Addressing DAGs of Heterogeneous CPU-GPU Parallel Tasks Through High-Productivity Single-Source PHAST Library

Biagio Peccerillo   Sandro Bartolini

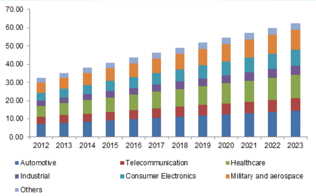
Università degli Studi di Siena

Siena, IWES 2018

Trends in the Embedded World

Science-fiction scenarios

- The demand for embedded applications and technologies is growing year by year



Europe embedded system market size, by application, 2012-2023 (USD Billion)

- Progress in the field is pushing us closer and closer to *science-fiction* scenarios:
  - Smart homes
  - Smart cities
  - Self-driving cars
  - Voice assistants
  - Virtual and Augmented Reality

Trends in the Embedded World

Constraints

- In order to make these scenarios happen, embedded devices must meet two major constraints:
  1. High-performance
  2. Low power consumption
- Today, these needs are better approximated by a plurality of *parallel* devices, by a *heterogeneous* approach:
  - multi-core CPUs
  - GPUs
  - FPGAs
  - TPUs (TensorFlow Processing Units)
- Mastering programming techniques for all these devices would be infeasible without *productivity-oriented heterogeneous frameworks*

PHAST Library

Main features

PHAST Library: Parallel Heterogeneous-Architecture STL-like Template Library

- High-level modern C++ library
- Heterogeneous: can be targeted on *NVIDIA GPUs* & *Multi-core CPUs* (at the moment...) via a single globally-defined macro
- Inner layers are implemented in *std::threads* & *CUDA*
- Permits to set parallelization parameters independently of application code
- Allows for low-level architecture-specific optimizations in **#ifdef**-protected blocks

---

B. Peccerillo and S. Bartolini, "PHAST – A portable high-level modern C++ programming library for GPUs and multi-cores," IEEE Transactions on Parallel and Distributed Systems, pp. 1–15, 2018 [Online]. Available: https://www.doi.org/10.1109/TPDS.2018.2855182

PHAST Library

Structure

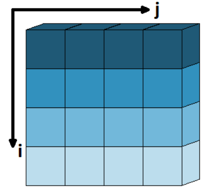
- Multi-dimensional Containers
  - 1D vector, 2D matrix, and 3D cube
- Iterators
  - Permit visiting containers *piece*-wise – not only *element*-wise
  - Various grains of parallelism explored with the same formalism
- Algorithms & Functors
  - STL portings and linear-algebra related ones
  - Functors allow users to personalize computation on container sub-portions of various shapes
- Parallelization Parameters
  - Estimated via heuristics, but can also be tuned by programmers
- Hierarchical Design
  - *In-functor* containers can be visited via *in-functor* iterators and manipulated in *in-functor* algorithms

PHAST Library

A quick example

```
#include <iostream>
#include "phast.h"
#include "print_utility.h"

template <typename T, unsigned int policy =
    phast::get_default_policy()>
struct linear_row :
    phast::functor::func_vec<float, policy>
{
    _PHAST_METHOD void operator()(
        phast::functor::vector<T>& row)
    {
        this->fill(row.begin(), row.end(),
            static_cast<T>(this->get_index()));
    }
};

int main(const int argc, const char* argv[])
{
    phast::matrix<float> mat(5, 4);
    phast::for_each(mat.begin_i(), mat.end_i(),
        linear_row<float>());

    std::cout << mat << std::endl;
    return 0;
}
```

1. Declare a matrix object;
2. Fill its rows with increasing values;
3. Print the matrix



matrix iterator_i

0	0	0	0
1	1	1	1
2	2	2	2
3	3	3	3
4	4	4	4

output

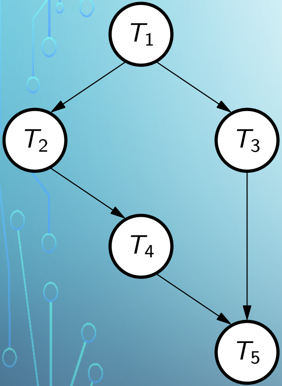
PHAST Library

PHAST parallelism

- PHAST philosophy resembles STL's: the same computation is applied to *collections* of elements, but:
  - The concept of *element* is flexible
  - Computation is parallel
  - Can be targeted on multiple devices
  - Container topology is less strict (up to three dimensions)
- STL's formalism is good to express *data-parallel* problems, but it comes short for other classes of parallel problems
- Data-parallel is in fact characterized by the application of *the same computation* on multiple data
- Many applications that arise in embedded context *are not data-parallel*!

Task Parallelism

Definition



- Task parallelism: *multiple* calculations on *multiple* data
- Dependencies between tasks can be expressed and visualized in the form of a Direct Acyclic Graph (DAG)
- These dependencies also regulate the order of execution and the opportunities of parallelization
- On multi-core processors, this can be achieved by executing tasks on different cores
- A synchronization mechanism is needed: *dependent* tasks cannot execute before their *dependencies*

Task Parallelism

Our proposal: the task class

```
template <typename Callable, typename... Args>
class task;

template <typename Callable, typename... Args>
task<Callable, Args...> make_task(Callable&& callable, Args&&... args);
```

- *task* is a C++14 template class that wraps a *callable* (free function, method, lambda, or functor) and its arguments as a tuple
- *make_task* is a free function that takes a callable and its arguments as parameters and returns a *task*
- The task exposes a *get()* method – its invocation executes the underlying callable on its arguments and returns its return-value to the caller
- If the task depends on other tasks, their *get()* methods are invoked before

The task class

How to express dependencies

- Dependencies between tasks can be expressed when *make_task* is invoked: *any of the parameters of the callable can be replaced with a task wrapping a callable that returns the needed value*
- The only constraint is that the type returned by the *independent task* must match the type of the argument of the callable invoked inside the *dependent task*
- This mechanism is achieved in three steps inside the dependent task *get()* method:
  1. For each task in the argument tuple, its *get()* method is concurrently launched via *std::async*
  2. The dependent task waits for the completion of each asynchronous execution and saves their results
  3. When all the asynchronous executions complete, the arguments of the underlying callable are ready and it can be invoked

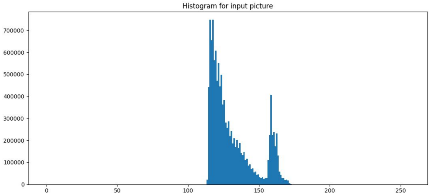
The task class

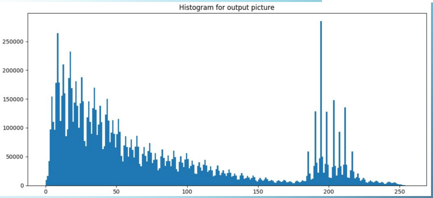
task and PHAST integration

- PHAST algorithms and methods are synchronous, but asynchronicity can be achieved by invoking them in tasks
- Users must be sure that no PHAST container is *modified* in more than one task at once
- This dependency can be expressed by returning PHAST containers from tasks and using them as arguments in dependent tasks' callables
- *Parallelism* and *heterogeneity* are achieved by executing data-parallel PHAST algorithms on a device (NVIDIA GPU or multi-core) *decoupled* from the device where tasks are scheduled (multi-core)

Task & PHAST

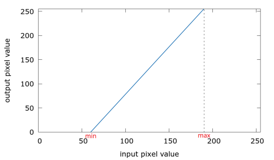
A full example: image histogram stretch





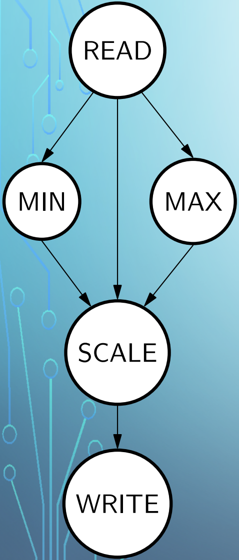
- A grayscale image is read – pixels are modeled as *uchar8* in the range [0, 255]
- The *minimum* and *maximum* pixel values are acquired
- For each pixel in the image, it is rescaled according to the equation

$$out = \frac{255-0}{max-min} \times (in - min)$$



Task & PHAST

A full example: image histogram stretch



Task-PHAST implementation of the image histogram stretch application

```
#include <opencv2/opencv.hpp>
#include "task.h"
#include "phast.h"
#include "print_utility.h"

using uchar8_t = unsigned char;

phast::matrix<uchar8_t>* read_image(const char* filename);

uchar8_t min_pixel(const phast::matrix<uchar8_t>* p_img);

uchar8_t max_pixel(const phast::matrix<uchar8_t>* p_img);

phast::matrix<uchar8_t>* scale(phast::matrix<uchar8_t>* p_img,
    uchar8_t min, uchar8_t max);

int write_image(phast::matrix<uchar8_t>* p_img, const char* filename);

int main(const int argc, const char* argv[])
{
    if (argc < 3)
    {
        std::cerr << "Using " << argv[0] << " <input image> <output image>\n";
        return -1;
    }

    auto read_task = make_task(read_image, argv[1]);
    auto min_task = make_task(min_pixel, read_task);
    auto max_task = make_task(max_pixel, read_task);
    auto scale_task = make_task(scale, read_task, min_task, max_task);
    auto write_task = make_task(write_image, scale_task, argv[2]);

    write_task.get();
    return 0;
}
```

Task & PHAST

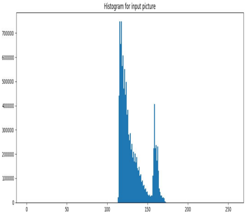
A full example: image histogram stretch

```
uchar8_t min_pixel(const phast::matrix<uchar8_t>* p_img)
{
    return *phast::min_element(p_img->begin_ij(), p_img->end_ij());
}

uchar8_t max_pixel(const phast::matrix<uchar8_t>* p_img)
{
    return *phast::max_element(p_img->begin_ij(), p_img->end_ij());
}

template <typename T, unsigned int policy = phast::get_default_policy()>
struct scaling : phast::functor::func_scal<T, policy>
{
    _PHAST_METHOD scaling(T min, T max) : min_(min), max_(max) {}
    _PHAST_METHOD void operator()(phast::functor::scalar<T>& pixel)
    {
        pixel = static_cast<T>(((255.0 - 0.0) / double(max_ - min_)) * (pixel - min_));
    }

    T min_;
    T max_;
};

phast::matrix<uchar8_t>* scale(phast::matrix<uchar8_t>* p_img, uchar8_t min, uchar8_t max)
{
    phast::for_each(p_img->begin_ij(), p_img->end_ij(), scaling<uchar8_t>(min, max));
    return p_img;
}
```

Task & PHAST

A full example: image histogram stretch

Input: A Temple in the Fushimi Inari Taisha, Kyoto

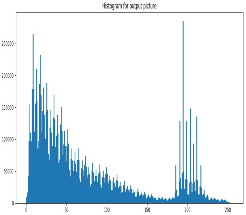


Input image histogram

Task & PHAST

A full example: image histogram stretch

Output: A (*better*) Temple in the Fushimi Inari Taisha, Kyoto



Output image histogram

Summary

.

- Embedded applications need heterogeneous systems from performance and power standpoints
- Programmers of heterogeneous systems need high-productivity frameworks, like PHAST
- Many interesting parallel problems need task support to be conveniently expressed
- PHAST Library can be extended to support task-DAGs with a minimum effort

.

Questions?

Website (Under Construction): https://phast.diism.unisi.it
e-mails: {bartolini, peccerillo}@diism.unisi.it