StarPU's C Extensions for Hybrid CPU/GPU Task Programming

an experience in turning a clumsy API into language extensions

Ludovic Courtès

GNU Tools Cauldron, Prague, 9–11 July 2012

hello, cauldron!

hello, cauldron!

hello, cauldron!

Runtime research team*

`http://runtime.bordeaux.inria.fr/`

*joint team with Université de Bordeaux & LaBRI

hello, cauldron!

(guile)

hello, cauldron!

not in this talk!

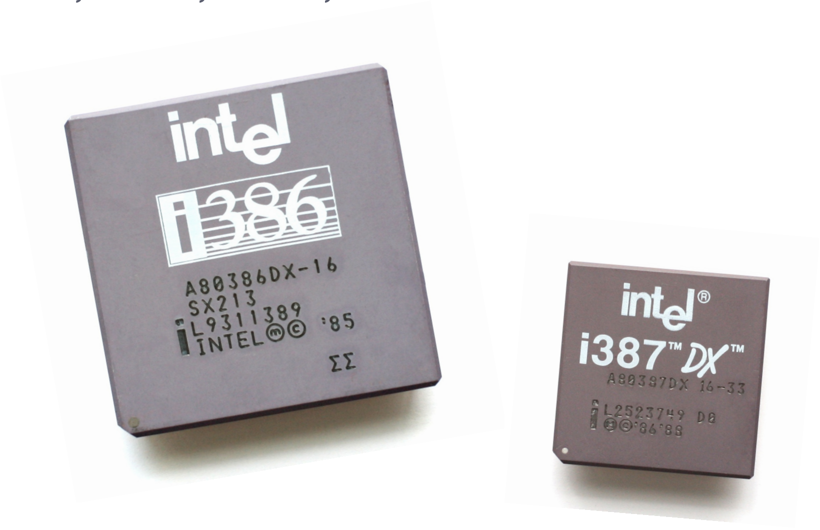
1. rationale
2. enter StarPU
3. the case for compiler support
4. on GCC extensions

1

rationale

HPC, CPUs, GPUs, and all that

HPC, CPUs, GPUs, and all that



HPC, CPUs, GPUs, and all that

HPC, CPUs, GPUs, and all that

HPC, CPUs, GPUs, and all that

HPC, CPUs, GPUs, and all that

HPC, CPUs, GPUs, and all that

wooooow, megaflops!

HPC, CPUs, GPUs, and all that

woooow, megaflops!

hmm, heterogeneity is upon us

HPC, CPUs, GPUs, and all that

wooooow, megaflops!

hmm, heterogeneity is upon us

damn, how do i program that?!

short-sightedness in the multicore + GPU era

1.1 Scope

This OpenACC API document covers only user-directed accelerator programming, where the user specifies the regions of a host program to be targeted for offloading to an accelerator device. The remainder of the program will be executed on the host. This document does not describe features or limitations of the host programming environment as a whole; it is limited to specification of loops and regions of code to be offloaded to an accelerator.

This document does not describe automatic detection and offloading of regions of code to an accelerator by a compiler or other tool. This document does not describe targeting loops or code regions to multiple accelerators attached to a single host. While future compilers may allow for automatic offloading, multiple accelerators of the same type, or multiple accelerators of different types, none of these features are addressed in this document.

short-sightedness in the multicore + GPU era

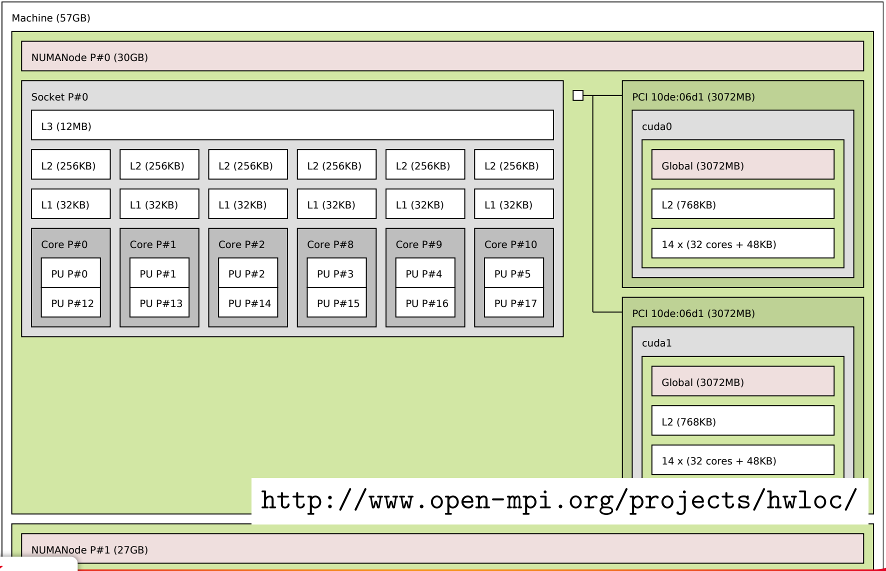
1.1 Scope

This OpenACC API document covers only user-directed accelerator programming, where the user specifies the regions of a host program to be targeted for offloading to an accelerator device. The remainder of the program will be executed on the host. This document does not describe features or limitations of the host programming environment as a whole; it is limited to specification of loops and regions of code to be offloaded to an accelerator.

This ... allow for auto... accelerators of different types, ...

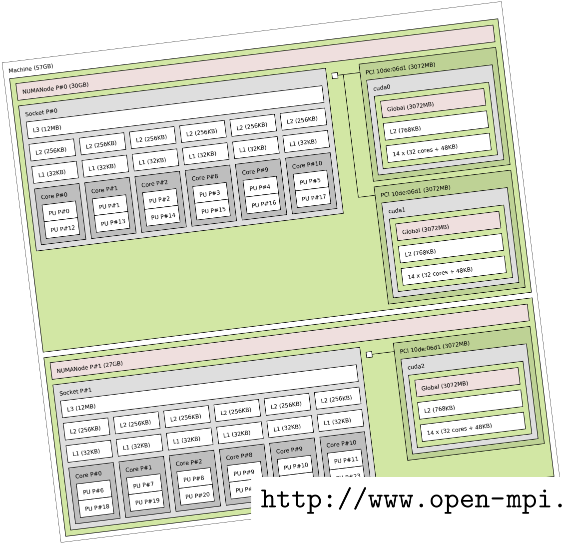
> “While future compilers may allow for [...] **multiple accelerators** of the same type, or multiple accelerators of different types, **none of these features** are addressed in this document.”

what today's machines really look like



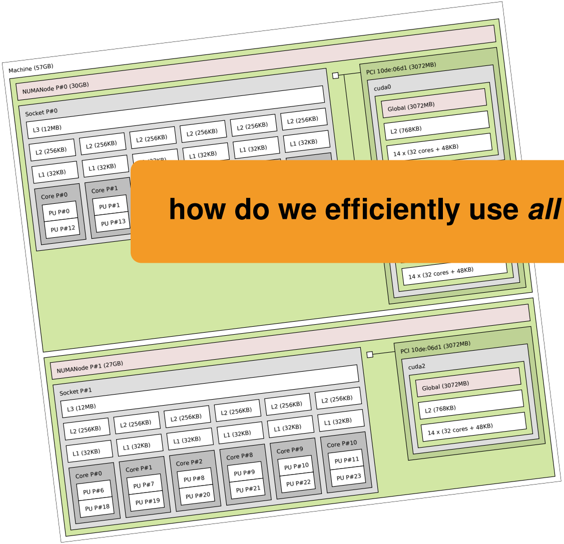
http://www.open-mpi.org/projects/hwloc/

what today's machines really look like



http://www.open-mpi.org/projects/hwloc/

what today's machines really look like



how do we efficiently use *all* these PUs?

short-sightedness in the multicore + GPU era

```
clGetDeviceIDs (NULL, CL_DEVICE_TYPE_DEFAULT, 1,
                &device_id, NULL);
queue = clCreateCommandQueue (context,  device_id , 0, NULL);


buf = clCreateBuffer (context, CL_MEM_READ_ONLY, ...);
 clEnqueueReadBuffer  (xfer_queues[device_id], buf,
                       CL_FALSE, ...);


...

global_work_size[0] = num_entries;
local_work_size[0] = 64;
clEnqueueNDRangeKernel (queue, kernel, 1, NULL,
                        global_work_size, local_work_size,
                        0, NULL, NULL);
```

short-sightedness in the multicore + GPU era

```
clGetDeviceIDs (NULL, CL_DEVICE_TYPE_DEFAULT, 1,
                &device_id, NULL);
queue = clCreateCommandQueue (context, device_id , 0, NULL);
```

explicit choice of device

```
buf = clCreateBuffer (context, CL_MEM_READ_ONLY, ...);
 clEnqueueReadBuffer  (xfer_queues[device_id], buf,
                       CL_FALSE, ...);
```

explicit data transfer

```
...

global_work_size[0] = num_entries;
local_work_size[0] = 64;
clEnqueueNDRangeKernel (queue, kernel, 1, NULL,
                        global_work_size, local_work_size,
                        0, NULL, NULL);
```

short-sightedness in the multicore + GPU era

```
clGetDeviceIDs (NULL, CL_DEVICE_TYPE_DEFAULT, 1,
                &device_id, NULL);
queue = clCreateCommandQueue (context, device_id , 0, NULL);
```

explicit choice of device

```
buf = clCreateBuffer (context, CL_MEM_READ_ONLY, ...);
 clEnqueueReadBuffer  (xfer_queues[device_id], buf,
                       CL_FALSE, ...);
```

explicit data transfer

```
...

global_w
local_wo
clEnqueueNDRangeKe
                        global_work_size, local_work_size,
                        0, NULL, NULL);
```

what about performance portability?

2

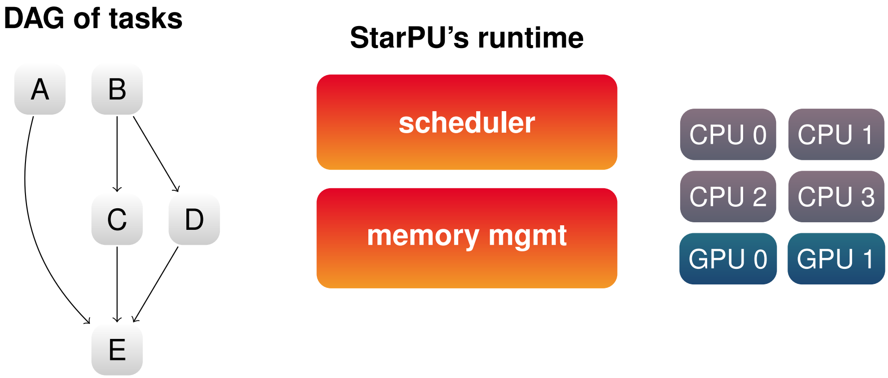
enter StarPU

runtime support to **schedule tasks** over all the available **processing units**

runtime support to
schedule tasks over all the
available **processing units**

- C library, LGPLv2.1+
- started in 2009

in a nutshell



the standard C API


```
void scale_vector_cpu (void *buffers[], void *args);
void scale_vector_opencl (void *buffers[], void *args);

static struct starpu_codelet scale_vector_codelet =
{
   .where = STARPU_CPU | STARPU_OPENCL,
   .cpu_func = scale_vector_cpu,
   .opencl_func = scale_vector_opencl,
   .nbuffers = 1,
   .modes =  STARPU_RW
};
```

the standard C API 2. defining the task's CPU implementation

```
void scale_vector_cpu (void *buffers[], void *arg)
{
   /* Unpack the arguments... */
   float *factor = arg;
   starpu_vector_interface_t *vector = buffers[0];
   unsigned n = STARPU_VECTOR_GET_NX (vector);
   float *val = (float *) STARPU_VECTOR_GET_PTR (vector);

   /* scale the vector */
   for (unsigned i = 0; i < n; i++)
     val[i] *= *factor;
}
```

the standard C API


```
starpu_data_handle vector_handle;
starpu_vector_data_register (&vector_handle, 0, vector,
                             NX, sizeof (vector[0]));

float factor = 3.14;

starpu_insert_task (&scale_vector_codelet,
                    STARPU_VALUE, &factor, sizeof factor,
                    STARPU_RW, vector_handle,
                    0);

...
starpu_task_wait_for_all ();
starpu_data_unregister (vector_handle);
```

the standard C API

4. invoking the task

```
starpu_data_handle vector_handle;
starpu_vector_data_register (&vector_handle, 0, vector,
                             NX, sizeof (vector[0]));

float factor = 3.14;

starpu_insert_task (&scale_vector_codelet,
                    STARPU_VALUE, &factor, sizeof factor,
                    STARPU_RW, vector_handle,
                    0);

...
starpu_task_wait_for_all ();
starpu_data_unregister (vector_handle);
```

the standard C API

```
starpu_data_handle vector_handle;
starpu_vector_data_register (&vector_handle, 0, vector,
                             NX, sizeof (vector[0]));

float factor = 3.14;

starp
                     STARPU_VALUE, &factor, sizeof factor,
                     STARPU_RW, vector_handle,
                     0);

...
starpu_task_wait_for_all ();
starpu_data_unregister (vector_handle);
```

can't it be made simpler & less error-prone?

3

the case for compiler support

promoting **library** interfaces
as **language constructs**

promoting **library** interfaces as **language constructs**

- GCC plug-in, GPLv3+
- started in 2011
- for GCC 4.5, 4.6, and 4.7

tasks are functions

```
void scale_vector (int size, float vector[size],
                   float factor)
  __attribute__ ((task));

void scale_vector_cpu (int size, float vector[size],
                       float factor)
  __attribute__ ((task_implementation
                     ("cpu", scale_vector)));

void
scale_vector_cpu (int size, float vector[size], float factor)
{
  for (int i = 0; i < size; i++)
    vector[i] *= factor;
}
```

tasks are functions

```
void scale_vector (int size, float vector[size],
                   float factor)
  __attribute__ ((task));

/* The implicit CPU implementation.  */
void
scale_vector (int size, float vector[size], float factor)
{
  for (int i = 0; i < size; i++)
    vector[i] *= factor;
}
```

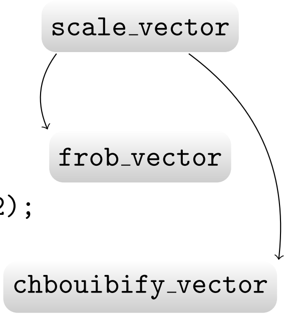
tasks submissions are async. function calls

```
    static float input[NX];
    ...

#pragma starpu register input
    ...
    scale_vector (NX, input, 42);



#pragma starpu wait
```

scale_vector

tasks submissions are async. function calls

```
    static float input[NX], out1[NX], out2[NX];
    ...

#pragma starpu register input
    ...
    scale_vector (NX, input, 42);
    ...
    frob_vector (NX, input, out1);
    chbouibify_vector (NX, input, out2);
#pragma starpu wait
```



tasks submissions are async. function calls

```
   static float input[NX], out1[NX], out2[NX];
   ...

#pragma starpu register input
   ...
   scale_vector (NX, input, 42);
   ...
   frob_vector (NX, input, out1);
   chbouibify_vector (NX, input, out2);
#pragma starpu wait
```

make sure data is in main memory

```
#pragma starpu unregister vector
  display_vector (vector);
```

memory management helpers

```
int
foo (int x)
{
  float vector[x]
    __attribute__ (( heap_allocated ));

  ...


  ...

}
```

allocated with `starpu_malloc`

deallocated at end-of-block

memory management helpers

```
int
foo (int x)
{
  float vector[x]
    __attribute__ (( heap_allocated , registered ));

  ...

  my_task (vector, x);
  ...

}
```

like `#pragma register`

unregistered at end-of-block

OpenCL task implementations

```
void vector_scal_opencl (int size, float vec[size],
                         float factor)
  __attribute__ ((task_implementation
                     ("opencl", vector_scal)));

void vector_scal_opencl (...)
{
  ...
  err = starpu_opencl_load_kernel (&kernel, &queue, &cl_prog
                                   "vector_scal_opencl", dev
  err = clSetKernelArg (kernel, 0, sizeof (val), &val);
  err |= clSetKernelArg (kernel, 1, sizeof (size), &size);
  ...
  err = clEnqueueNDRangeKernel (queue, kernel, 1, NULL, &glo
                                &local, 0, NULL, &event);
```

OpenCL task implementations

```
void vector_scal_opencl (int size, float vec[size],
                         float factor)
  __attribute__ ((task_implementation
                    ("opencl", vector_scal)));

void vector_scal_opencl (...)
{
  ...
  err = starpu_opencl_load_kernel (&kernel, &queue, &cl_prog
                                   "vector_scal_opencl", dev
  err = clSetKernelArg (kernel, 0, sizeof (val), &val);
  err |= clSetKernelArg (kernel, 1, sizeof (size), &size);
  ...
  err = clEnqueueNDRangeKernel (queue, kernel, 1, NULL, &glo
                                &local, 0, NULL, &event);
```

say no to copy/paste!

OpenCL task implementations

```
void vector_scal_opencl (int size, float vec[size],
                         float factor)
  __attribute__ ((task_implementation
                   ("opencl", vector_scal)));

#pragma starpu opencl vector_scal_opencl \
    "vector-scale.cl" "vector_scal_kern" \
    group_size ngroups
```

future work

- automatic registration of static arrays
- error out for buffers provably not registered
- OpenMP integration—e.g., generating tasks with `parallel for`
- OpenCL kernel code generation? (GRAPHITE-OpenCL?)
- ...

4

on GCC extensions

a personal journey

- Scheme, an extensible language

```
(define-syntax and
  (syntax-rules ()
    ((_) #t)
    ((_ x) x)
    ((_ x y ...) (if x (and y ...) #f))))
```

a personal journey

- Scheme, an extensible language
- Guile, a simple optimizing compiler

```
(define (optimize! x env opts)
  (fix-letrec!
    (cse
      (peval (expand-primitives!
               (resolve-primitives! x env))
             env))))
```

a personal journey

- Scheme, an extensible language
- Guile, a simple optimizing compiler
- GCC, an inspiring & intricate beast

```
tree bind = build3 (BIND_EXPR, void_type_node,
                    vars, stmts,
                    build_block (vars, NULL_TREE,
                                 task_impl, NULL_TREE));
DECL_SAVED_TREE (task_impl) = bind;
DECL_INITIAL (task_impl) = BIND_EXPR_BLOCK (bind);
rest_of_decl_compilation (task_impl, true, 0);
allocate_struct_function (task_impl, false);
cgraph_finalize_function (task_impl, false);
```

the case for plug-ins

1. **technical reasons**
2. **“administrative” reasons**

the case for plug-ins

1. **technical reasons**
   - **enhance** programming interfaces
   - avoid common programming **errors**
2. **“administrative” reasons**

the case for plug-ins

1. **technical reasons**
   - **enhance** programming interfaces
   - avoid common programming **errors**
2. **“administrative” reasons**
   - **independent**, maturing project
   - **tightly related** to StarPU runtime support development

frustration of a plug-in writer

technically working *in* GCC
but **socially** *outside* of it

- A: “I wrote a GCC plug-in for...”

- A: “I wrote a GCC plug-in for...”
- B: “Why didn’t you use L**M? It’s sooo fancy, and modular, and bla bla bla...”

- A: “I wrote a GCC plug-in for...”
- B: “Why didn’t you use L**M? It’s sooo fancy, and modular, and bla bla bla...”
- ...

the programming interface issue

- API + ABI instability
- API insecurity
  - using APIs that Thou Shall Not Use (`tree.h`!)

solving the programming interface issue

- wrap the "real" API into an "officially stable" API?

solving the programming interface issue

- wrap the “real” API into an “officially stable” API?
- embed a high-level extension language?
  - **Guile**, MELT, Python

it's all about freedom

making it easier
to **exert freedom #1**
on GCC

summary

- a step forward in **portable heterogeneous programming**
- the case for **language & compiler support**
- GCC plug-ins allow for **richer programming interfaces**

ludovic.courtes@inria.fr

`http://runtime.bordeaux.inria.fr/StarPU/`