An Overview of FastFlow: Combining Pattern-Level Abstraction and Efficiency in GPGPUs

Marco Aldinucci, Computer Science Department, University of Turin, Italy
PI of the CUDA research center at University of Turin, Italy
M. Torquati (University of Pisa, Italy), M. Drocco, G. Peretti Pezzi (University of Turin, Italy), C. Spampinato (University of Catania, Italy)

Date: 25 March 2014, San Jose, CA, USA - Presentation S4729
Outline

• Motivational example
  • An effective (and quite universal) image/video denoiser
  • Paradigmatic programming pattern for GPGPUs?

• On patterns for multicore and GPGPUs
  • FastFlow
  • Some performance results
  • A demo
Salt & Pepper and Gaussian noises

- Electronic and signal noise
- S&P Uniform distribution of “saturated” white/black pixels
  - Measured as percentage of affected vs overall pixels
- Gaussian: White additive noise in the frequency domain
  - Affect all pixels, with an additive “white” value distributed as a Gaussian
- Typically restored using statistic filters: e.g. median, median-adaptive
- Not satisfactory for high levels of noise
Two-stage restoring

- progressive-switching/adaptive median
- neural/bayesian networks, fuzzy, ...
- variational
- statistic

Decouple detection decoupled from restoration

- Pixels considered not outliers are not altered by restoration
- False positives impair restoration quality
Two-stage restoring

- progressive-switching/adaptive median
- neural/bayesian networks, fuzzy, …
- variational
- statistic

Statistic detection + variational restoration

- High quality, edge-preserving filtering
- Much more computational demanding, not really viable without parallelism
- Matlab on 256x256 image with 50% of noise requires dozen of minutes
- Stages can be pipelined
Variational De-noising:
an iterative optimisation problem

Try any possible color \( k \) for the pixel, choose \( u \),
the one that minimize the value of \( F(\text{neighb8}(i,j)) \)

\( F(\ldots) \) weight differently
noisy and not noisy pixels

\[
\text{for each } i,j \\
\quad \text{if } (\text{noisyMap}[i,j]) \\
\quad \quad \text{let } N = \text{neighb8}(\text{img},i,j) \\
\quad \quad \text{let } k \text{ in } 0..255 \\
\quad \quad \quad u = \text{argmin}(F(k,N,\text{noiseMap})) \\
\quad \quad \text{img}[i,j] = u \\
\quad \text{while (process not converge)}
\]
Convergence can’t be evaluated with a reduce (involves three iterations, i.e. memory)

- **Noisy Img**
- **Img (k-1)**
- **Img (k)**
- **Img (k+1)**

<table>
<thead>
<tr>
<th>Diff of residuals</th>
<th>Reduce of diffs</th>
<th>Residuals</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>

\[ \sum \Delta^{(k)} = 10 \]
\[ \sum \Delta^{(k+1)} = 3 \]

\[ \frac{\| \sum \Delta^{(k)} \| - \| \sum \Delta^{(k+1)} \|}{\| \sum \Delta^{(k)} \|} < \epsilon \in \mathcal{R} \]
Patterns natural in GPGPUs

... and this week-known from long time

Think In Parallel

- The GPU is a data-parallel processor
  - Thousands of parallel threads
  - Thousands of data elements to process
  - All data processed by the same program
  - SPMD computation model
  - Contrast with task parallelism and ILP

- Best results when you “Think Data Parallel”
  - Design your algorithm for data-parallelism
  - Understand parallel algorithmic complexity and efficiency
  - Use data-parallel algorithmic primitives as building blocks

Data-Parallel Algorithms

- Efficient algorithms require efficient building blocks

This talk: data-parallel building blocks

- Map
- Gather & Scatter
- Reduce
- Scan

but are “standard” and “flat”

data-parallel pattern expressive enough for the problems?
Rationale: patterns are there, but

- Detect-Denoise can be naturally pipelined
- Denoise is a (sort of) map with a stencil
  - Where, $x = <x_1, x_2, ..., x_n>$, \( \text{map } f \ x = <f(x_1), f(x_2), ..., f(x_n)> \)
  - Can be written as a map, but is neither natural nor easy
  - Try to think it without shared memory (halo management)
- Convergence evaluation is map across three iterations and reduce
  - Even more complex to write it as a MapReduce (if not imposible)
- Cholesky LU or C4.5 tree pruning with map, reduce or MapReduce?
**stencilReduce**

- a (low-level) powerful pattern
- presented here, need more validation
- we believe it capture most of the interesting data parallel computations, especially on GPGPUs
- Subsumes: map, reduce, mapReduce
- Programmers do not need to write any line of host code to drive the GPGPU
- D2H/H2D, data feeding, synchronisations, block configurations, …

```c
loop
before (...) stencil<stencilK,reduceK> (data[i], env)
reduce op data
after (...)  Compute on host possibly in parallel on CPU cores
```

- CUDA code

Unified Memory greatly simplify this part

Compute on host possibly in parallel on CPU cores
FastFlow (FF)

- C++ header-only library
  - Portable everywhere exists a C++ compiler (C++11 for some features)
- Provides stream-oriented and data-parallel patterns
  - compositional, efficient
- Accommodate diversity via progressive abstraction layers: if you need a different pattern, do it extending a C++ class
- Multi-core, GPGPUs, distributed
- https://sourceforge.net/projects/mc-fastflow

Parallel applications
  efficient and portable

High-level patterns
  mapreduce, stencil, D&C, ...

Core patterns
  pipeline, farm, feedback

Building blocks
  queues, ff_node, ...

Multi-core and many-core platforms
  Clusters of multicore + many-core

CUDA
Open CL
TCP/IP
IB/OFED
**FF building blocks: nodes and channels**

- **node**
  - channel name or channel
  - threads or processes (non-blocking)
    - can be suspended using a native protocol

- **mi-node**
  - channel names

- **mo-node**
  - channel names

**Building blocks**
- queues, ff_node, ...

**Core patterns**
- pipeline, farm, feedback

**Parallel applications**
- efficient and portable

**High-level patterns**
- mapreduce, stencil, D&C, ...

**Multicore and many-core platforms**
- Clusters of multicore + many-core

**Building blocks**
- CUDA, TCP/IP, IB/OFED

**FF bound shmem FIFO channel**
- Single-Producer-Single-Consumer
- lock-free fence-free queue

**FF unbound shmem FIFO channel**
- Single-Producer-Single-Consumer
- lock-free fence-free queue

**Distributed zero-copy channel**
- 0MQ/TCP or native IB/OFED

**Threads or processes**
- non-blocking

**Shmem channels**
- communicate pointers in a message passing style

**Xeon E7-4820 @2.0GHz Sandy Bridge**

**Figure**
- MVAPICH ~ 190ns
- faster and more scalable than CAS/test-and-set implement.

Semantics of the node: dataflow activation

```cpp
class mynode: public ff_node {
public:
    int svc_init() {
        /* after constructor - running as a thread */
        return 0;
    }

    void * svc(void * task) {
        int * t = (mytask_t *) task;
        // do something on task
        cout << "mynode " << ff_node::get_my_id()
             << " received task " << t->payload << "\n";
        return task;
    }

    void svc_end() {
        /* before destructor - running as a thread */
    }
};
```

mynode is created as a standard C++ class extending ff_node
Semantics of the node: dataflow activation

After class construction
mynode is turn into a thread

Nodes are not tasks (as in TBB),
they are executors

The svc_init() method is executed

```cpp
class mynode: public ff_node {
    public:
        int svc_init() {
            /* after constructor - running as a thread */
            return 0;
        }
        void * svc(void * task) {
            int * t = (mytask_t *) task;
            // do something on task
            cout << "mynode " << ff_node::get_my_id() << " received task " << t->payload << "\n";
            return task;
        }
        void svc_end() {
            /* before destructor - running as a thread */
        }
};
```
Semantics of the node: dataflow activation

```cpp
class mynode: public ff_node {
public:
    int svc_init() {
        /* after constructor - running as a thread */
        return 0;
    }

    void * svc(void * task) {
        int * t = (mytask_t *) task;
        // do something on task
        cout << "mynode " << ff_node::get_my_id()
             << " received task " << t->payload << "\n";
        return task;
    }

    void svc_end() {
        /* before destructor - running as a thread */
    }
};
```

the node enters in a infinite loop

1. get a task from input channel (i.e. a pointer)
2. execute svc method
3. put a task the output channel (i.e. a pointer)

`svc()` might output more tasks via `ff_send_out` call (not shown)

The node terminate on returning a NULL pointer
class mynode: public ff_node {
public:
  int svc_init() {
    /* after constructor - running as a thread */
    return 0;
  }

  void * svc(void * task) {
    int * t = (mytask_t *) task;
    // do something on task
    cout << "mynode " << ff_node::get_my_id()
    << " received task " << t->payload
    "\n";
    return task;
  }

  void svc_end() {
    /* before destructor - running as a thread */
  }
}
FF core patterns: pipe, farm, feedback.

They are streaming networks, not task graphs.

**Core patterns:**
- Pipeline
- Farm
- Feedback

**High-level patterns:**
- MapReduce
- Stencil
- D&C

**Parallel applications:**
- Efficient and portable

**Multicore and many-core platforms:**
- Clusters of multicore + many-core

**Building blocks:**
- Queues, `ff_node`
- `opencl`
- `cuda`
- `tcp/ip`
- `ib/ofed`

**Pipeline**

```
E ----> W1 ----> W2 ----> Wn ----> E
```

**Farm with feedback**

```
E ----> W1 ----> W2 ----> Wn ----> E
```

**Pipeline with feedback**

```
S1 ----> S2 ----> ... ----> Sn
```

**Farms with feedback**

(e.g. D&C and Master-Workers)

**Or any composition of them**
GPGPUs

- Fill stencilReduce methods with CUDA kernel code
  - No CUDA-related host code at all need to be written
- Possibly nest stencilReduce into another pattern
  - e.g. farm to use many GPGPUs
    - the async copy engine is automatically used via CUDA streams
  - Helpful to mix threading (or distributed) with GPGPUs
  - If you already have your standard host+CUDA code just copy-paste into a svc( ) method
FF core patterns: rationale

- farm: process in parallel independent tasks (e.g. C++ objects)
  - true dependencies are enforced only along edges of the graph
  - workers might synchronise (e.g. w locks/atomics), synchronisation in the business code

- farm, pipeline and feedback (to build cyclic networks) are enough to write all other patterns

- Think to GPGPUs design
  - They be though as machines to compute a map, reduce, stencil, …
  - … but in hardware they are built as a farm that dispatches independent blocks onto multiprocessors (+global memory)
FF high-level patterns

- Proposed as code annotations
  - Similarly to openMP, openacc, ...
  - used to generate a graph at the core pattern level

- Examples
  - parallel_for
  - map, reduce, MapReduce, … (targeting GPGPUs)
  - and as many as you want: developing a new pattern is just developing a new class
Example: map (derived from stencilReduce)

- 2 GPGPUs on the same platform
  - nest a stencilReduce, map, reduce … into a (host) farm with 2 workers

- offload code onto distributed GPGPUs
  - nest a stencilReduce, map, reduce … into a (host) distributed farm with 2 workers
  - data serialisation is up to user, the framework just provides callback to do it

- In both cases
  - be sure that tasks are independent (otherwise you need another pattern)
Example: map (derived from stencilReduce)

```cpp
FFMAPFUNC(mapF, unsigned int, in, return (in * 2));

class cudaTask: public baseCUDATask<unsigned int> {
public:
    void setTask(void* t) {
        if (t) {
            cudaTask *t_ = (cudaTask *) t;
            setInPtr(t_->in);     // match of data pointer H2D
            setOutPtr(t_->in);    // match of data pointer D2H
            setSizeIn(inputsize); // needed without Unified memory
        }
    }
    unsigned int *in, *out;
};

main () {
    ...
    // put the input in task->in
    FFMAPCUDA(cudaTask, mapF) *myMap = new FFMAPCUDA(cudaTask, mapF)(*task);
    myMap->run_and_wait_end();
    // result is in task->out
    ...
}
```

This is CUDA code

Simple in this case, but any CUDA code is valid here

It will be compiled with NVCC

This is just a name to distinguish this kernel

This is a macro

For multicore we use C++11 lambda. Theoretically possible to use Lambda for kernel code? (maybe with UnifiedMemory)
Example: Qt-mandelbrot (from Qt samples)

Original (sequential)

```
... const int Limit = 4;
bool allBlack = true;
if (restart) break;
if (abort) return;

for (int y = -halfHeight; y < halfHeight; ++y) {

    uint *scanLine =
        reinterpret_cast<uint *>(image.scanLine(y + halfHeight));
    double ay = centerY + (y * scaleFactor);
    ...
}
private:
...
```

FastFlow (parallel)

```
...
const int Limit = 4;
bool allBlack = true;
if (restart) break;
if (abort) return;

pf_det.parallel_for(-halfHeight, halfHeight, 1, halfHeight,
    [&](const long y) {
    uint *scanLine =
        reinterpret_cast<uint *>(image.scanLine(y + halfHeight));
    double ay = centerY + (y * scaleFactor);
    ...
    }
private:
ParallelFor pf_det;
...
```
“Demo”

Don’t forget Murphy’s law
Performance (multicore)

Bowtie2 tool (DNA mapping)

- Bt2
- Bt2-int
- Bt2FF
- Bt2FF-pin
- Bt2FF-pin+int

Speedup

Dataset
- lane2-CTL-qseq
- SRR568427
- SRR534301

Smith-Waterman (SSE2) against OpenMP, Cilk, TBB

Cholesky LU against PLASMA

Original version: pthreads + test&set spinlocks

FF differs no more than 30 lines of code from the original on several thousands (including memory affinity management)
Performance (CPUs + GPGPUs)

Video frames 768x512

Nehalem 32 cores + 1 K20

- 32 cores + K20
- 32 core

S&P 10%  S&P 50%  Gauss V10  Gauss V100

SandyBridge 16 cores + 2 Tesla M2090

- 16 cores + 2xM2090
- 16 cores + 1xM2090
- 16 cores

S&P 10%  S&P 50%  Gauss V10  Gauss V100
FastFlow: a framework for research

- Open-minded patterns
  - A pattern is missing? Not happy of the implementation? Modify it extending a class …
  - Multicore, GPGPUs, distributed under the same theoretical umbrella. No compilers to modify (!)
  - Non expert programmers does not need to deal with synchronisations and data copies, just select patterns
  - Productivity: portability, porting of legacy applications

- Comparable or better with OpenMP and TBB on fine grain
  - Comparable with OpenMP on data-parallel - really fast on streaming (especially very high-frequency)

- Entirely C++ (C++11), minimalistic design, solid enough to test new solutions
  - E.g. FastFlow lock-free parallel memory allocator extend with CUDA UnifiedMemory
  - Graceful optimisation curve: does not inhibits the optimisation of your kernel

- Main platform is Linux, but works almost everywhere exist a C++ compiler
  - MacOS, Win — x86, x86_64, Xeon Phi, PPC, Arm, Tilera, NVidia (CUDA and OpenCL) — gcc, clang, icc, nvcc
Coding details in the tomorrow talk:

**S4585 - FastFlow: Combining Pattern-Level Abstraction and Efficiency in GPGPUs**

Day: Wednesday, 03/26 - 15:00 - 15:50 - Room LL21A

Univ. of Turin: M. Aldinucci, G. Peretti, A. Secco, F. Tordini, M. Drocco, C. Misale

Univ. of Pisa: M. Torquati, M. Danelutto

EU-FP7 - 3.5M€
IMPACT
EU-FP7 - 3.7M€
Unito - 440K€
EU-FP7 Network of Excellence

https://sourceforge.net/projects/mc-fastflow/

3-years researcher positions available: deadline May 5, 2014

http://www.di.unito.it/train2move/