The PEPPHER Composition Tool: Performance-Aware Dynamic Composition of Applications for GPU-based Systems

Usman Dastgeer, Lu Li, Christoph Kessler
Linköping University, Sweden
Outline

- PEPPHER project (briefly)
- PEPPHER component model (briefly)
- PEPPHER composition tool
- Experimental results
- Conclusion and Outlook
Performance Portability and Programmability for Heterogeneous Many-Core Architectures

- **Universities**
  - University of Vienna (coord.), Austria
  - Chalmers University, Sweden
  - Karlsruhe Institute of Technology, Germany
  - Linköping University, Sweden
  - Vienna University of Technology, Austria

- **Research center**
  - INRIA, France

- **Companies**
  - Intel, Germany
  - Codeplay Software Ltd., UK
  - Movidius Ltd. Ireland
We focus on heterogeneous systems
  - Single-chip (instance: Cell BE)
  - Single-node (instance: CPU+GPU(s))

Different programming models may be suitable for different core types

**We do not** focus on parallelization per se

**We do not** propose a new programming model

We develop mechanisms for combining existing programming models
**Aim**: Enable **productive, performance-portable, and efficient** programming of heterogeneous multi-core systems

**Work Areas**

- **methods and tools for component-based software**
- portable compilation techniques
- adaptive, auto-tuned components, algorithms and data structures
- **efficient, flexible run-time systems**
- hardware support mechanisms for auto-tuning, synchronization and scheduling

- Survey article in *IEEE Micro*  Sep./Oct. 2011
PEPPHER Component Model
Component Interface and Implementations

Component Interface

«interface»
C

\( f(\text{parameters}) \)

Interface meta-data
Component Interface and Implementations

- Platforms (CPU, accelerator cores)
- Algorithms
- Tunable parameter settings
- Compiler transformations
- ...

Component Interface

```

```

Variant meta-data

Component Implementation Variants
```
Components Interface and Implementations

**Component Interface**

```
C

sort (float *arr, int size)
```

**Variant Implementations**

- **C1**
  - `sort_ms_cpu` ...

- **Cn**
  - `sort_qs_cuda` ...

**Component Implementation Variants**

- Platforms (CPU, accelerator cores)
- Algorithms
- Tunable parameter settings
- Compiler transformations
- ...

---

*PEPHER*
Which implementation to use for a given execution context (architectural features, system workload, problem size, input data etc.)?
What is a PEPPHER component?
- interface and implementations (variants)
- Enriched with meta-data
  - Dependences, Resource requirements, ...
- State-less
- Composition points (calls) on CPU only

Separation of concerns
- Declaration of functionality
- Implementation of functionality
- Architecture/platform description

Parallelism sources
- Parallel components (intra-component parallelism)
  - OpenMP, pthreads, CUDA, OpenCL, Offload-C++, SDK ...
  - Portable coordination constructs (inter-component parallelism)

Open and extensible
- Future implementations, languages, architectures
**Component Meta-Data**

### Interface Descriptor
- Name
- Parameters’ modifier (read, write...)
- Supported performance metric
- ...

### Component Descriptor
- Provided interface
- Required interfaces (if any)
- Supported target platform(s)
- Source file(s)
- Deployment, e.g. compilation rules
- Resource requirements
- Performance model
- Tunable parameters (if any)
- ...

---

**XML Interface descriptor**

```xml
<peppher:interface ...>
  <method name="foo">
    ...
  </method>
</peppher:interface>
```
Staged Composition

**Static composition:**
at deployment time, user-guided or off-line autotuning + static narrowing of set of candidates

**Dynamic composition:**
on-line tuning, selection of the expected best component

Contributed or generated variants with static performance predictions

Composition tool generates wrappers for both.
PEPPHER Composition Tool
Input: Application directory and main module descriptor (Main.xml)
Start from Main.xml
Recursively explore the component interfaces and variants
  Non-matching variants are discarded
Process interfaces and their component variants bottom-up in “required by” topological order
For each interface:
  Parse and represent matching component variant descriptors
  Generate platform-specific headers
Later: Look up performance model data and/or call perf.-prediction code
Later: Build dispatch tables for static composition
  Generate stubs (wrapper, proxy functions)
  Call the native compiler
Generate project makefile → builds executable
Composition Tool IR

1. Read XML Descriptors.
3. Create Internal representation (IR).

IR (component tree)

Composition Processing

1. Static Composition
2. Component expansion
3. Other composition decisions

(Modified) IR (component tree)

Code Generation

1. Stubs generation (Wrappers)
2. Header files (peppher.h)
3. Compilation (Makefile)

Executable StarPU code

IR decouples processing (e.g., static composition decisions) from the XML schema.

* = not implemented in the current prototype.
+ = partially implemented in the current prototype.
Smart Containers

- **C++ STL like containers**
  - 1D Vector, 2D Dense/Sparse matrix, Scalar
  - Generic on element type using C++ templates

- **Internally do memory management**
  - Interact with StarPU data management API
  - Ensure consistency for data accesses.

- **Support asynchronous component execution**
  - Support inter-component parallelism
  - Task dependencies are inferred implicitly based on the data dependencies.
Smart Containers

We assume that all 4 component calls are executed on the GPU.
Demo of the Composition Tool

Example:  Simple Vector-scale  \( v \leftarrow v \times c \)

(main + 1 component: Vector scale routine)
Demo of the Composition Tool: Summary

**Step 1:** $> \textit{compose} \ -\text{generateCompFiles=} \textit{vector\_scale.h}$

**Step 2:** // edit generated files if necessary...

**Step 3:** $> \textit{compose} \ \textit{main.xml}$

**Step 4:** $> \textit{make}$

**Run:** $> ./\textit{main}$
**Step 1:** Create skeletons of XML descriptor files from method declaration.

```bash
$> compose -generateCompFiles="vector_scale.h"
```

```cpp
void vector_scale (peppher::Vector<float> &vec, float factor);
```
**Step 1:** Create skeletons of XML descriptor files from method declaration.

Before

![](image)

After

![](image)
Demo of the Composition Tool: Step 2

- **Step 2**: Complete the **XML descriptors** generated by the tool...
Demo of the Composition Tool: Step 2

XML descriptor for vector_scale interface

```xml
<?xml version="1.0"?>
<peppher:component xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:peppher="http://www.peppher.eu/ComponentMetaData0.1"
    xsi:schemaLocation="http://www.peppher.eu/ComponentMetaData0.1 ComponentMetaData.xsd">
    <peppher:interface name="vector_scale">
        <peppher:parameters>
            <peppher:parameter name="arr" type="peppher::Vector" elemType="float" accessMode="readwrite" />
            <peppher:parameter name="factor" type="float" accessMode="read" />
        </peppher:parameters>
    </peppher:interface>
</peppher:component>
```
Demo of the Composition Tool: Step 2

XML descriptor for CUDA implementation

```xml
<?xml version="1.0"?>
<pepper:component xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:pepper="http://www.pepper.eu/ComponentMetaData0.1"
xsi:schemaLocation="http://www.pepper.eu/ComponentMetaData0.1 ComponentMetaData.xsd" >

<pepper:implementation name="scale_cuda_func">

  <pepper:sourceFiles>
    <pepper:sourceFile name="vector_scale_cuda.cu" version="1.0" language="CUDA">
      <pepper:compilation>
        <pepper:compiler>
          <pepper:name>nvcc</pepper:name>
        </pepper:compiler>
      </pepper:compilation>
    </pepper:sourceFile>
  </pepper:sourceFiles>

  <pepper:providedInterfaces>
    <pepper:providedInterface name="vector_scale"></pepper:providedInterface>
  </pepper:providedInterfaces>

  <pepper:targetPlatform name="CUDA">
  </pepper:targetPlatform>

</pepper:implementation>
</pepper:component>
```
Demo of the Composition Tool: Step 2

Main function of the application

```c++
int main(int argc, char **argv)
{
    PEPPHER_INITIALIZATE(); // initialize PEPPHER framework...

    // scoping to ensure that destructor of peppher containers is called before we call PEPPHER_SHUTDOWN

    // A vector of 30 floats with PEPPHER vector container */
    peppher::Vector<float> vector(30);

    vector.randomize(1,50); // initialize with random values between 1 and 50
    std::cout"BEFORE: "<<vector<<"\n";

    // compose call which is actually to a wrapper which intercept and execute as StarPU task... Asynchronous
    for(int i=0;i<3;i++)
        vector_scale(vector, 3.5f);
    std::cout"AFTER: "<<vector<<"\n"; // will block until vector data becomes available

    }

    PEPPHER_SHUTDOWN(); // de-initialize PEPPHER framework...
    return 0;
}
Step 3: Write a main.xml descriptor and run the composition tool:

$> compose main.xml

**Before**

- vector_scale.xml
- main.c
- main.xml
- vector_scale
  - vector_scale_opencl.xml
  - vector_scale_opencl.c
  - vector_scale_opencl_codelet.cl
  - vector_scale_cuda.xml
  - vector_scale_cuda.cu
  - vector_scale_cpu.xml
  - vector_scale_cpu.c

**After**

- vector_scale_opencl.xml
- vector_scale_opencl.c
- vector_scale_opencl_codelet.cl
- vector_scale_cuda.xml
- vector_scale_cuda.cu
- vector_scale_cpu.xml
- vector_scale_cpu.c
- makefile
- peppher.h
- vector_scale_wrapper.h
Step 3: Write a main.xml descriptor and run the composition tool:

$> compose main.xml

The above statement generates:

- Wrappers (stubs)
  - intercept invocations to components,
  - call dispatch code (in case of static composition)
  - do the marshaling/unmarshaling of arguments to match the StarPU task format.
    - void task(void *buffers[], voig *args);

- Makefile
  - To compile and link the program.
Wrapper function (stub) for vector_scal generated by the Composition Tool

```c
// #include and #define code ....

struct ROA_vector_scale
{
    float factor;
};

struct struct_vector_scale
{
    starpu_codelet cl_vector_scale;
    bool cl_vector_scale_init;
    struct vector_scale()
    {
        cl_vector_scale_init = false;
    }
};

extern void scale_cpu_func(pepper::Vector<float> &arr, float factor);

void scale_cpu_func_wrapper(void *buffers[], void *_args)
{
    // pepper container reconstruction from raw pointer
    pepper::Vector<float> arr_handle((float *)STARPU_VECTOR_GET_PTR((starpu_vector_interface_t *)buffers[0]),
                                     STARPU_VECTOR_GET_NX((starpu_vector_interface_t *)buffers[0]));

    scale_cpu_func(arr_handle, ((ROA_vector_scale *)_args)->factor);
}

extern void scale_cuda_func(pepper::Vector<float> &arr, float factor);

void scale_cuda_func_wrapper(void *buffers[], void *_args)
{
    // pepper container reconstruction from raw pointer
    pepper::Vector<float> arr_handle((float *)STARPU_VECTOR_GET_PTR((starpu_vector_interface_t *)buffers[0]),
                                     STARPU_VECTOR_GET_NX((starpu_vector_interface_t *)buffers[0]));

    scale_cuda_func(arr_handle, ((ROA_vector_scale *)_args)->factor);
}
```
Wrapper function (stub) for `vector_scal` generated by the Composition Tool

```c
struct starpu_opencl_program *codelet_vector_scale = NULL;
cl_kernel kernel_vector_scale;
cl_command_queue queue_vector_scale;

extern void scale_opencl_func(pepper::Vector<float> &arr, float factor);

/* wrapper from StarPU convention to OpenCL implementation convention */
void scale_opencl_func_wrapper(void *buffers[], void *args[])
{
    int id = starpu_worker_get_id();
    int devid = starpu_worker_get_devid(id);
    int err = starpu_opencl_load_kernel(&kernel_vector_scale, &queue_vector_scale, codelet_vector_scale,
                   "vector_mult_opencl", devid);

    if (err != CL_SUCCESS) STARPU_OPENCL_REPORT_ERROR(err);

    // pepper container reconstruction from raw pointer
    pepper::Vector<float> arr_handle((float *)STARPU_VECTOR_GET_PTR((starpu_vector_interface_t *)buffers[0]),
                             STARPU_VECTOR_GET_NX((starpu_vector_interface_t *)buffers[0]));

    scale_opencl_func(arr_handle, ((ROA_vector_scale *)_args)->factor);
    clFinish(queue_vector_scale);
    starpu_opencl_release_kernel(kernel_vector_scale);
}
```
Wrapper function (stub) for vector_scal generated by the Composition Tool

```c
/* wrapper for converting component call into a runtime task execution */
void vector_scale(pepper::Vector<float> &arr, float factor)
{
    static struct_vector_scale objSt_vector_scale;
    // codelete initialization only once, at first invocation
    if(!objSt_vector_scale.cl_vector_scale_init)
    {
        objSt_vector_scale.cl_vector_scale.where = 0|STARPU_CPU|STARPU_CUDA|STARPU_OPENCL;
        objSt_vector_scale.cl_vector_scale.cpu_func = scale_cpu_func_wrapper;
        objSt_vector_scale.cl_vector_scale.cuda_func = scale_cuda_func_wrapper;
        objSt_vector_scale.cl_vector_scale.opencl_func = scale_opencl_func_wrapper;
        objSt_vector_scale.cl_vector_scale.nbuffers = 1;
        // load the OpenCL kernel if not loaded yet
        if(!codelet_vector_scale)
        {
            codelet_vector_scale = new starpu_opencl_program();
            int i = starpu_opencl_load_opencl_from_file("vector_scale/vector_scale_opencl_codelet.cl",
                    codelet_vector_scale, NULL);
        }
    } objSt_vector_scale.cl_vector_scale_init = true;

    // construct pointer to ROA_vector_scale struct containing factor
    ROA_vector_scale *arg_vector_scale = (ROA_vector_scale *)malloc(sizeof(ROA_vector_scale));
    arg_vector_scale->factor = factor;

    // prepare task
    struct starpu_task *task = starpu_task_create();
    task->synchronous = 0; // asynchronous execution
    task->cl = &(objSt_vector_scale.cl_vector_scale);

    // The pepper::Vector implicitly manages creation of its data handle
    task->buffers[0].handle = arr.registerWithStarPU();
    task->buffers[0].mode = STARPU_RW;

    // destroy ROA_vector_scale pointer when the task execution finishes
    task->callback_func = free;
    task->callback_arg = (void*)arg_vector_scale;

    task->cl_arg = arg_vector_scale;
    task->cl_arg_size = sizeof(ROA_vector_scale);

    // submit the task with CPU, CUDA and OpenCL implementations
    int ret = starpu_task_submit(task);

    // check for erroneous task submission.
    if(ret == -ENODEV)
    {
        fprintf(stderr, "ERROR: No worker may execute this task\n");
        return;
    }
}
```
CFLAGS += $(shell pkg-config --cflags libstarpu)
LDFLAGS += $(shell pkg-config --libs libstarpu)

main : main.o vector_scale_cpu.o vector_scale_cuda.o vector_scale_opencl.o
    g++ main.o vector_scale_cpu.o vector_scale_cuda.o vector_scale_opencl.o $(CFLAGS) $(LDFLAGS) -o main

main.o : main.c
    g++ main.c $(CFLAGS) -I/opt/global/peppher/composTool/containers -c -o main.o

vector_scale_cpu.o : ./vector_scale/vector_scale_cpu.c
    g++ ./vector_scale/vector_scale_cpu.c $(CFLAGS) -I/opt/global/peppher/composTool/containers -c -o vector_scale_cpu.o

vector_scale_cuda.o : ./vector_scale/vector_scale_cuda.cu
    nvcc ./vector_scale/vector_scale_cuda.cu $(CFLAGS) -I/opt_new/global/peppher/composTool/containers -c -o vector_scale_cuda.o

vector_scale_opencl.o : ./vector_scale/vector_scale_opencl.c
    g++ ./vector_scale/vector_scale_opencl.c $(CFLAGS) -I/opt/global/peppher/composTool/containers -c -o vector_scale_opencl.o

clean:
    rm -f main *.o ~

PEPPHER
Demo of the Composition Tool: Step 4

Step 4: $>\text{make}$

```
[dastgeer@node05 vector_scale_cont_part_async]$ $>\text{make}$
[dastgeer@node05 vector_scale_cont_part_async]$ make

gcc  main.c -I/opt_new/global/peppher/install/include -I/opt_new/global/cuda/cuda4.0/cuda/include -I/usr/include/libxml2 -I/opt_new/global/peppher/composTool/containers -c -o main.o

$>\text{make}$

nvcc ./vector_scale/vector_scale_cpu.c -I/opt_new/global/peppher/install/include -I/opt_new/global/cuda/cuda4.0/cuda/include -I/usr/include/libxml2 --compiler-options -fpermissive -I/opt_new/global/peppher/composTool/containers -c -o vector_scale_cpu.o

nvcc ./vector_scale/vector_scale_cudacu.c -I/opt_new/global/peppher/install/include -I/opt_new/global/cuda/cuda4.0/cuda/include -I/usr/include/libxml2

cc1: warning: command line option "-fpermissive" is valid for C++/ObjC++ but not for C
cc1: warning: command line option "-fpermissive" is valid for C++/ObjC++ but not for C

$>\text{make}$

gcc  ./vector_scale/vector_scale_opencl.c -I/opt_new/global/peppher/install/include -I/opt_new/global/cuda/cuda4.0/cuda/include -I/usr/include/libxml2 -L/opt_new/global/peppher/install/lib -L/opt_new/global/cuda/cuda4.0/cuda/lib64 -L/opt_new/global/cuda/cuda4.0/cuda/lib -lstarpu -lcudart -lcublas -lcuda -lstdc++ -lOpenCL -lhwloc -o main

[dastgeer@node05 vector_scale_cont_part_async]$ $>\text{make}$

Warning: StarPU was configured with --with-fxt, which slows down a bit

BEFORE: 50 16 34 32 37 2 2 31 31 10 36 4 29 10 9 47 3 43 46 33 1 1 38 32 30 42 37 34 34 13
AFTER:

************ Hello in vector-scale CPU call ************

************ Hello in vector-scale CUDA call ************

************ Hello in vector-scale OpenCL call ************

2143.75 686 1457.75 1372 1586.38 85.75 85.75 1329.12 1329.12 428.75 1543.5 171.5 1243.38 428.75 385.875 2015.12 128.625 1843.62 197
2.25 1414.88 42.875 42.875 1629.25 1372 1286.25 1800.75 1586.38 1457.75 1457.75 557.375

[dastgeer@node05 vector_scale_cont_part_async]$ $>\text{make}$
Demo of the Composition Tool: Summary

Step 1: `$> compose -generateCompFiles="vector_scale.h"`

Step 2: // edit generated files if necessary…

Step 3: `$> compose main.xml`

Step 4: `$> make`

Run: `$> ./main`
Experimental Results
## Benchmark applications

<table>
<thead>
<tr>
<th>Application</th>
<th>Domain</th>
<th>ShortName</th>
</tr>
</thead>
<tbody>
<tr>
<td>RungeKutta ODE Solver</td>
<td>Linear Algebra (ODE)</td>
<td>ODE Solver</td>
</tr>
<tr>
<td>CFD Solver <em>(Rodinia)</em></td>
<td>Fluid Dynamics</td>
<td>cfd</td>
</tr>
<tr>
<td>LU Decomposition <em>(Rodinia)</em></td>
<td>Linear Algebra</td>
<td>lu</td>
</tr>
<tr>
<td>Needleman-Wunsch <em>(Rodinia)</em></td>
<td>Bioinformatics</td>
<td>nw</td>
</tr>
<tr>
<td>Particle Filter <em>(Rodinia)</em></td>
<td>Medical Imaging</td>
<td>particlefilter</td>
</tr>
<tr>
<td>Path Finder <em>(Rodinia)</em></td>
<td>Grid Traversal</td>
<td>pathfinder</td>
</tr>
<tr>
<td>Breadth-First Search <em>(Rodinia)</em></td>
<td>Graph Algorithms</td>
<td>bfs</td>
</tr>
<tr>
<td>HotSpot <em>(Rodinia)</em></td>
<td>Physics Simulation</td>
<td>hotspot</td>
</tr>
<tr>
<td>Dense Matrix-Matrix Multiplication</td>
<td>Linear Algebra</td>
<td>GEMM</td>
</tr>
<tr>
<td>Sparse Matrix Vector multiplication</td>
<td>Linear Algebra</td>
<td>SPMV</td>
</tr>
</tbody>
</table>
Performance evaluation
Hybrid Execution

- SpMV, hybrid execution (by StarPU runtime)
  1 GPU (CUDA, CUSP) + 4 CPUs in parallel

Platform: System with NVIDIA C2050 GPUs, 4 Intel Xeon CPUs E5520@2.27GHz

Relative Speedup (Direct CUSP CUDA = 100%)

<table>
<thead>
<tr>
<th>Matrices from UF Collection:</th>
<th>Non-zeros</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structural</td>
<td>2.7M</td>
</tr>
<tr>
<td>HB</td>
<td>219.8K</td>
</tr>
<tr>
<td>Convex</td>
<td>0.9M</td>
</tr>
<tr>
<td>Simulation</td>
<td>4.6M</td>
</tr>
<tr>
<td>Network</td>
<td>565K</td>
</tr>
<tr>
<td>Chemistry</td>
<td>758K</td>
</tr>
</tbody>
</table>

Hybrid variant requires less communication.
Performance Portability (1: C2050 GPU)

Times (incl. partitioning and communication) averaged over different problem sizes

Normalized exec. time

OpenMP
CUDA C2050
PEPPHER Ctool
Performance Portability (2: C1060 GPU)

Times (incl. partitioning and communication) averaged over different problem sizes
ODE Solver Application

- 9 components (each with C++ and CUDA implementation variants)
- 10613 invocations in total
- Tight data dependence structure → using a single GPU performs best
- **Low overhead** of composition code + StarPU task handling 😊

![Graph showing execution time vs. problem size](chart.png)
Programmer Efficiency (LOC) evaluation
## Programmer Efficiency (LOC analysis)

<table>
<thead>
<tr>
<th>Application</th>
<th>Tool (LOC)</th>
<th>Direct (LOC)</th>
<th>Difference (LOC, %)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SpMV</td>
<td>293</td>
<td>376</td>
<td>83, 29</td>
</tr>
<tr>
<td>SGEMM</td>
<td>140</td>
<td>229</td>
<td>89, 63</td>
</tr>
<tr>
<td>bfs</td>
<td>256</td>
<td>364</td>
<td>108, 42</td>
</tr>
<tr>
<td>cfd</td>
<td>200</td>
<td>323</td>
<td>123, 62</td>
</tr>
<tr>
<td>hotspot</td>
<td>327</td>
<td>447</td>
<td>120, 37</td>
</tr>
<tr>
<td>lud</td>
<td>510</td>
<td>586</td>
<td>76, 15</td>
</tr>
<tr>
<td>nw</td>
<td>359</td>
<td>449</td>
<td>90, 25</td>
</tr>
<tr>
<td>particlefilter</td>
<td>652</td>
<td>748</td>
<td>96, 15</td>
</tr>
<tr>
<td>pathfinder</td>
<td>186</td>
<td>275</td>
<td>89, 48</td>
</tr>
<tr>
<td>ODE Solver</td>
<td>800</td>
<td>1252</td>
<td>452, 57</td>
</tr>
</tbody>
</table>

Comparison of total source LOC (Lines of Code) written by the programmer when using the composition tool compared to an equivalent code written directly using the runtime system.
Summary and Outlook

- **PEPPHER component model**
  - Interface, implementations, meta-data

- **Composition tool prototype**
  - Static and *dynamic* composition
  - StarPU integration for on-line tuning of selection
  - Smart containers (if using C++)
    - Support inter-component parallelism
    - Lazy memory copying, reduced task initialization overhead
  - Generic components (if using C++)
  - Also: Adaptive off-line sampling and tuning to generate time predictors *(IWAPT’12)*

- **Future/ongoing work**
  - Stand-alone variant
    - Integration of a Platform Description Language
    - Nested and recursive components (StarPU limitation)
  - User-defined vs. history-based performance prediction
  - Support for individual call annotations and additional coordination support
  - Optimization across multiple invocations
  - Versioning support and component repositories
QUESTIONS?
Legacy application (C++)

main()
{
    ...
    foo(A,n);
}

function outlining

Legacy application’

main()
{
    ...
    foo(A,n);
}

compile

PEPPHERing (1 step)

Interface descriptor

<peppher:interface name="Fooing">
...
</peppher:interface>

Platform descriptors

XML

void foo_cpu (...)
{
    ...
}
void foo_cuda_alg1 (...)
{
    ...
}
void foo_cuda_alg2 (...)
{
    ...
}

Component source files

Component descriptors

<peppher:component name="foo_cuda_alg1">
...
</peppher:component>
<peppher:component name="foo_cuda_alg2">
...
</peppher:component>
<peppher:component name="foo_cpu">
...
</peppher:component>
<peppher:component name="foo">
...
</peppher:component>
<peppher:component name="foo_cuda">
...
</peppher:component>

Platform-specific header files

class Fooing ...
{
    void foo(...);
};

Platform-specific header files

extern void foo(...);

Architecture / Platform specifications

<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>

Platform descriptor repository

Component source files

Component descriptors

compile as specified

Platform-specific header files

<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
...
</peppher:platform>
<peppher:platform ...>
We assume that all 4 component calls are executed on the GPU.
We assume that all 4 component calls are executed on the GPU.
We assume that all 4 component calls are executed on the GPU.
We assume that all 4 component calls are executed on the GPU.
Smart Containers

We assume that all 4 component calls are executed on the GPU.
We assume that all 4 component calls are executed on the GPU.
We assume that all 4 component calls are executed on the GPU.
Wrapper function (stub) for vector_scal generated by the Composition Tool

```c
#ifndef VECTOR_SCAL_WRAPERR
#define VECTOR_SCAL_WRAPERR

#include <map>
#include <string>
#include <iostream>

using namespace std;

struct readOnlyArgs_vector_scal
{
    float factor;
};

typedef struct readOnlyArgs_vector_scal ROA_vector_scal;
starpu_codelet cl_vector_scal;
extern void scal_cpu_func ( float * arr, unsigned size, float factor );
void scal_cpu_func_wrapper ( void *buffers[], void *args )
{
    scal_cpu_func (((float *)STARPU_VECTOR_GET_PTR( (starpu_vector_interface_t *)buffers[0]),STARPU_VECTOR_GET_PTR( (starpu_vector_interface_t *)buffers[0]),((ROA_vector_scal *)args)->factor);
}

extern void scal_cuda_func ( float * arr, unsigned size, float factor );
void scal_cuda_func_wrapper ( void *buffers[], void *args )
{
    scal_cuda_func (((float *)STARPU_VECTOR_GET_PTR( (starpu_vector_interface_t *)buffers[0]),STARPU_VECTOR_GET_PTR( (starpu_vector_interface_t *)buffers[0]),((ROA_vector_scal *)args)->factor);
}

struct starpu_opencl_program codelet_vector_scal;
cl_kernel kernel_vector_scal;
cl_command_queue queue_vector_scal;
```