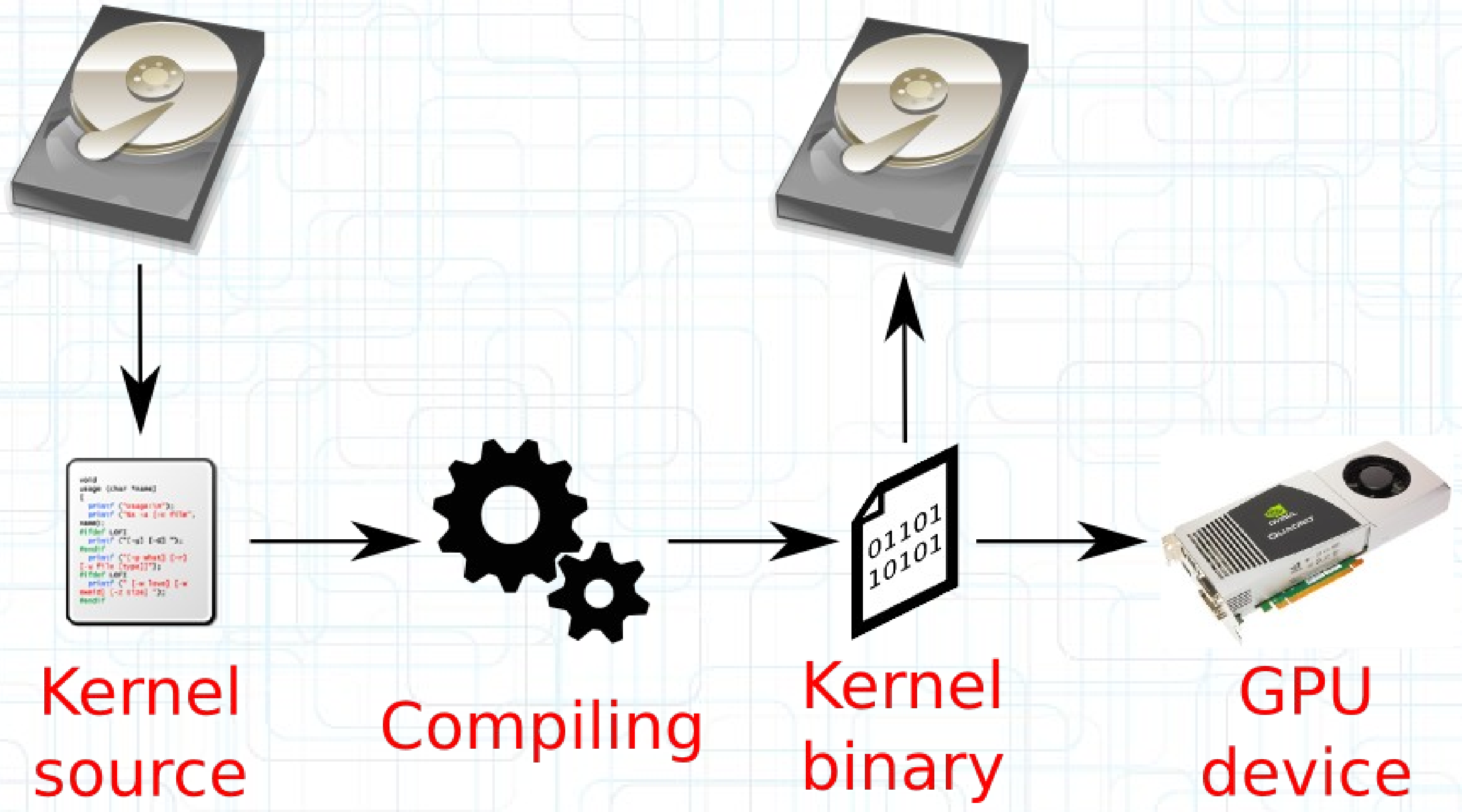
Solutions:
- Precompiled kernels
- Direct access to octave memory objects
- OpenCL context, kernel caching, host/GPU memory IO on demand: defining a new octave GPU object

# Code optimization: Precompiled kernels



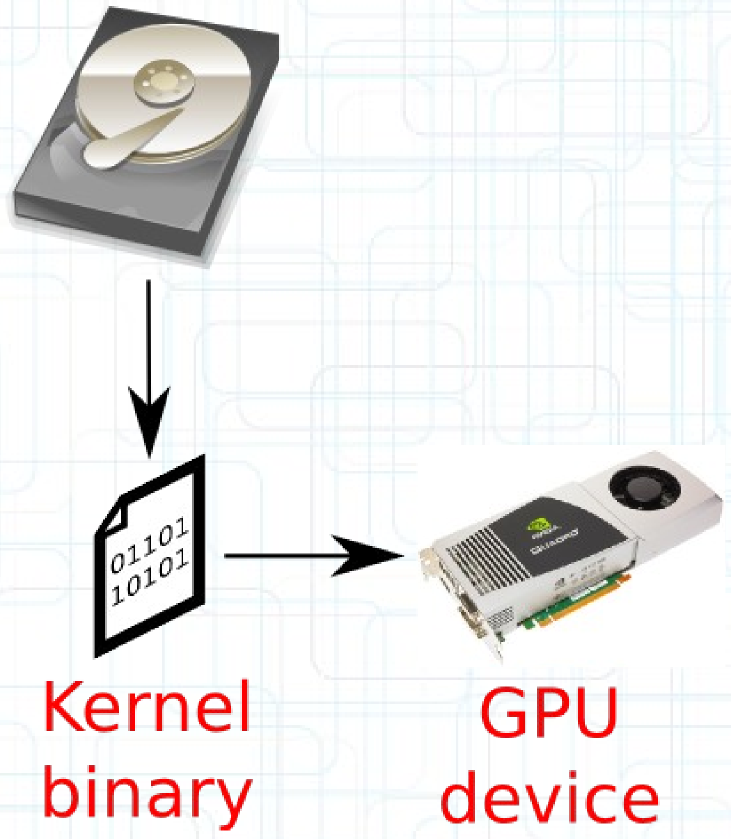
Usually in OpenCL the kernels are compiled at run-time

## Code optimization: Precompiled kernels



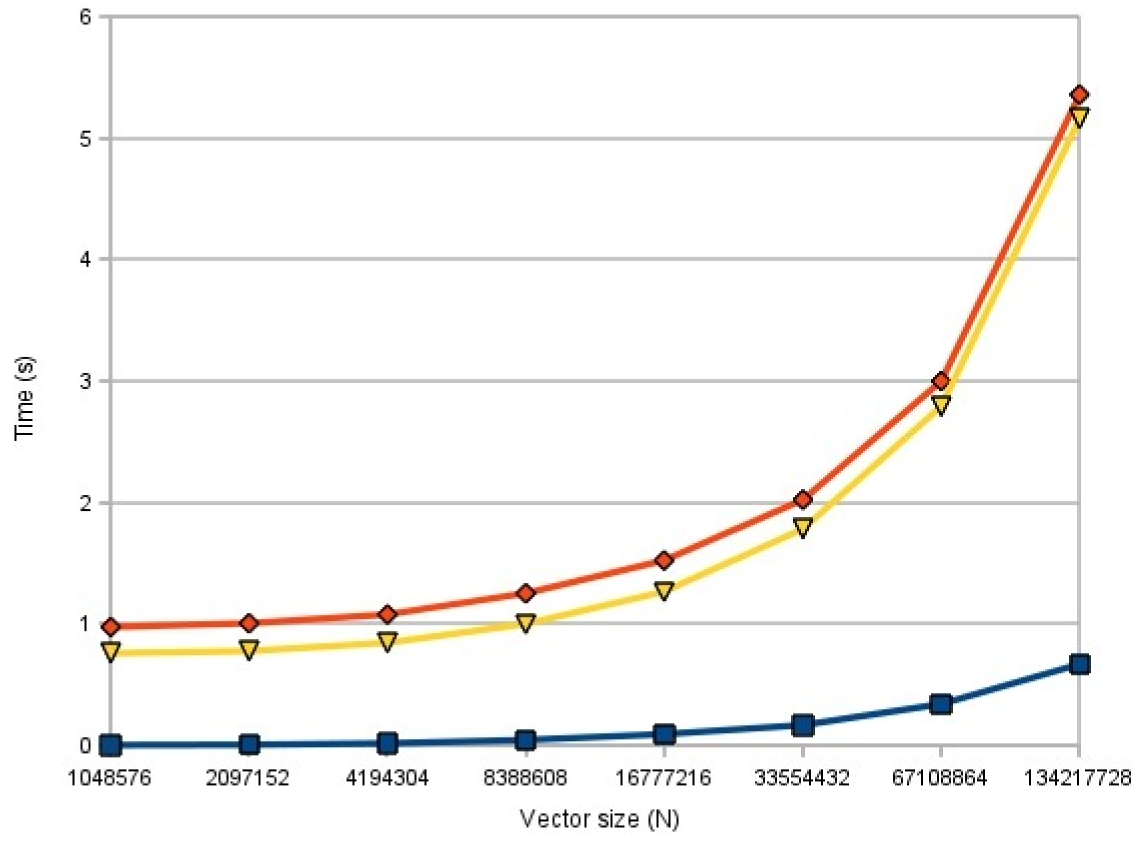
We build binary GPU kernels during the first execution.

# Code optimization: Precompiled kernels

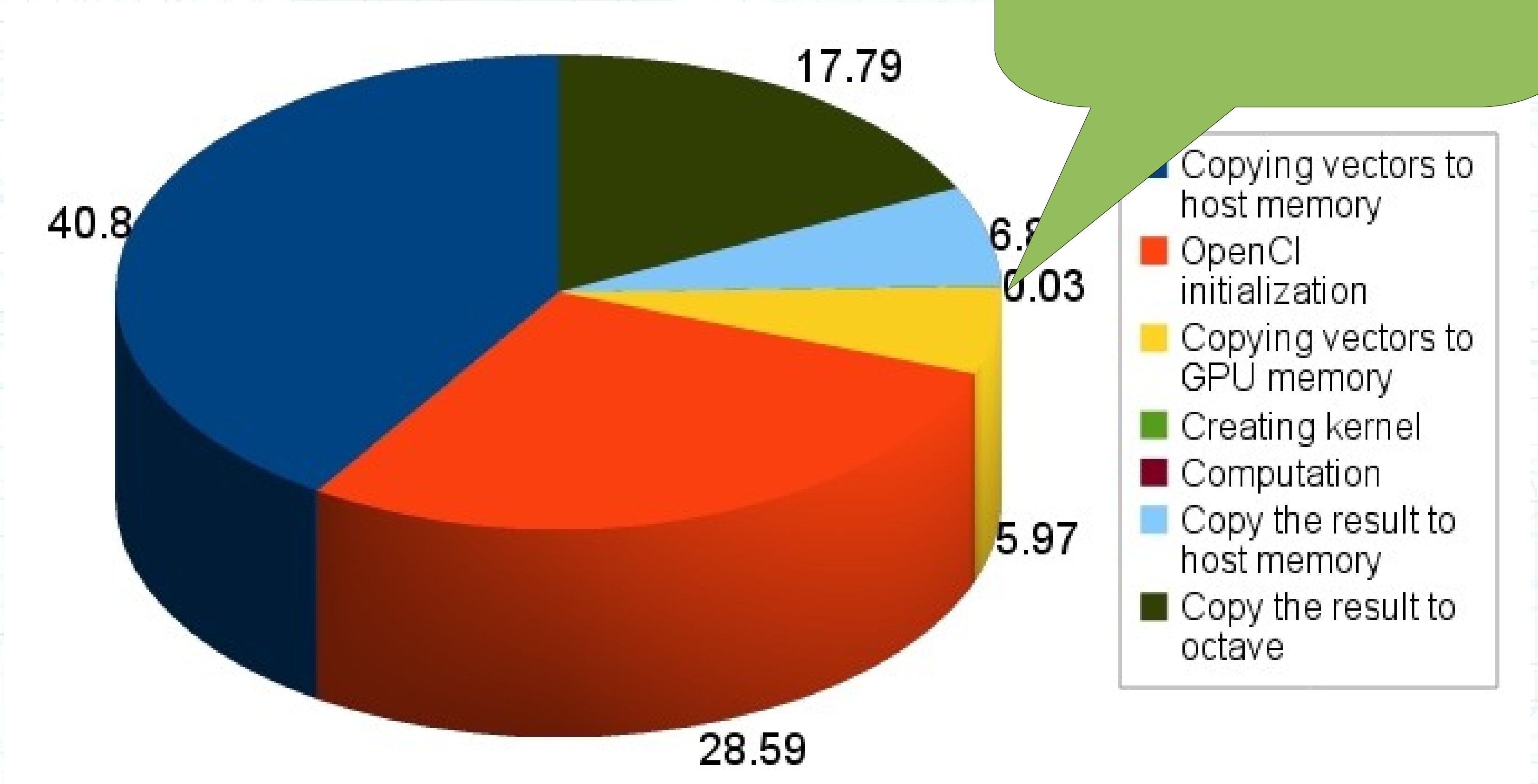


Further executions use the compiled kernels

# “Vector add” Precompiled kernel binaries The benchmark



## Precompiled kernel "Vector add" profiling



Creating kernel time << 1%

## What now?

GPU computing is a winner for hard numerical computation

Bottleneck is clearly IO and double-buffering operation:

Octave → host → GPU → host → Octave

Let's have a look at the real advantage of GPU computing: numerical operation

## *vector_add* static

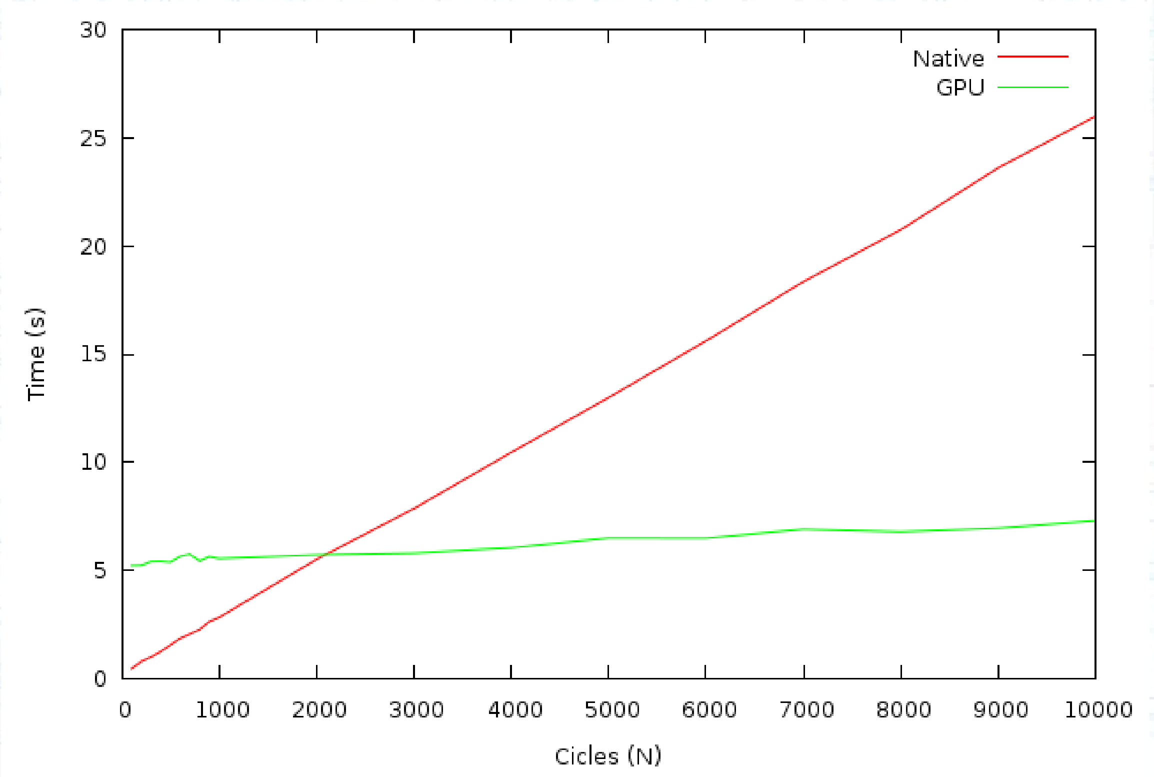
In order to better understand internal behavior of Octave we developed a static version of our *vector_add* command.

This version call the Octave library and is not usable from the interpreter.

Using it we compared the same computation made with the octave native sum and the OpenCL one...

...summing 1M size vectors several times...

# “Vector add” static: the benchmark



- Benchmarks highlight the offset of 5 seconds (overhead due to double-buffering)
- GPU performance win on host processing increasing the computing density.

# Conclusions

We implemented a simple Octave vector function using GPU devices, using OpenCL.

It's clear we need to remove all possible I/O data transfer overheads: OpenCL init, Kernel handling and GPU/host memory copy.

Precompiled kernel has been already implemented

Next step:
-Access directly octave objects data avoiding multiple bufferization
-define a specific GPU-Octave object, introducing memory data caching, host/GPU copy on demand, pure multiple operation on GPU  and initialization procedure.
-Operator overloading in order to keep the native operation symbolism such as “+”, “-”,…

## Acknowledgment

This work has been financed by “Centro Competenze Open Source della Regione Umbria”. Progetto n.79/10

This work has been developed in collaboration with the MaCGo project of the INFN Perugia Section

This work has been developed in collaboration with the Perugia Physics Department Computing Team

# Launching the kernel

Once the array data is stored on the host we may lauch the OpenCL kernel the usual way.

Vector add OpenCL kernel:

```
__kernel void vector_add(__global const int *A, __global const int *B, __global int *C) {
    int i = get_global_id(0);
    C[i] = A[i] + B[i];
}
```

```
fp = fopen("vector_add_kernel.cl", "r");
...
cl_context context = clCreateContext( NULL, 1, &device_id, NULL, NULL, &ret);
...
cl_program program = clCreateProgramWithSource(context, 1, (const char **)&source_str, (const
size_t *)&source_size, &ret);
...
ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);
cl_kernel kernel = clCreateKernel(program, "vector_add", &ret);
...
ret = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL, &global_item_size,
&local_item_size, 0, NULL, NULL);
```