
# A Parallel Edge Preserving Algorithm for Salt and Pepper Image Denoising

**Marco Aldinucci - Uni. of Torino, Italy**
*Parallel and distributed computing research cluster*

Massimo Torquati - Uni. Pisa, Italy
Maurizio Drocco - Uni. of Torino, Italy
Concetto Spampinato - Uni. of Catania, Italy
Simone Palazzo - Uni. of Catania, Italy

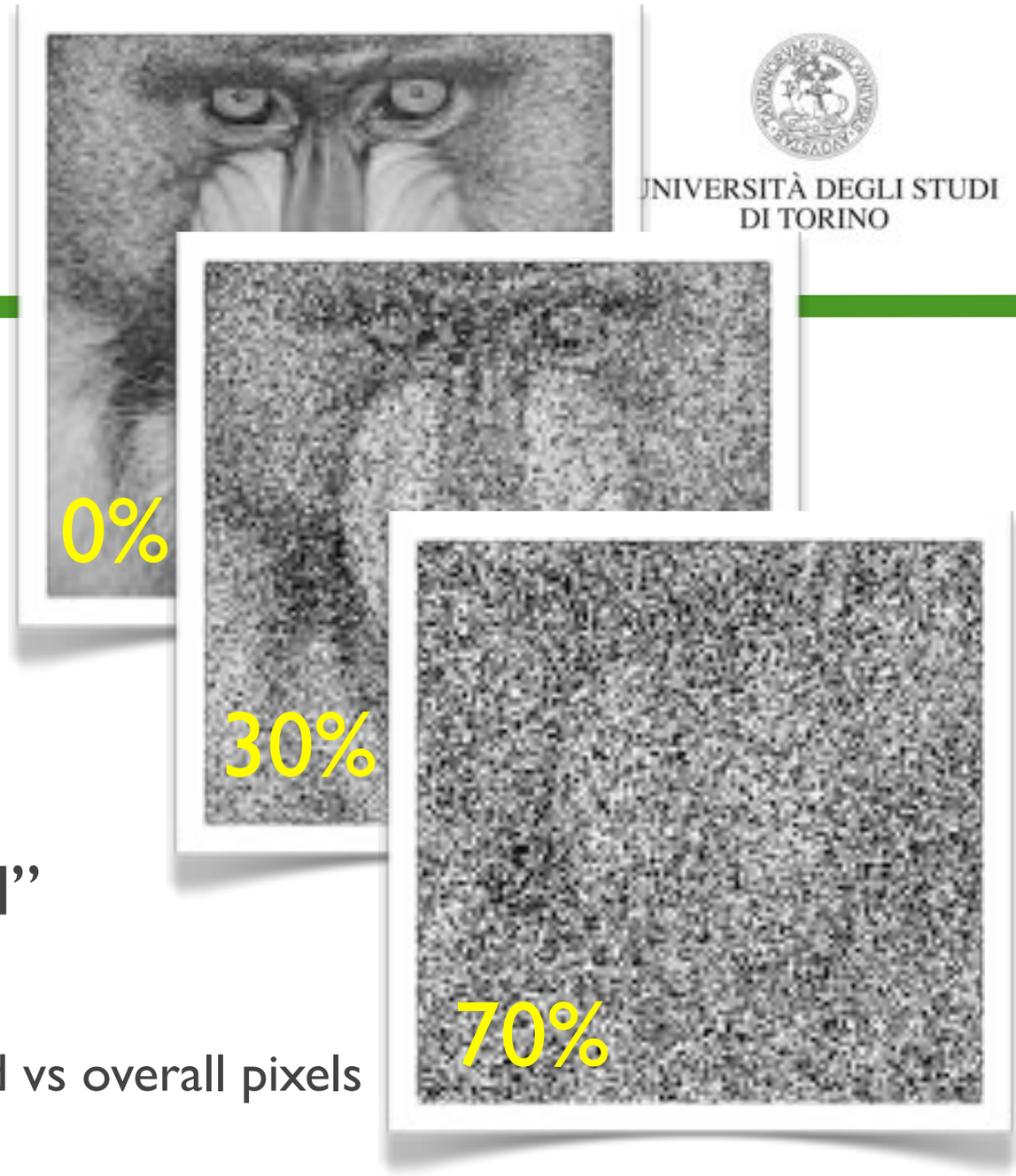
# Outline

UNIVERSITÀ DEGLI STUDI
DI TORINO

- A two-phase image/video denoiser
- FastFlow
  - A programming model (and a library)
  - Fast (<10 nS) core-to-core lock-free messaging
  - Fast zero-copy distributed messaging
  - Supporting multi-core and accelerators
- two-phase denoiser: parallel implementation
  - clean salt-and-pepper noise up to 90%
  - demo

# A two-phase image denoiser for salt & pepper

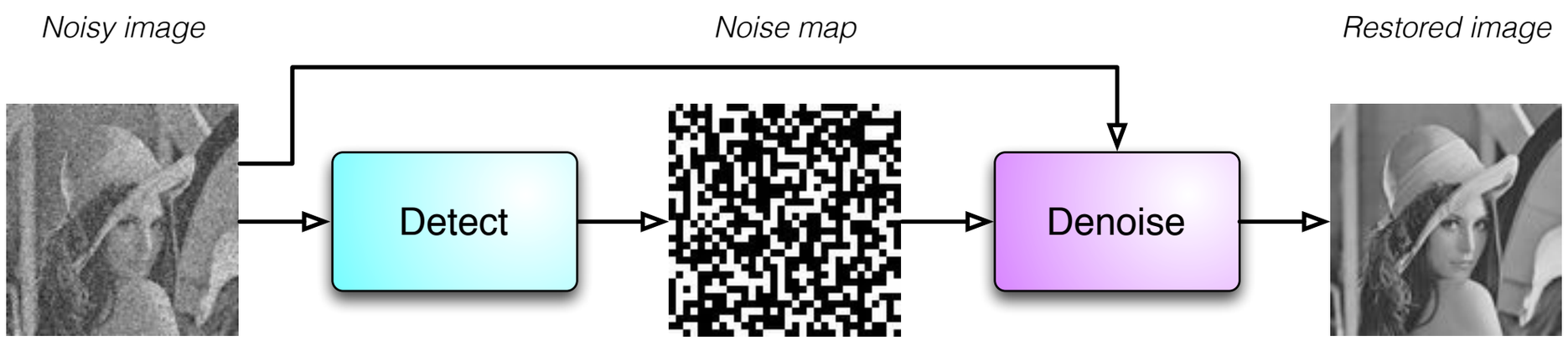
# Salt&Pepper noise




- Salt&Pepper noise
  - Electronic and signal noise
  - Uniform distribution of "saturated" white and black pixels
    - Often measured as percentage of affected vs overall pixels
- Typically restored using statistic filters
  - e.g. median, median-adaptive
  - Not satisfactory for high levels of noise, also good pixels are filtered (image is smoothed)

# Improved S&P filtering

UNIVERSITÀ DEGLI STUDI DI TORINO




Impulse detector
- progressive switching median
- adaptive median
- neural/bayesian networks
- fuzzy/neuro-fuzzy logic

Restoration
- median
- variational (Nikolova)

## ✳ Performance

- ✦ Improved, if detection faster than restoration
- ✦ Rationale: efficient detection + high-quality denoising
  - they can be pipelined and independently parallelized

# Variational method

UNIVERSITÀ DEGLI STUDI DI TORINO

- Iterative optimization problem
  - Theoretically and practically more expensive than statistical filtering
    - Matlab w 256x256 image with 50% of noise needs hours of computation time
  - Many local minimum, fixed point criteria for termination not trivial
    - Quasi-Newtonian methods with PSNR and MAE variation can be used

$$\min_{u \in N} F(u) = \alpha \int R(u) + \beta \int D(u, d)$$

*regularization term* *data fidelity term*

$$F_d|_N(u) = \sum_{(i,j) \in N} [|u_{i,j} - d_{i,j}| + \frac{\beta}{2}(S_1 + S_2)]$$

$$S_1 = \sum_{(m,n) \in V_{i,j} \cap N} 2 \cdot \varphi(u_{i,j} - d_{m,n})$$

$$S_2 = \sum_{(m,n) \in V_{i,j} \cap N^c} \varphi(u_{i,j} - u_{m,n})$$

$$\varphi(t) = |t|^\alpha \text{ with } 1 < \alpha \leq 2$$

# Results: quality (1024x1024 grayscale)

UNIVERSITÀ DEGLI STUDI DI TORINO

Original
Baboon standard
test image
1024x1024

10% impulsive noise

50% impulsive noise

90% impulsive noise

Restored

PNSR 43.29dB MAE 0.35

PNSR 32.75dB MAE 2.67

PNSR 23.4 MAE 11.21

noise 70%

noise 70%

restored

noise 70%

restored

original

# Results: quality (256x256 grayscale)

UNIVERSITÀ DEGLI STUDI DI TORINO

*PSNR and MAE on standard benchmarks*

| | Lena — Noise level | | | | | Peppers — Noise level | | | | | Bridge — Noise level | | | | | Baboon — Noise level | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 10% | 30% | 50% | 70% | 90% | 10% | 30% | 50% | 70% | 90% | 10% | 30% | 50% | 70% | 90% | 10% | 30% | 50% | 70% | 90% |
| PSNR | 41.96 | 35.76 | 32.56 | 29.47 | 24.43 | 42.27 | 35.49 | 31.57 | 28.23 | 23.12 | 36.61 | 31.50 | 28.11 | 25.23 | 21.22 | 35.39 | 29.73 | 26.94 | 24.61 | 22.06 |
| MAE | 0.36 | 1.21 | 2.27 | 3.85 | 8.82 | 0.30 | 1.10 | 2.18 | 3.94 | 9.90 | 0.82 | 2.61 | 4.85 | 8.06 | 15.01 | 0.94 | 3.14 | 5.63 | 8.87 | 13.96 |

*Comparison with single phase Chan's variational method*

| | Method | Lena — Noise level | | | | Bridge — Noise level | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 30% | 50% | 70% | 90% | 30% | 50% | 70% | 90% |
| PSNR | Chan's | 35.20 | 32.21 | 29.03 | 22.46 | 30.29 | 27.91 | 25.00 | 19.02 |
| PSNR | Our | 35.76 | 32.56 | 29.47 | 24.43 | 31.50 | 28.11 | 25.23 | 21.22 |
| MAE | Chan's | 1.5 | 2.97 | 4.2 | 12.45 | 3.75 | 5.30 | 8.10 | 21.25 |
| MAE | Our | 1.21 | 2.27 | 3.85 | 8.82 | 2.61 | 4.85 | 8.06 | 15.01 |

# On the freedom of low-level methods

UNIVERSITÀ DEGLI STUDI DI TORINO

- MPI, thread synchronizations, and CUDA can be used to parallelize almost everything
- They give you lot freedom, you can write almost everything, move you data in any fashion
- Eventually *"they are as a "car", you can drive where you like, when you want, ..."*
  - (cit. D.K. Panda, leader of the MPI-MVAPICH group)

# On the freedom of low-level methods

UNIVERSITÀ DEGLI STUDI DI TORINO

## Design your algorithm

# On the freedom of low-level methods

UNIVERSITÀ DEGLI STUDI DI TORINO

## ... and running it!

EU FP7 NoE

EU FP7 Strep 3.5 M€

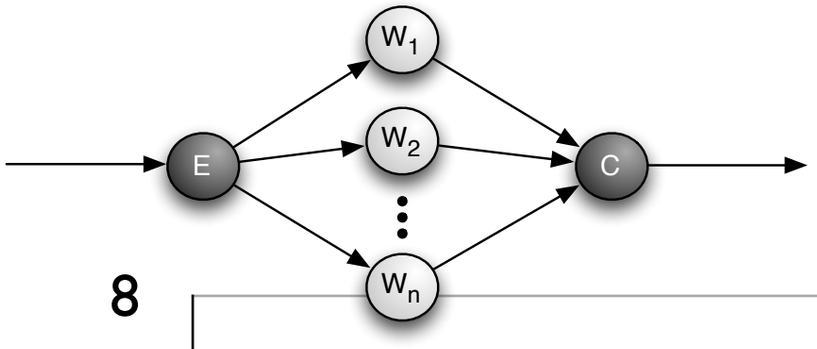
http://mc-fastflow.sourceforge.net/

# FastFlow


- C++ pattern-based framework, open-source LGPL
- A tiny, lightweight & open research framework for HPC
  - 5K lines of code
- 3 years old - over 8K downloads - over 40K contacts
- x86/PPC/ARM + Linux/Mac/Win/iOS
- Multicore, GPGPU, distributed (TCP & Infiniband)

# Lock vs Nonblocking CAS (fine grain 0.5 µS)





# FastFlow (multicore)

**Applications on multicore, many-core**
*Efficient and portable - designed with high-level patterns*

**FastFlow**

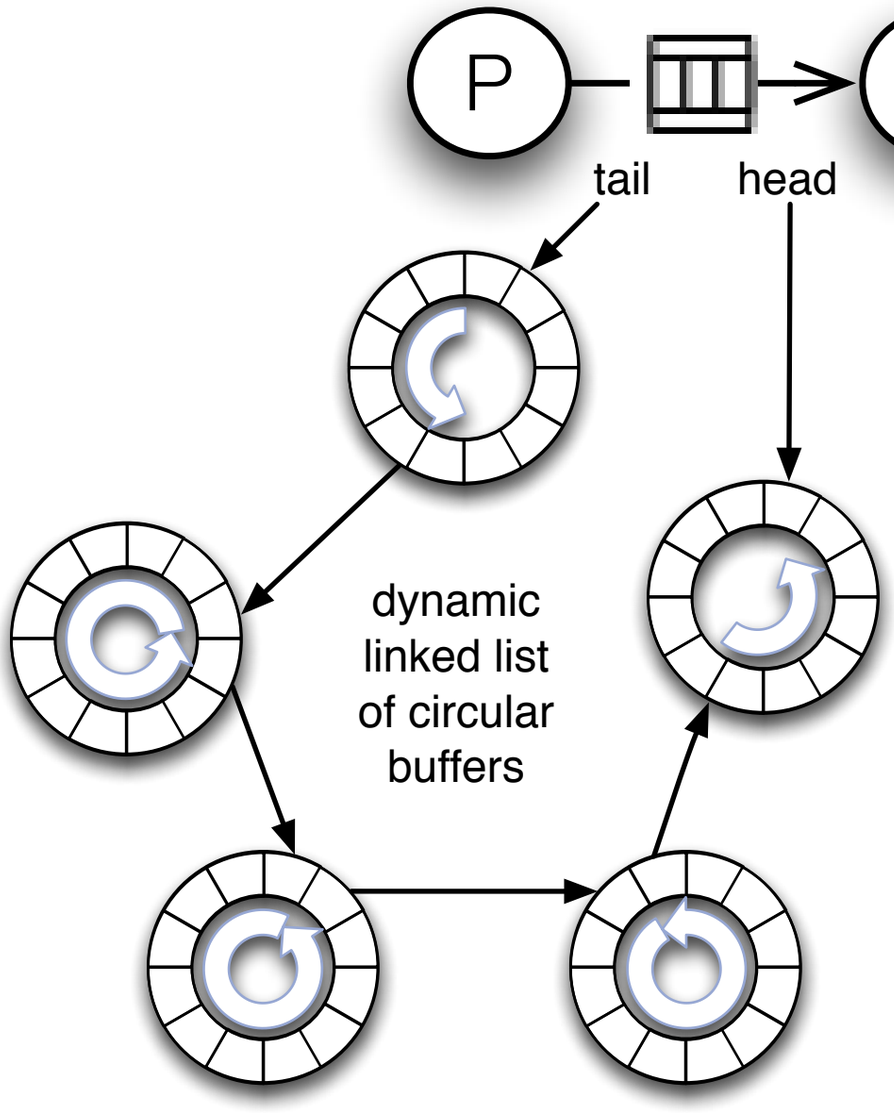
**Streaming network patterns**
*Skeletons: pipeline, map farm, reduce, D&C, ...*

**Arbitrary streaming networks**
*Lock-free SPSC/MPMC queues + FF nodes*

**Simple streaming networks**
*Lock-free SPSC queues + threading model*

**Multicore and manycore**
*SMP: cc-UMA & cc-NUMA*

# Layer 1: Simple streaming networks




```
1  int  size  = N; //SPSC size
2  bool push(void* data) {
3    if (buf_w->full()) {
4      SPSC* t = pool.next_w();
5      if (!t) return false;
6      buf_w = t;
7    }
8    buf_w->push(data);
9    return true;
10 }
11 bool pop(void** data) {
12   if (buf_r->empty()) {
13     if (buf_r == buf_w) return false;
14     if (buf_r->empty()) {
15       SPSC* tmp = pool.next_r();
16       if (tmp) {
17         pool.release(buf_r);
18         buf_r = tmp;
19       }
20     }
21   }
22   return buf_r->pop(data);
23 }
```

```
25 struct Pool {
26   dSPSC inuse;
27   SPSC cache;

29   SPSC* next_w() {
30     SPSC* buf;
31     if (!cache.pop(&buf))
32       buf = allocateSPSC(size);
33     inuse.push(buf);
34     return buf;
35   }
36   SPSC* next_r() {
37     SPSC* buf;
38     return (inuse.pop(&buf)? buf : NULL);
39   }
40   void release(SPSC* buf) {
41     buf->reset(); // reset pread and pwrite
42     if (!cache.push(buf))
43       deallocateSPSC(buf);
44   }
45 }
```

Fig. 3: Unbounded wait-free uSPSC queue implementation.

M. Aldinucci, S. Campa, M. Danelutto, M. Torquati. An Efficient Synchronisation Mechanism for Multi-Core Systems. EuroPar 2012.

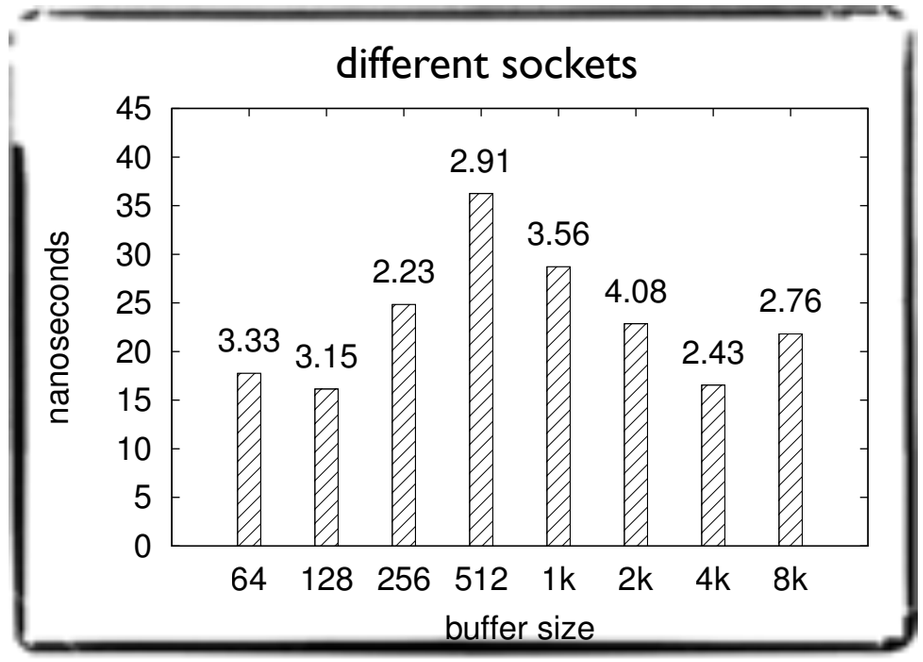
# Layer 1: Simple streaming networks

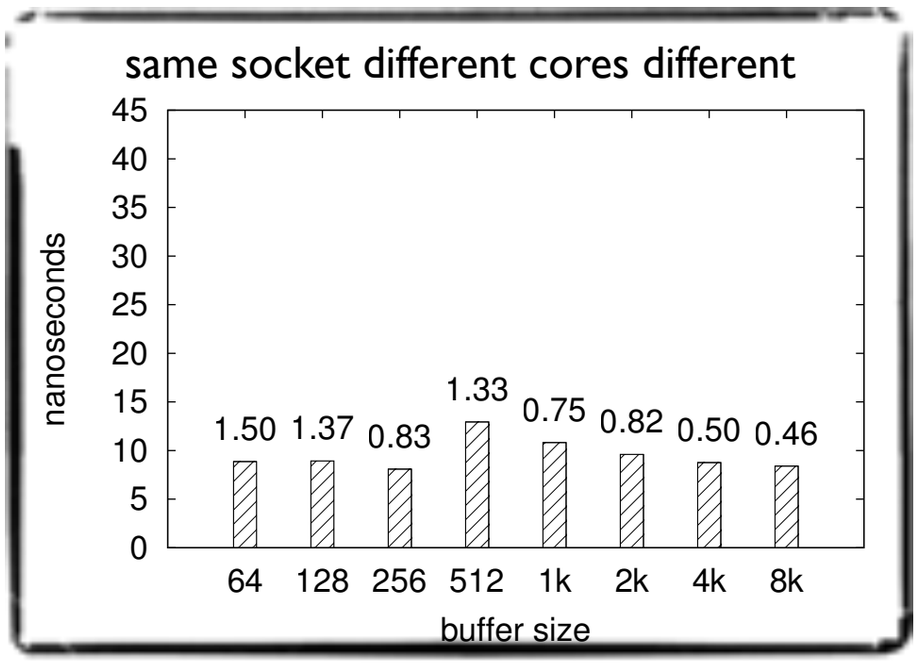
UNIVERSITÀ DEGLI STUDI DI TORINO

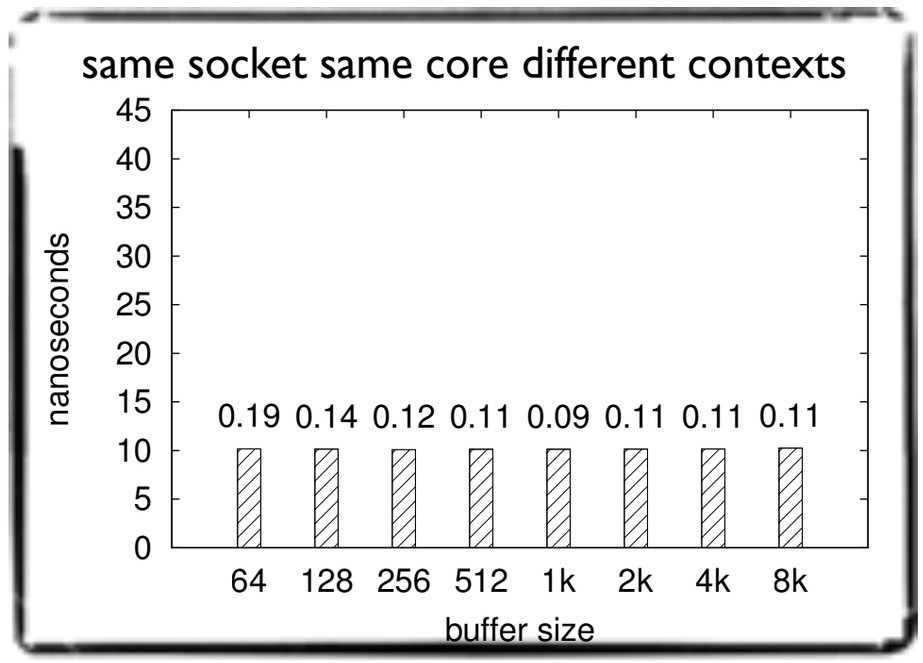
4 sockets x 8 core x 2 contexts

Xeon E7-4820 @2.0GHz Sandy Bridge
18MB L3 shared cache, 256K L2

**MPI is ~190 ns (D.K. Panda)**

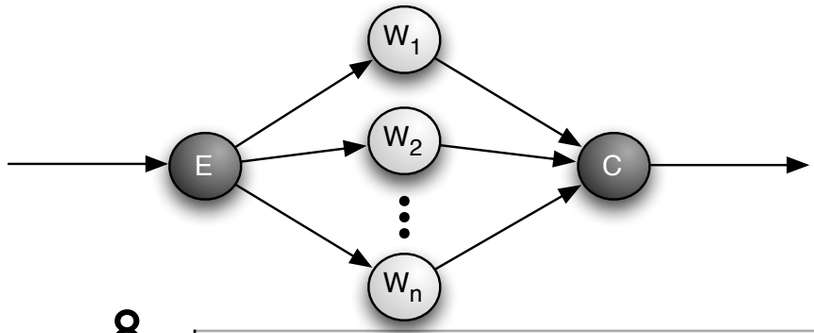


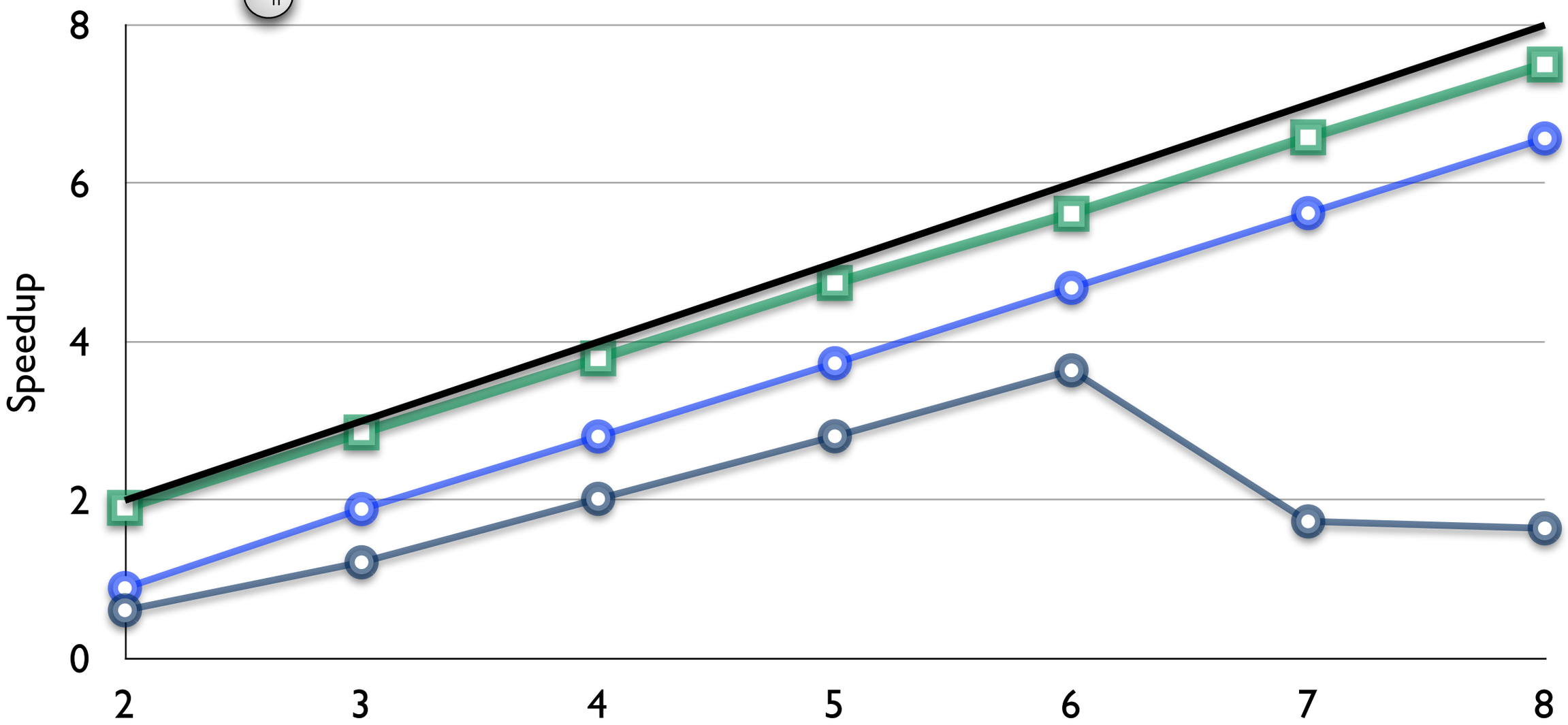



# Lock vs CAS vs SPSC FastFlow (5 µS)

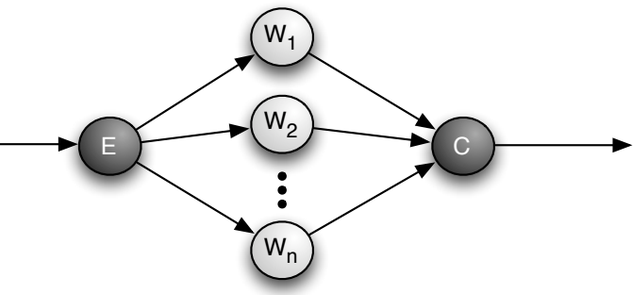
UNIVERSITÀ DEGLI STUDI DI TORINO

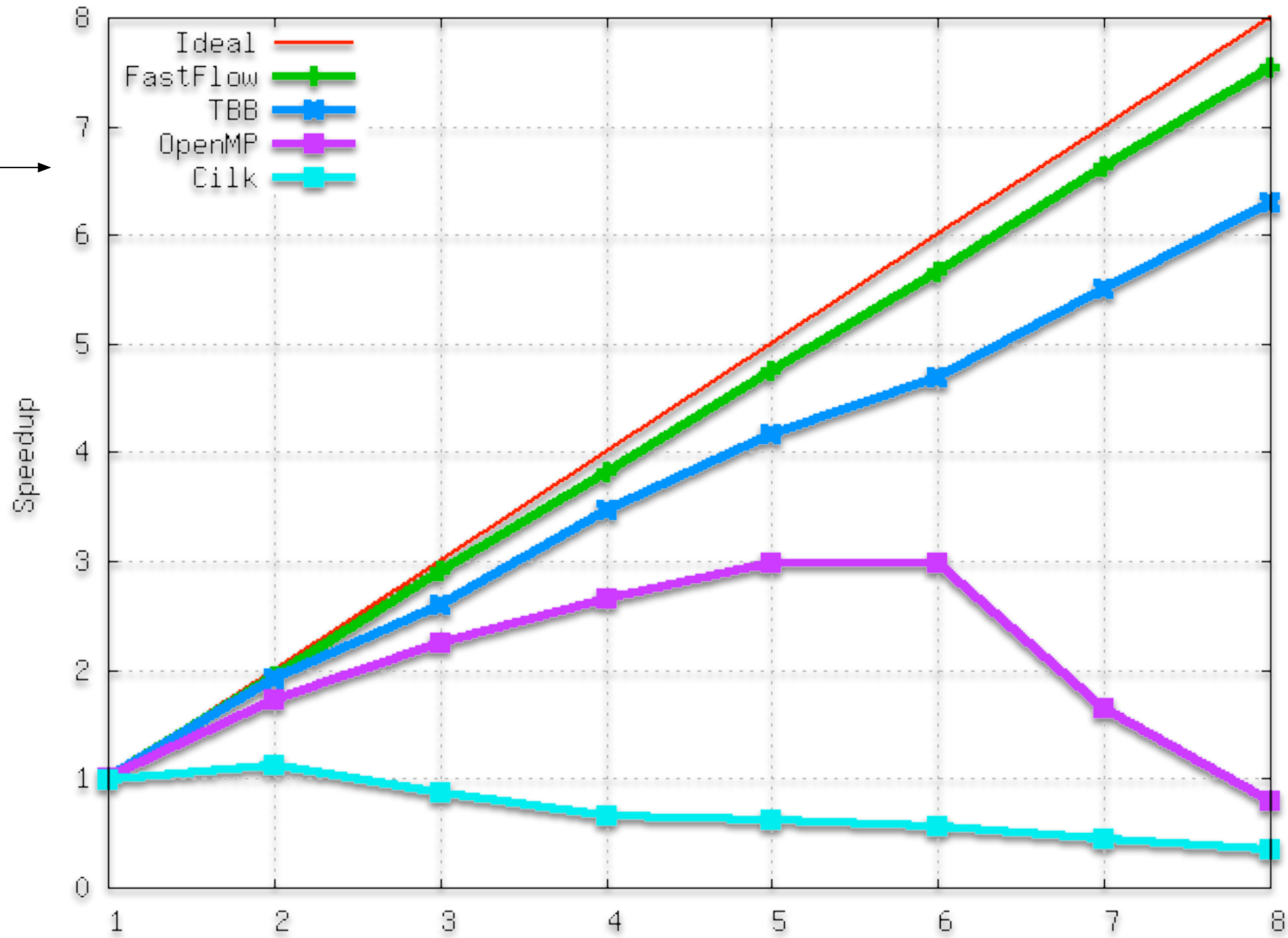


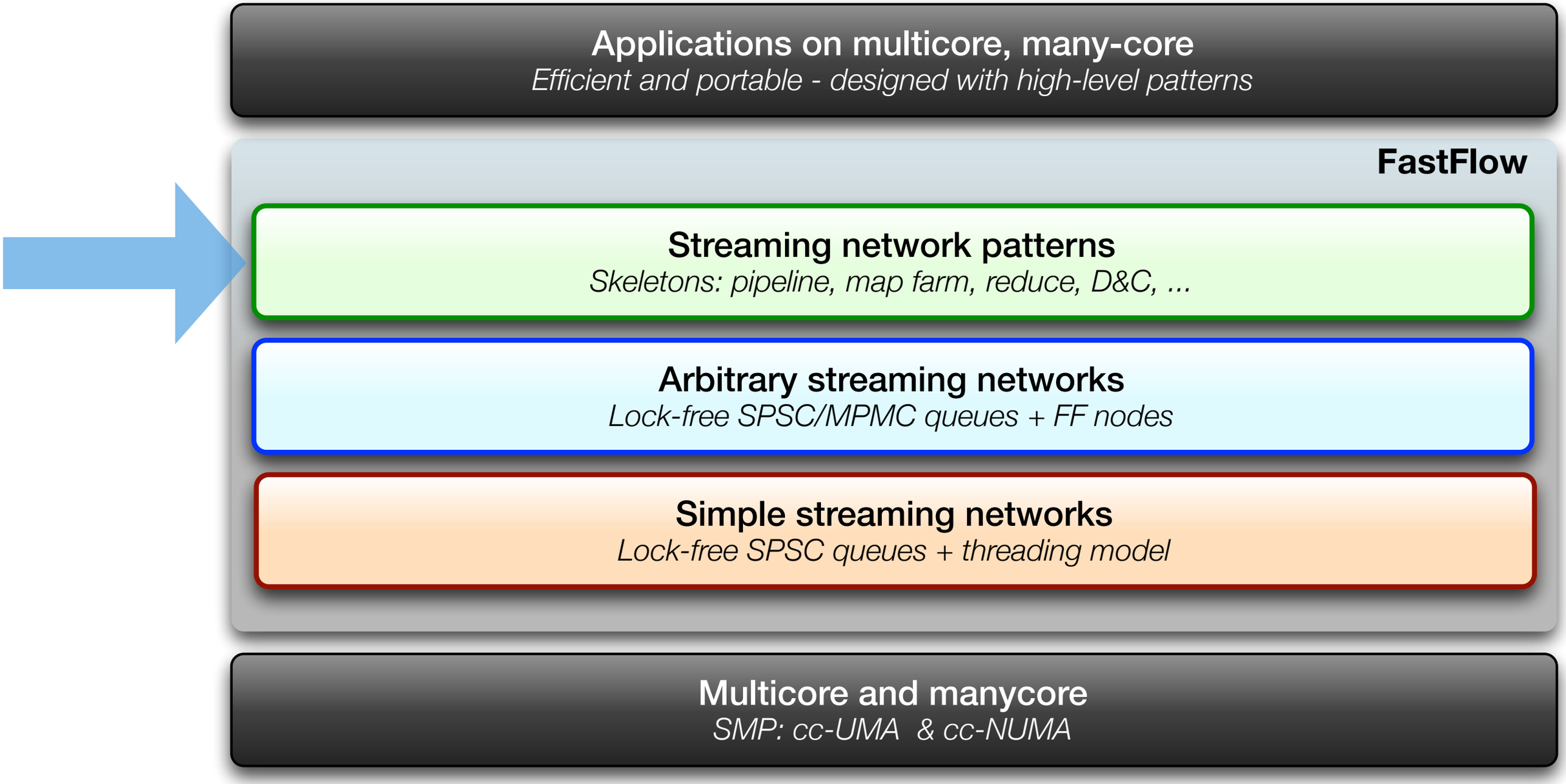
# Medium grain (5 µS workload)

UNIVERSITÀ DEGLI STUDI DI TORINO



# FastFlow (multicore)



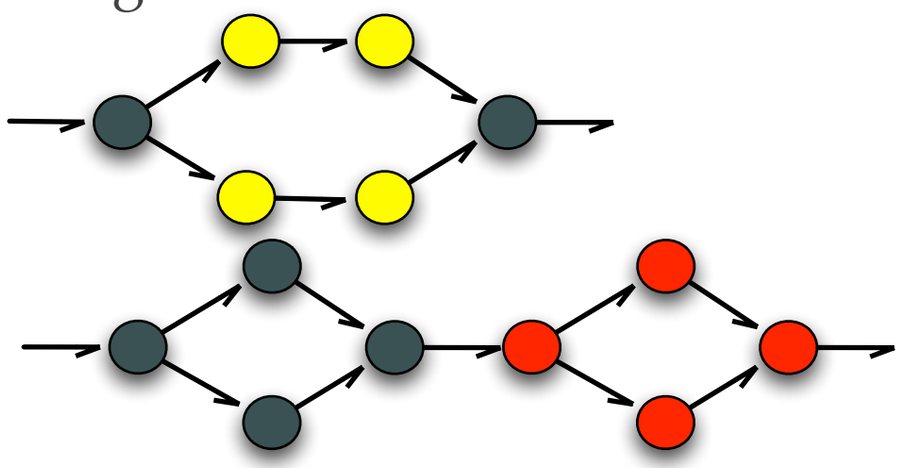
# Taxomy

UNIVERSITÀ DEGLI STUDI
DI TORINO

- Data Parallel
  - is a method for parallelizing a single task by processing independent data elements of this task in parallel. The flexibility of the technique relies upon stateless processing routines implying that the data elements must be fully independent. Data Parallelism also supports Loop-level Parallelism where successive iterations of a loop working on independent or read-only data are parallelized in different flows-of-control and concurrently executed.
- Task Parallel
  - is explicit in the algorithm and consists of running the same or different code on different executors (cores, processors, machines, etc.). Different flows-of-control (threads, processes, etc.) may communicate with one another as they work. Communication usually takes place to pass data from one thread to the next as part of the same data-flow graph.
- Stream Parallel
  - can be used when there exists a partial or total order in a computation. By processing data elements in order, local state may be maintained in each filter.

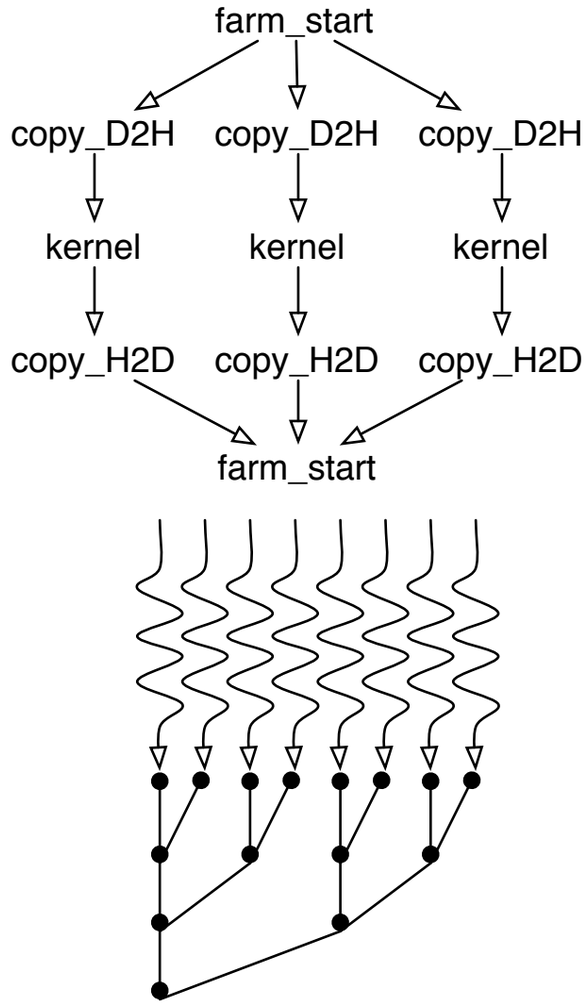
# Layer 3: streaming networks patterns

- Composition via C++ template meta-programming
  - CPU: Graph composition
  - GPU: CUDA streams
  - CPU+GPU: offloading
- farm{ pipe }
- pipe(farm, farm)
- pipe(map, reduce)
- ....



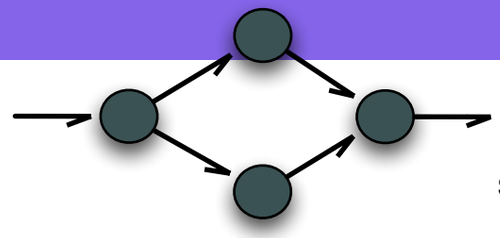
Multi-core

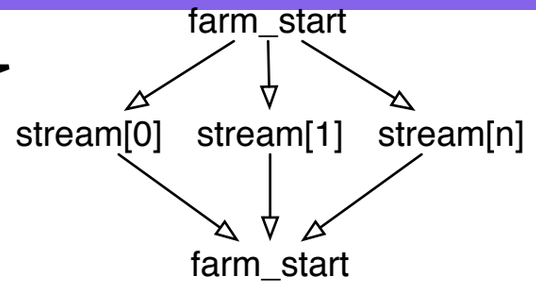
GPGPU

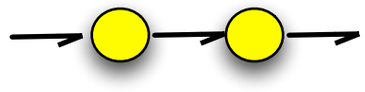


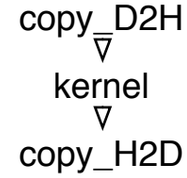
# Layer 3: streaming networks patterns

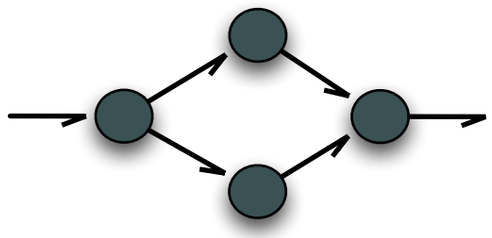
- farm
  - on CPU - master-worker - parallelism exploitation
  - on GPU - CUDA streams - automatic exploitation of asynch comm
- pipeline
  - on CPU - pipeline
  - on GPU - sequence of kernel calls or global mem synch
- map
  - on CPU - master-worker - parallelism exploitation
  - on GPU - CUDA SIMT - parallelism exploitation
- reduce
  - on CPU - master-worker - parallelism exploitation
  - on GPU - CUDA SIMT (reduction tree) - parallelism exploitation
- D&C
  - on CPU - master-worker with feedback - // exploitation
  - on GPU - working on it, maybe loop+farm

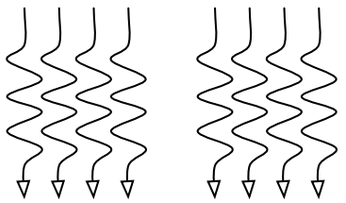


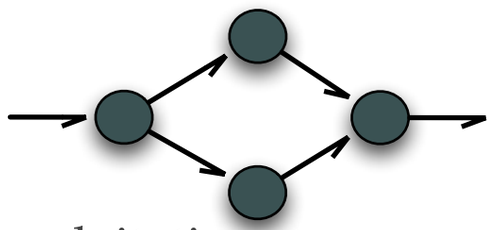


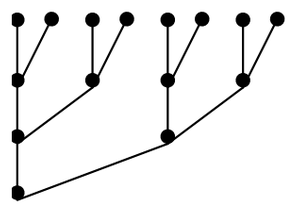



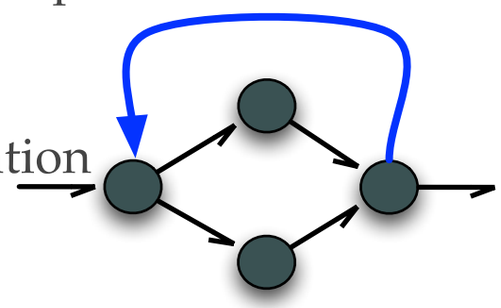


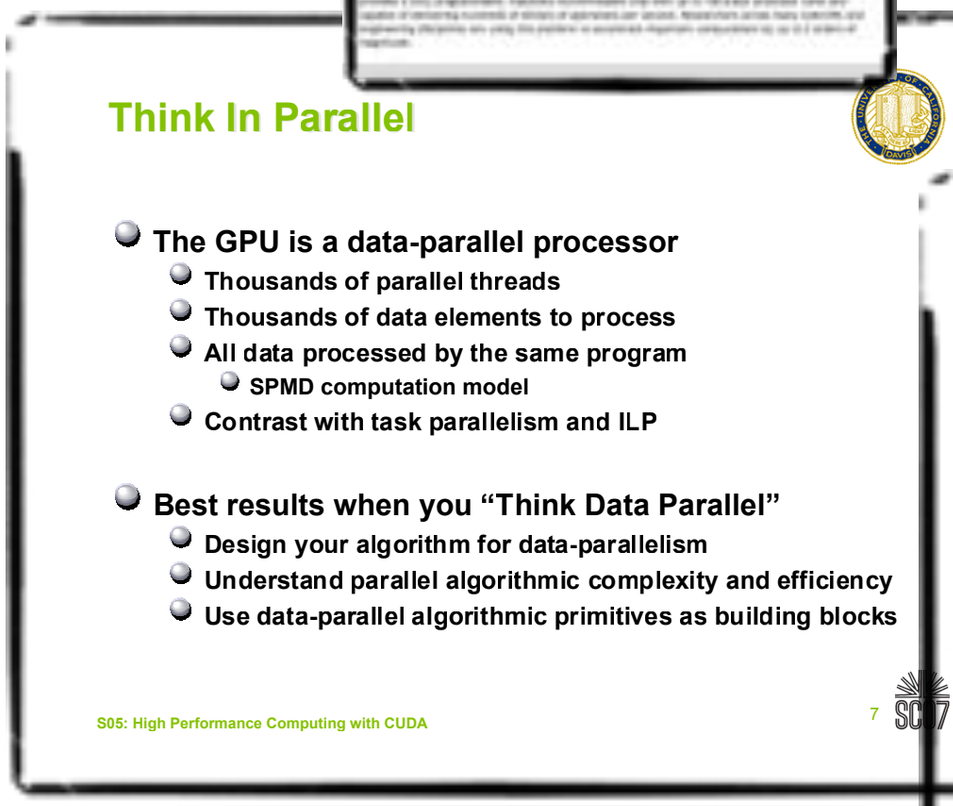


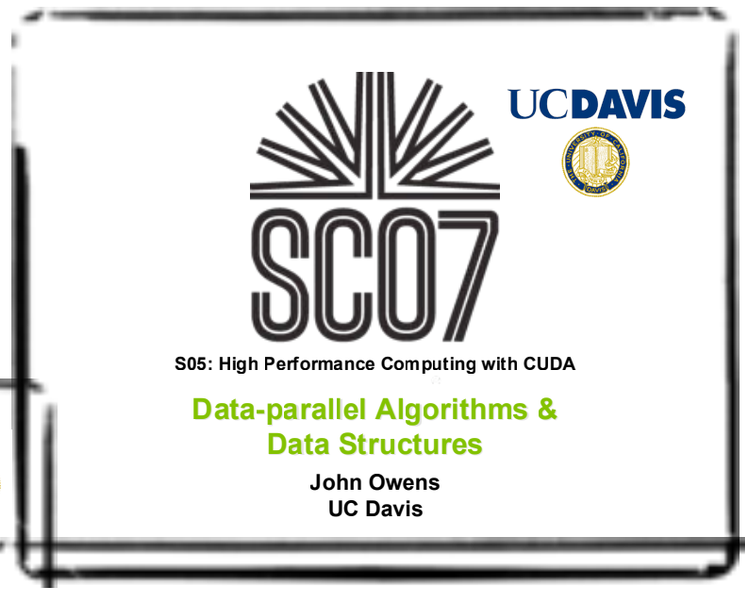


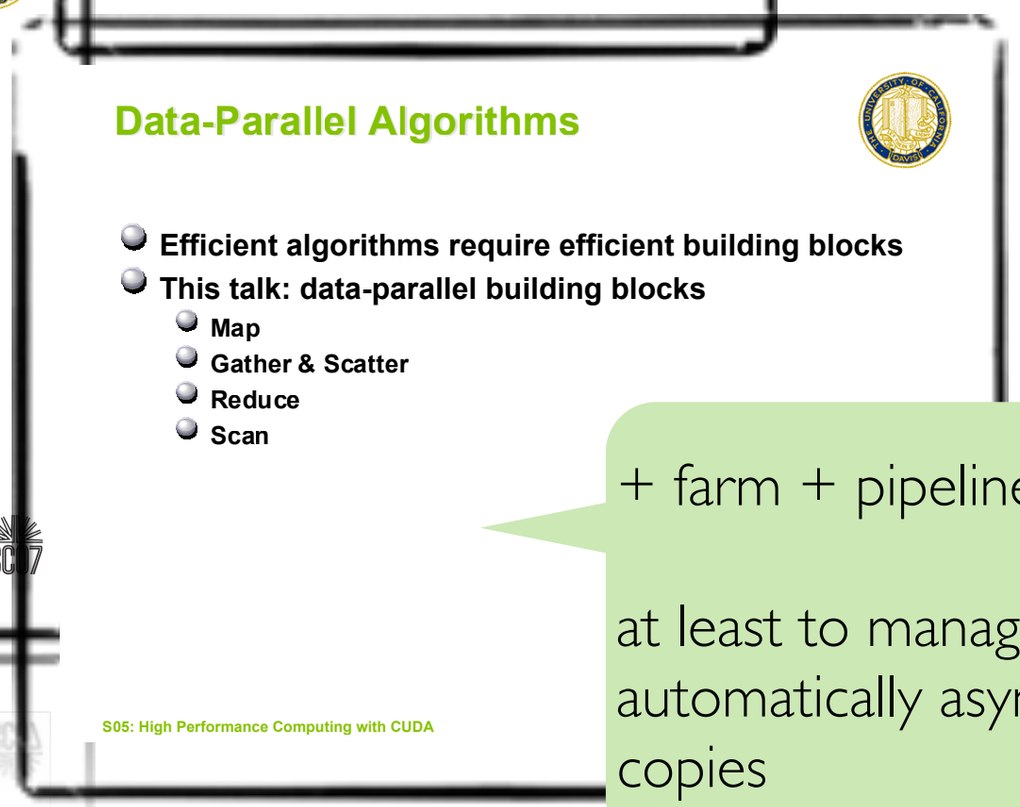




# At the bottom line, also on GPUs, it is matter of abstracting & engineering well-known concepts

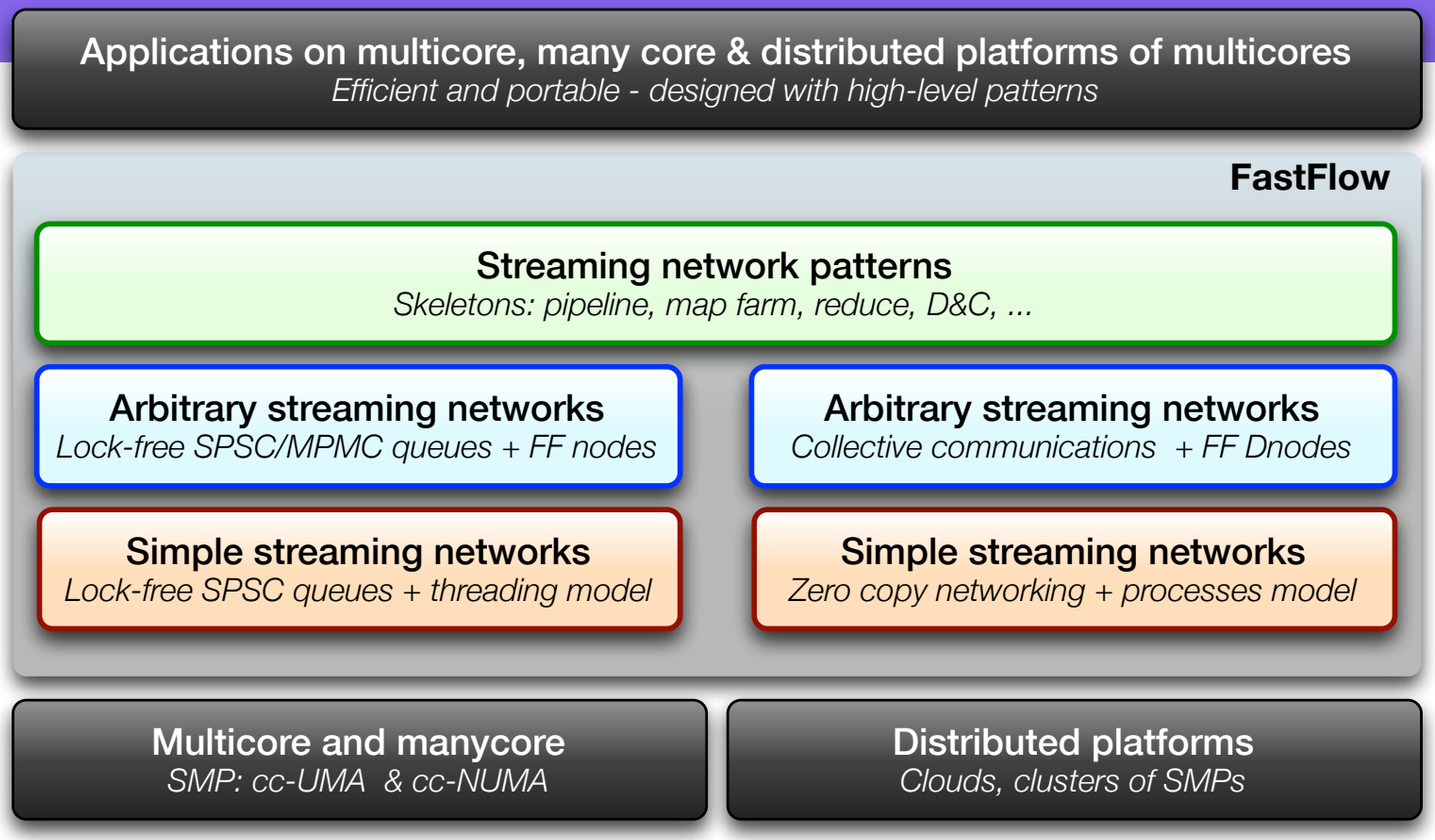




+ farm + pipeline + D&C

at least to manage automatically asynchronous copies

# + distributed




- network channels
  - P2P or collective
  - used as frontier node of streaming graph
  - can be used to merge graphs across distributed platforms
- No changes to programming model
  - when passing pointers data is serialised

M. Aldinucci, S. Campa, M. Danelutto, M. Torquati, P. Kilpatrick. Targeting distributed systems in FastFlow. CGW/EuroPar 2012

# Pattern-based approach: rationale

- Abstract parallelism exploitation pattern by parametric code
  - E.g. higher order function, code factories, C++ templates, ...
  - Can composed and nested as programming language constructs + offloading
  - Stream and Data Parallel
- Platform independent
  - Implementations on different multi/many-cores
  - Support for hybrid architectures thanks to pattern compositionality
- Rationale
  - Decrease to bare minimum synchronization overhead (speedup)
  - Provide ready-to-use patterns (productivity, time-to-market)

# Two-phase edge preserving parallel de-noising

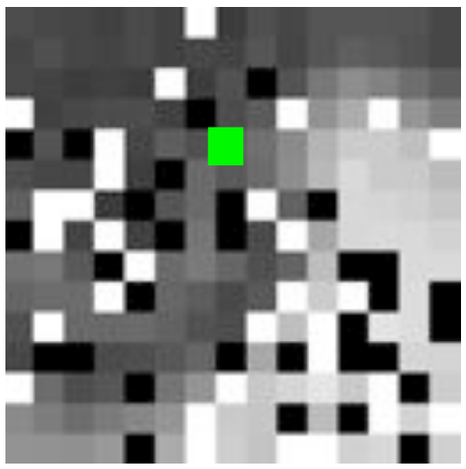
# Denoising explained (video)

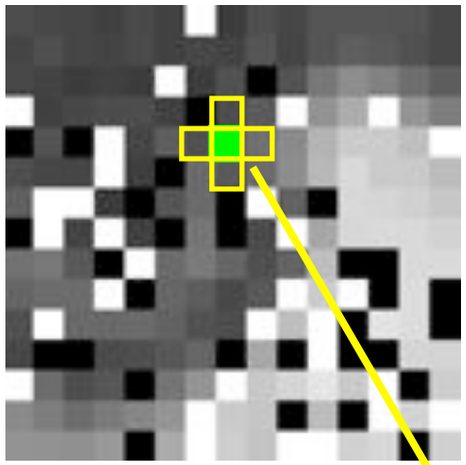
UNIVERSITÀ DEGLI STUDI DI TORINO

**pipeline**

```
map p in pixels                                   detect
  while (winsize<MAX)
    if (homogenous(p,winsize))
      winsize++;
    else if isImpluse(p) return NOISY;
  return NOT_NOISY;
```

```
while !fixpoint                                   denoise
  map u in N (noisy pixels)
    new_u = value_that_minimize F(u);
  reduce(u in N,new_u in NEW_N, diff);
```





## Adaptive median filter

different pixels are independent and can be easily processed in parallel

**pixels are read-only**

## Iterative variational method

answer to the question: which is the greyscale level that better "integrate" in the surrounding (i.e. keeps edges)

**at each iteration an approximation of restored pixels is available**

*In the video case the two stages can be pipelined on*
*m-core+m-core*
*m-core+GPGPU*
*GPGPU+GPGPU*

*and you haven't to decide it at design time o port the code*

$$F_d|_N(u) = \sum_{(i,j)\in N} [|u_{i,j} - d_{i,j}| + \frac{\beta}{2}(S_1 + S_2)]$$

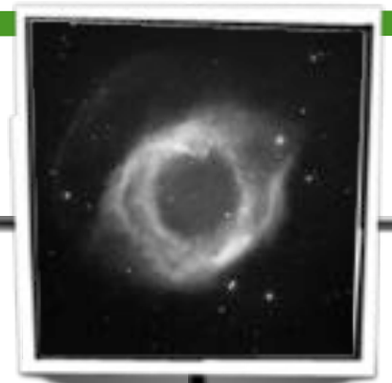
$$\varphi(t) = |t|^\alpha \text{ with } 1 < \alpha \leq 2$$

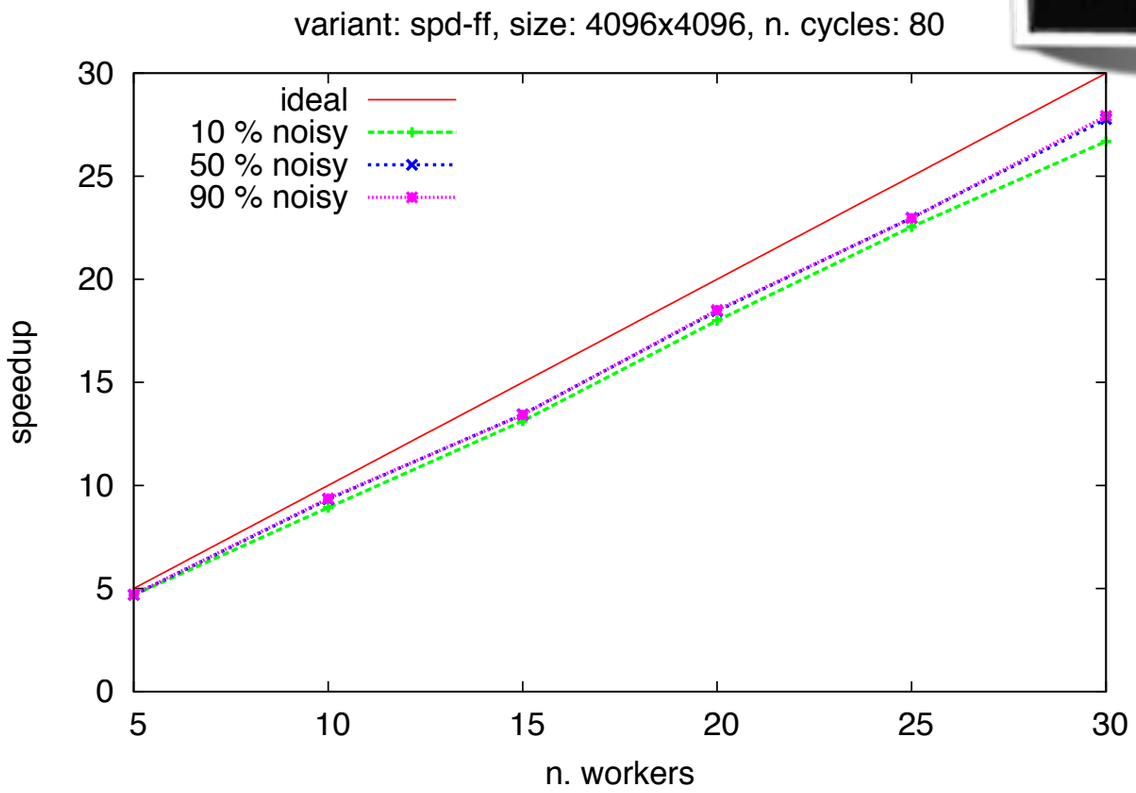
$$S_1 = \sum_{(m,n)\in V_{i,j}\cap N} 2 \cdot \varphi(u_{i,j} - d_{m,n})$$

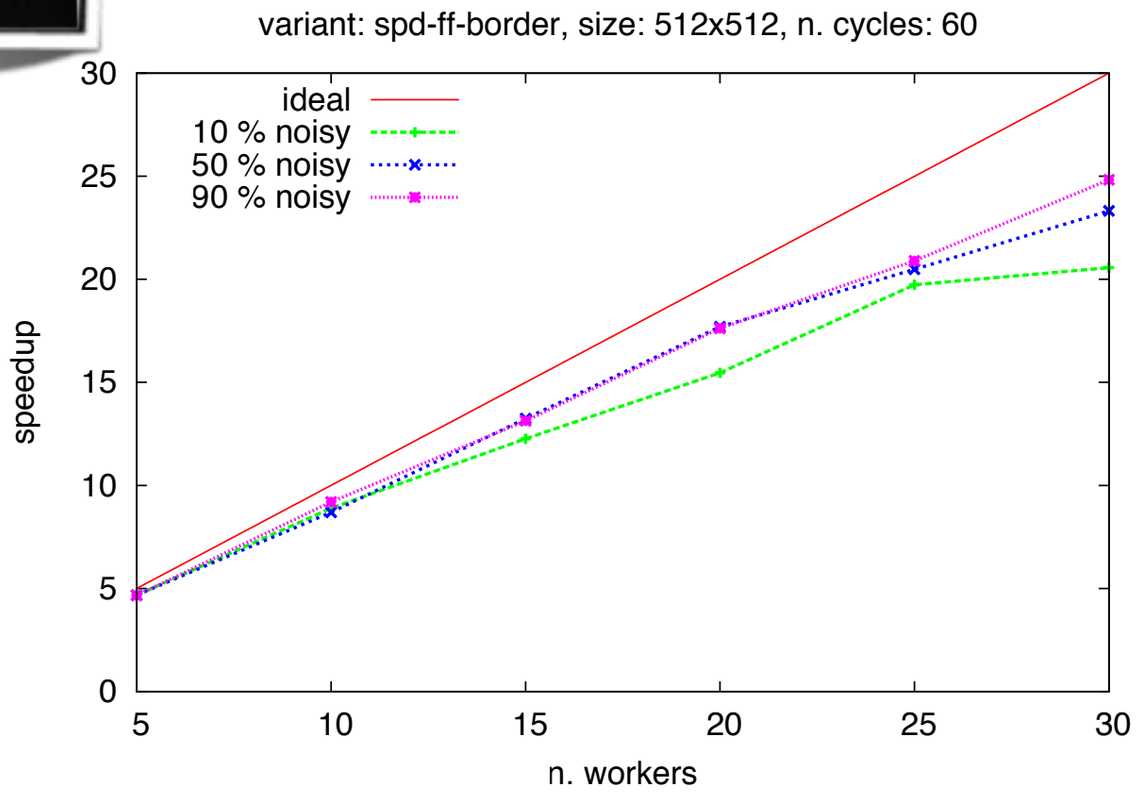
$$S_2 = \sum_{(m,n)\in V_{i,j}\cap N^c} \varphi(u_{i,j} - u_{m,n})$$

# Speedup (multi-core)

UNIVERSITÀ DEGLI STUDI DI TORINO






4 sockets x 8 core x 2 contexts
Xeon E7-4820 @2.0GHz Sandy Bridge
18MB L3 shared cache, 256K L2

# Edge-preserving denoiser: video

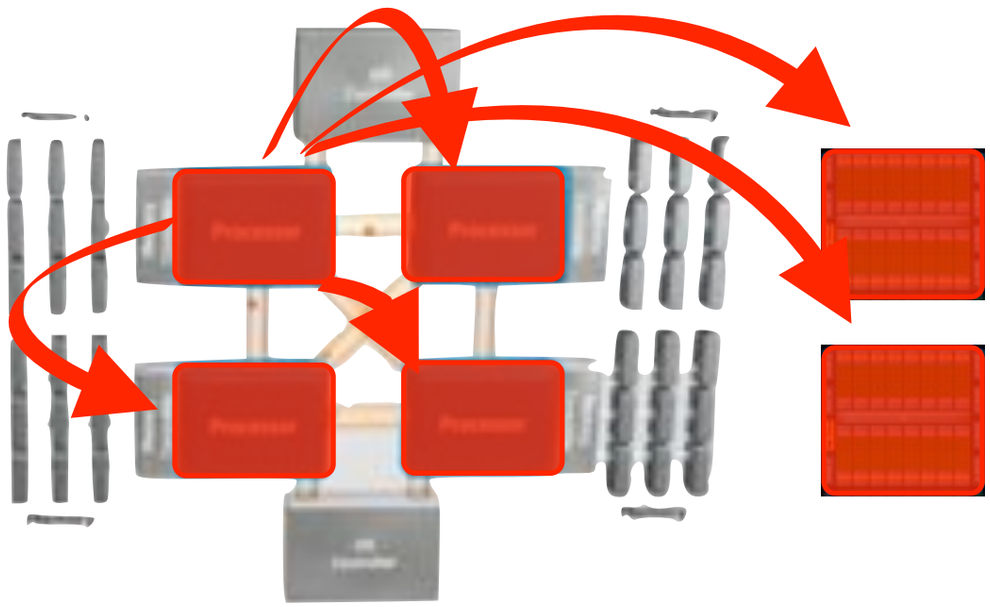
UNIVERSITÀ DEGLI STUDI DI TORINO

```
#include <opencv/highgui.h>
#include <opencv/cv.h>

int main(int argc, char *argv[]) {
  CvCapture *capture;
  IplImage * frame,clean_frame;
  char key;
  vector<noisy_t> noisy;
  cvNamedWindow("Video", CV_WINDOW_AUTOSIZE);
  capture = cvCreateCameraCapture(CV_CAP_ANY);
  //capture = cvCreateFileCapture("/path/to/your/video/test.avi");
  while(true) {
    frame = cvQueryFrame(capture);          // get a frame from device
    noisy = myDetect(frame);                // detect noisy pixels
    clean_frame = myDenoise(frame,noisy); // denoise the frame
    cvShowImage( "Video", clean_frame);   // show the denoised frame
    key = cvWaitKey(100);
  }
  cvReleaseCapture(&capture);
  cvDestroyWindow("Video");
}
```

# Offloading on soft (i.e. not used cores) and HW accelerators

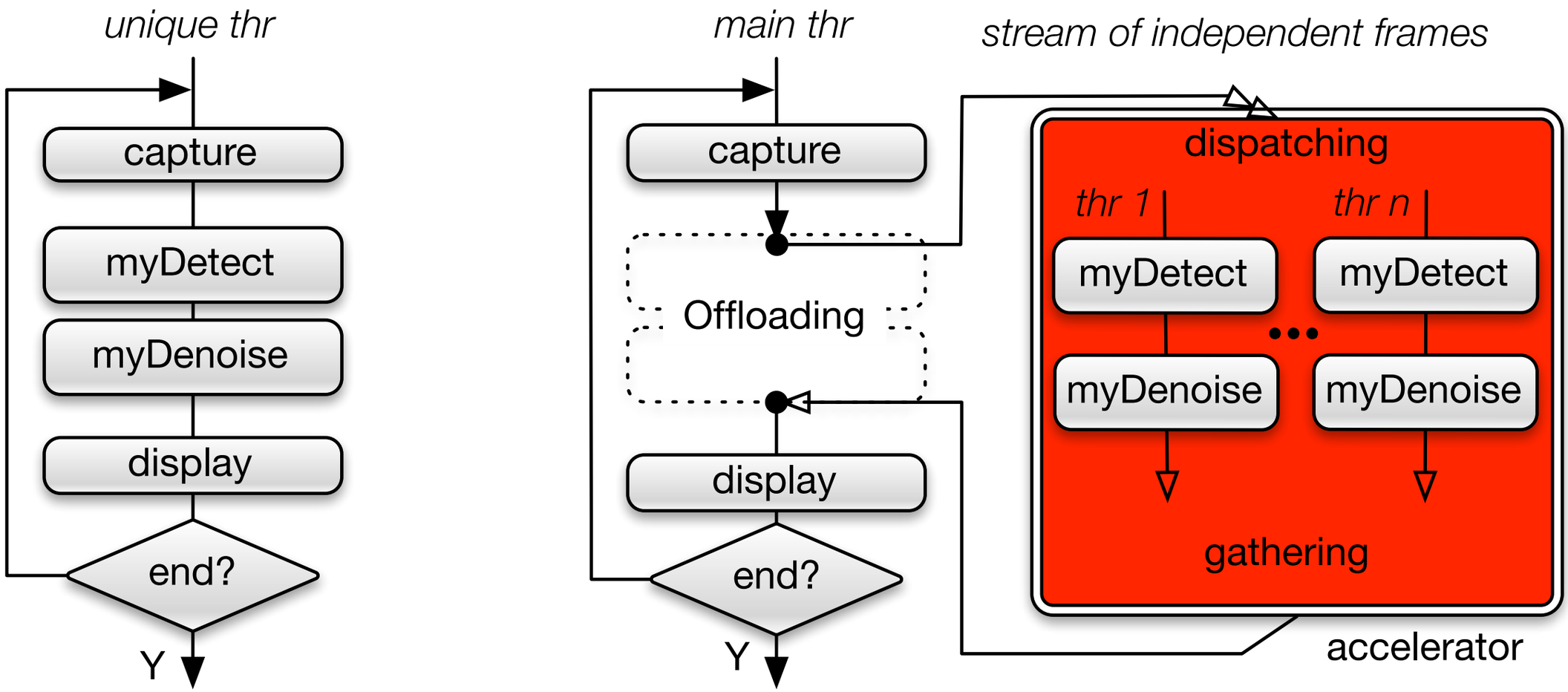
UNIVERSITÀ DEGLI STUDI DI TORINO

- offloading
  - onto other cores and accelerators

# Single design - many implementations

UNIVERSITÀ DEGLI STUDI DI TORINO



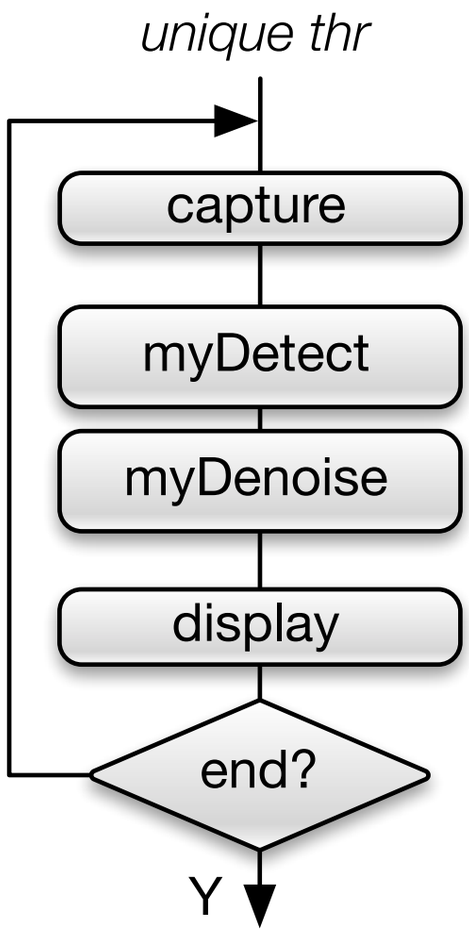

**Sequential**

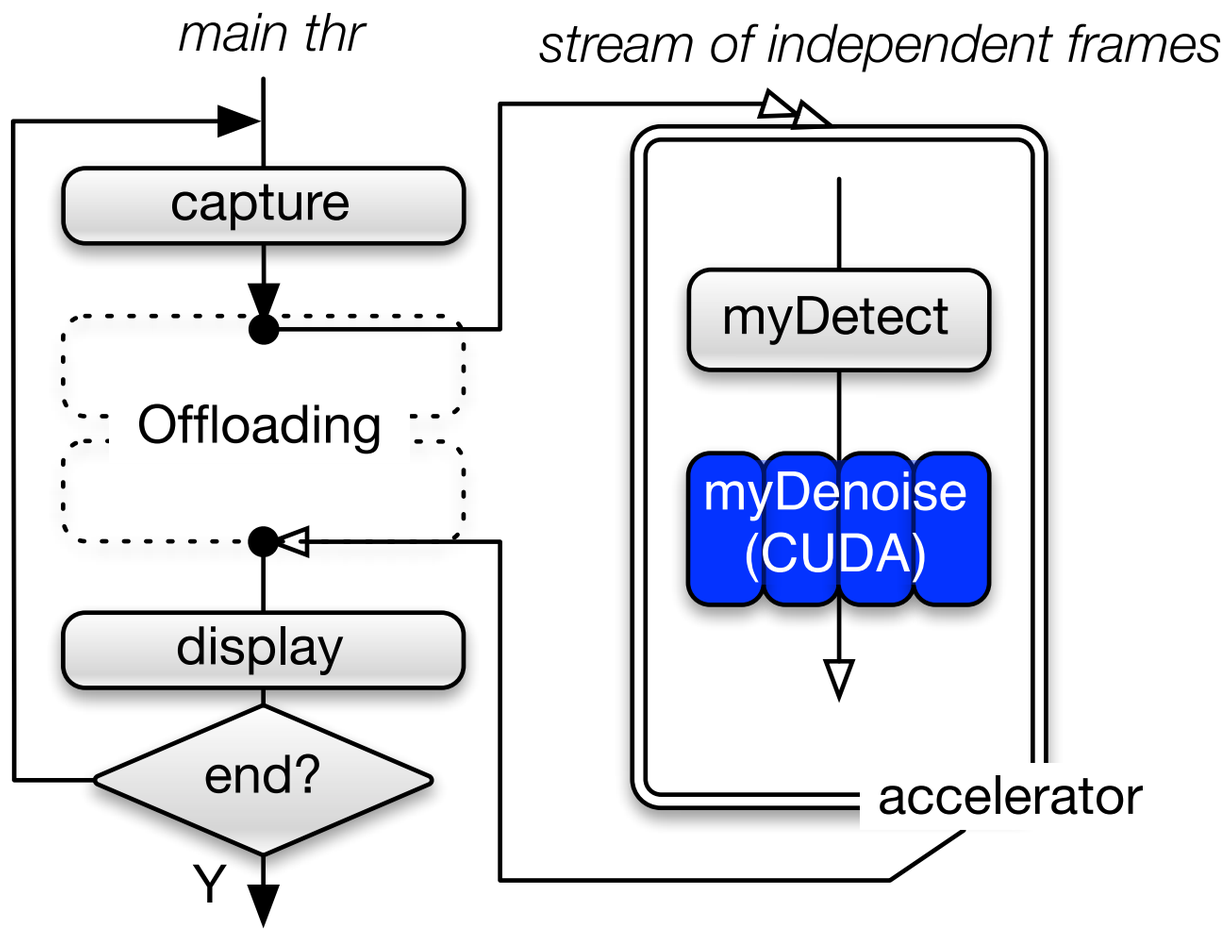
**Parallel - Farm(Pipeline(myDetect,myDenoise))**

Changing the structure does not require re-writing business code (gray)

# Single design - many implementations

UNIVERSITÀ DEGLI STUDI DI TORINO



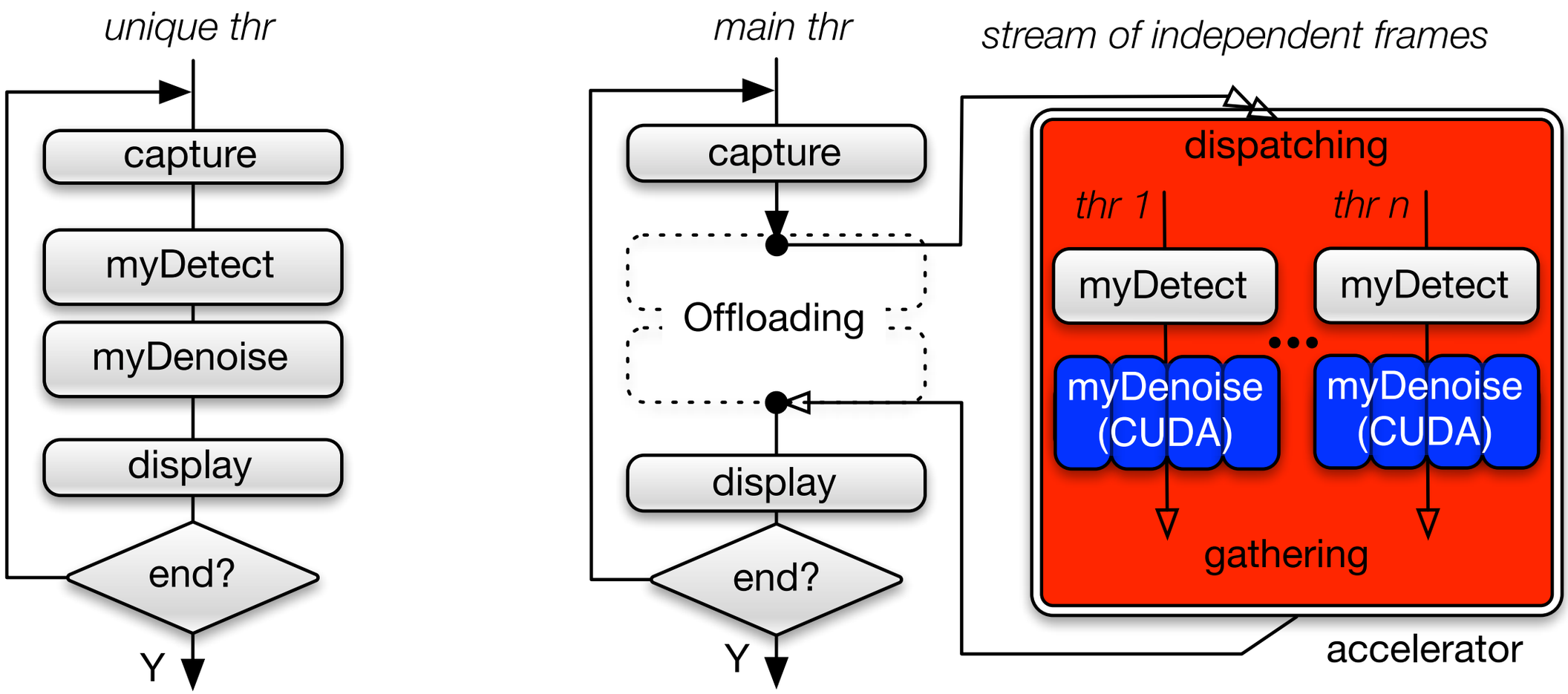

**Sequential**

**Parallel - Pipeline(myDetect,Map(myDenoise))**

Changing the structure does not require re-writing business code (gray)

# Single design - many implementations

UNIVERSITÀ DEGLI STUDI DI TORINO




**Sequential**

**Parallel - Farm(Pipeline(myDetect, Map(myDenoise)))**

Changing the structure does not require re-writing business code (gray)

# FF multi-core vs FF hybrid

UNIVERSITÀ DEGLI STUDI DI TORINO

| noise | FF 32 cores Intel 4x8x2 2GHz | FF 8 cores (detect) + Tesla C2050 (denoise) | Seq Intel 4x8x2 2GHz | |
|---|---|---|---|---|
| Lena 512x512 | | | | |
| 10 | 1.8 s | 1.9 s | 32 s | |
| 50 | 6.5 s | 2.3 s | 162 s | |
| 90 | 10.9 s | 2.8 s | 290 s | |
| Space 4096x4096 | | | | |
| 10 | 78 s | 12 s | 2093 s | 174x |
| 50 | 373 s | 46 s | 10400 s | 226x |
| 90 | 665 s | 77 s | 18571 s | 271x |

# Demo

# Conclusions

UNIVERSITÀ DEGLI STUDI
DI TORINO

- Two-phase denoising
  - Variational methods can be made fast and efficient
    - Works up to 95% or noise, comparable to jpeg on 50% of noise
  - CPU/GPGPUs/Hybrid
  - Edge-preserving restoration works also for other kinds of noise
- GPGPUs
  - Needs high-level, CUDA/OpenCL too close to the metal
  - Well integrate with functional style and higher order patterns
- Image analysis is just an example of usage of FastFlow
  - classification, mining, compressing, string-alignment, network inspection (also used in nTop), and many more... See http://di.unito.it/fastflow
  - MonteCarlo simulations
    - precessed CMS@CERN channel H→ZZ→4l Higg's boson for July 2012 claim

UNIVERSITÀ DEGLI STUDI
DI TORINO

# Thank you

# Nondeterministic variants & Convergence speed

UNIVERSITÀ DEGLI STUDI DI TORINO

- **flat** (used in CUDA version)
  - use block halo, block size = 1
  - **do independent**
  - deterministic - **slow convergence**
  - easy CPU and GPU - can be linearized
- border
  - use block halo
  - **do independent** on tiles, **do across** within tiles
  - nondeterministic - **fast convergence**
  - easy on CPU and GPU - cannot be linearized
- **std** (used in multicore version)
  - don't use a block halo
  - **do independent** on tiles, **do across** within tiles
  - nondeterministic - **fast convergence**
  - easy CPU and GPU - cannot be linearized
- cluster
  - **do independent** on tiles, **do across** within tiles
  - deterministic - **fast convergence**
  - easy on CPU, difficult on GPU - can be linearized