GPU TECHNOLOGY CONFERENCE
MARCH 24-27, 2014 | SAN JOSE, CA

The Brightest People
The Best Ideas
The Biggest Opportunities

FastFlow: Combining Pattern-Level Abstraction and Efficiency in GPGPUs

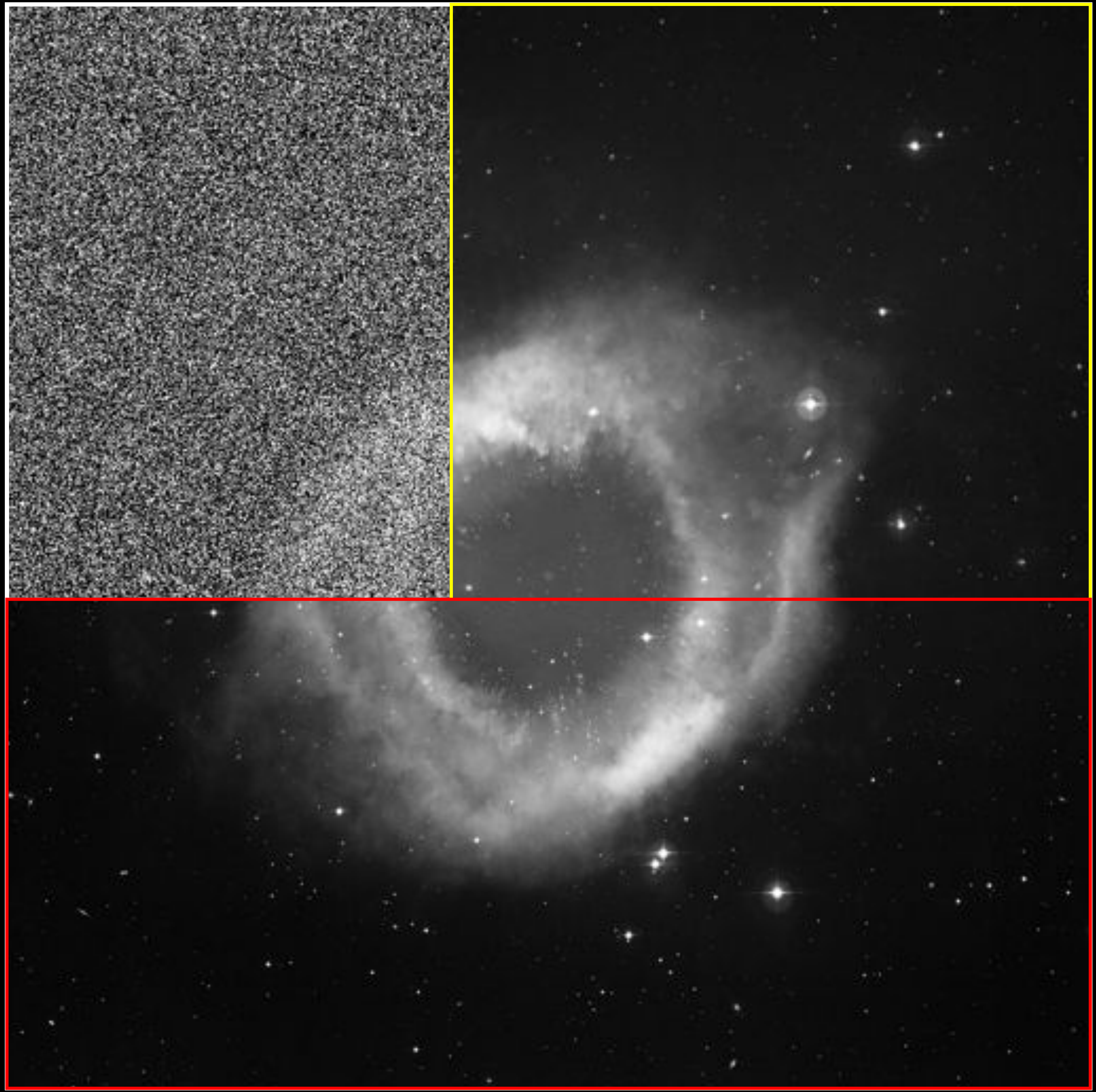
Marco Aldinucci, Computer Science Department, University of Turin, Italy
PI of the CUDA research center at University of Turin, Italy

M. Torquati (University of Pisa, Italy), M. Drocco, G. Peretti Pezzi (University of Turin, Italy), C. Spampinato (University of Catania, Italy)

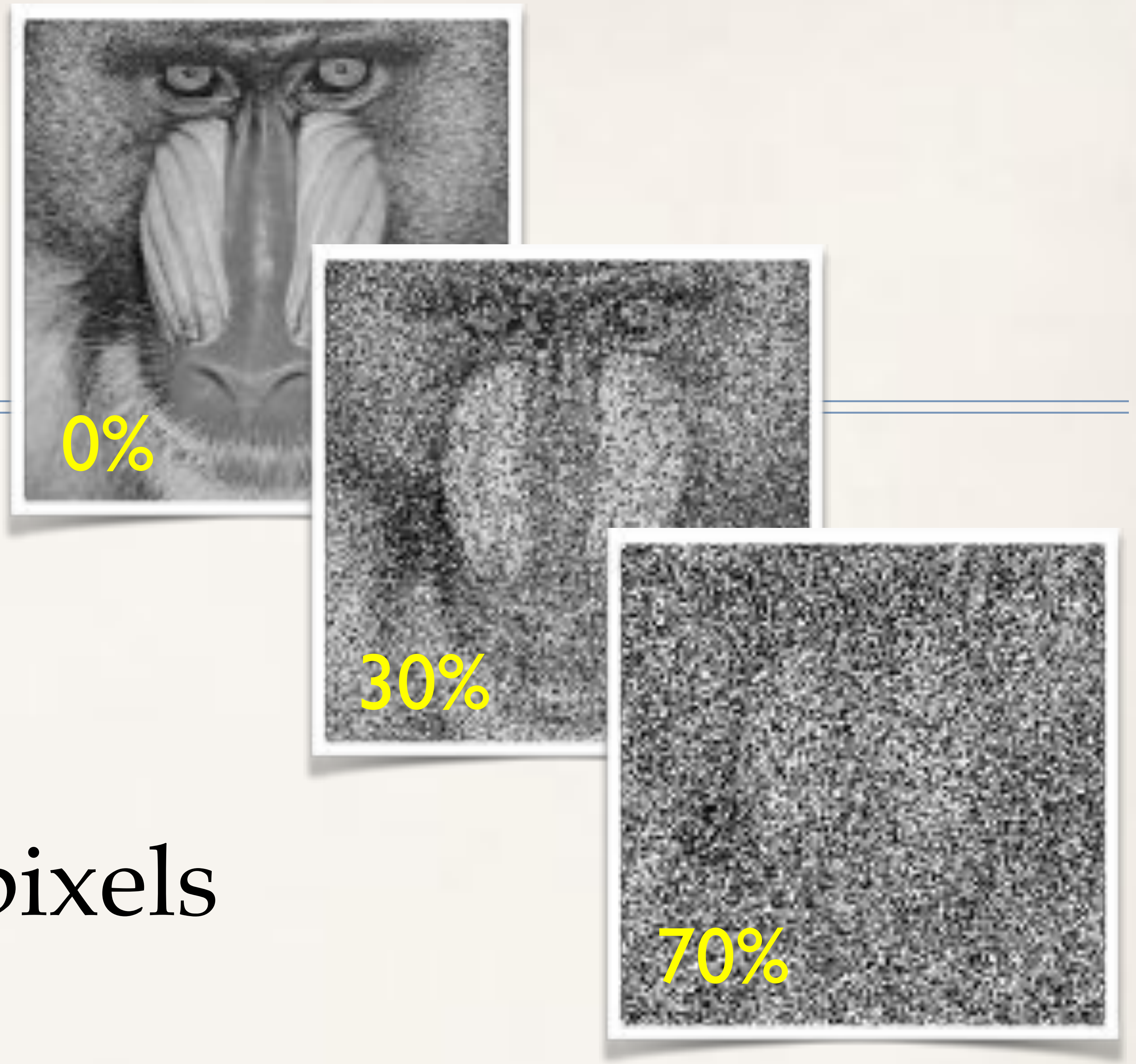
Date: 25 March 2014, San Jose, CA, USA - Presentation S4585

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
noise 70%
Restored
Original

Salt & Pepper noise



- Electronic and signal noise
- Uniform distribution of "saturated" white/black pixels
  - 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
  - not only outliers are filtered (image results smoothed)

Salt & Pepper noise

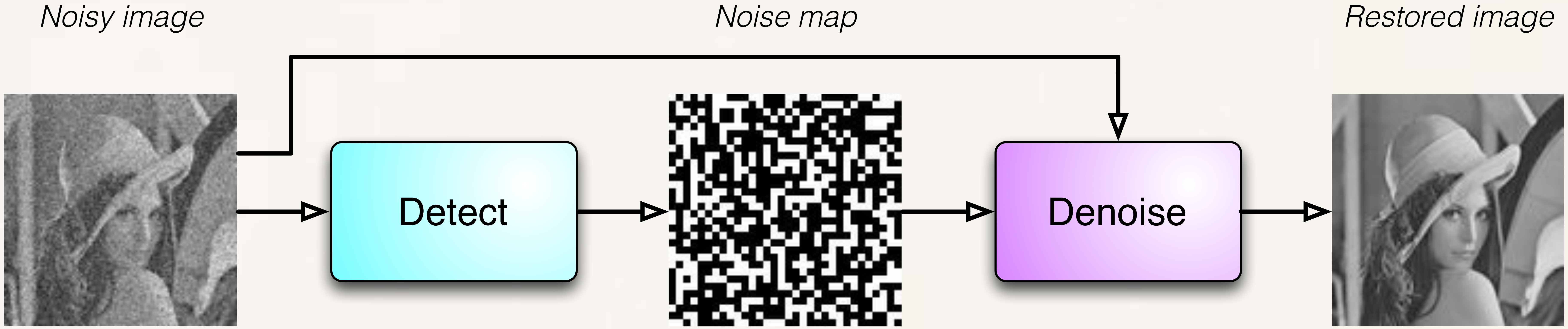
- Electronic and signal noise
- Uniform distribution of “saturated” white/black pixels
  - Measured as percentage of affected vs overall pixels
- Typically restored using statistic filters: e.g. median, median-adaptive
- Not satisfactory for high levels of noise
  - not only outliers are filtered (image results smoothed)

Gaussian noise

Original
Var 10
Var 30
Var 50

- Poor illumination, temperature, circuit noise
- White additive noise in the frequency domain
  - Measured with mean and variance of the Gaussian distribution
  - Affect all pixels, with an additive “white” value distributed as a Gaussian
- Typically restored using statistic filters: e.g. median, Guassian smoothing
- More difficult to manage: restored image results smoothed

Two-stage restoring

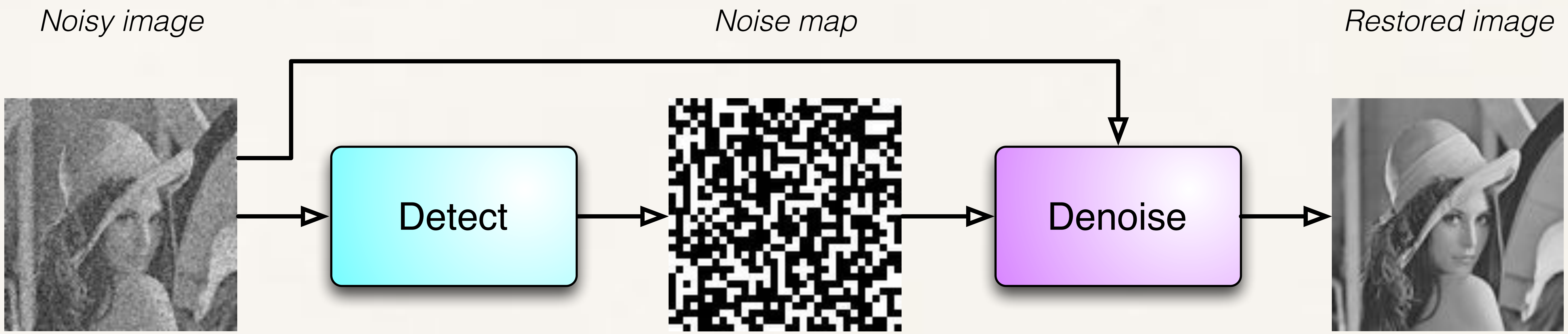


- progressive-switching/adaptive median
- neural/bayesian networks, fuzzy, …

- variational
- statistic

- Decouple detection decoupled from restoration
  - Pixels considered not outliers are not altered by restoration
  - False positives impair restoration quality

Two-stage restoring



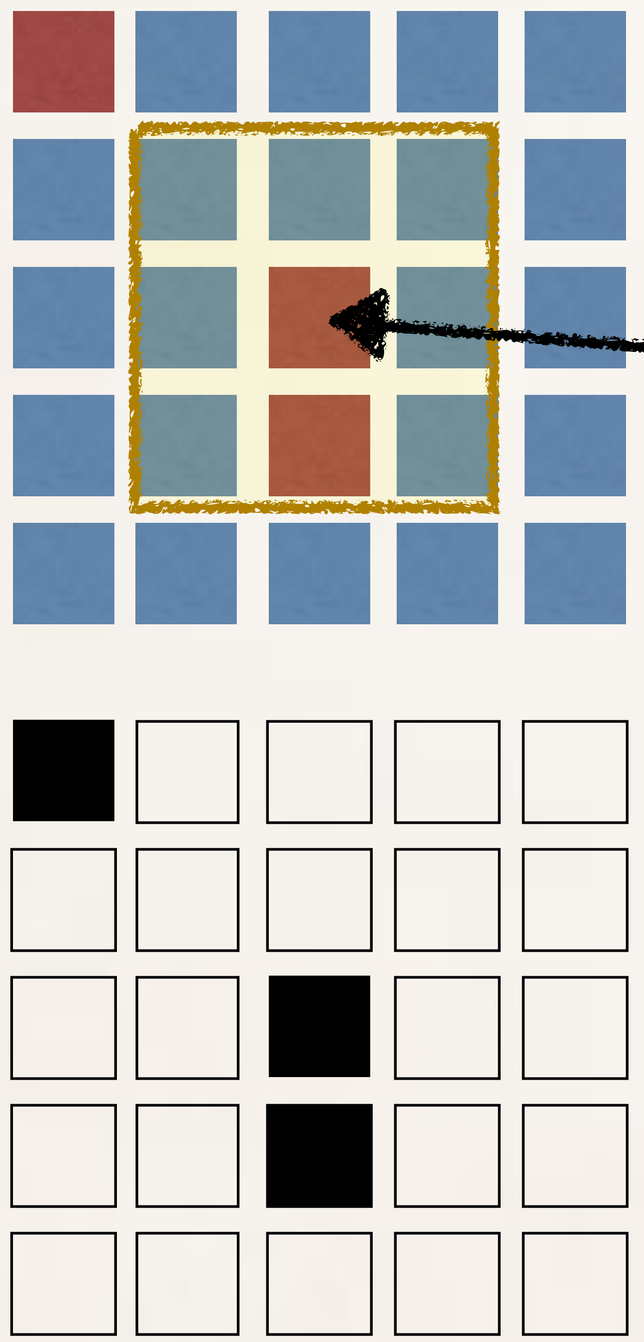
- **progressive-switching/adaptive median**
- neural/bayesian networks, fuzzy, …

- **variational**
- statistic

- Statistic detection + variational restoration
  - High quality, edge-preserving filtering
  - Much more computational demanding, not really viable without parallelism
    - Matlab on a single 256x256 image with 50% of noise requires dozen of minutes
  - Stages can be pipelined

Variational De-noising: an iterative optimisation problem

img



noiseMap

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

F(…) weight differently noisy and not noisy pixels

```
do
  foreach i,j
    if (noisyMap[i,j])
      let N = neighb8(img,i,j)
      let k in 0..255
        u=argmin(F(k,N,noiseMap))
      img[i,j]=u
while (process not converge)
```

You can write it directly with C++ and CUDA but what happens splitting the work onto 2 GPGPUs?

Variational Denoise: F(…) details (almost universal for different noise types)

regularization term data fidelity term

$$\operatorname*{argmin}_{u \in N} F(u) = \alpha \int R(u) + \beta \int D(u,d)$$

In the spatial domain

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

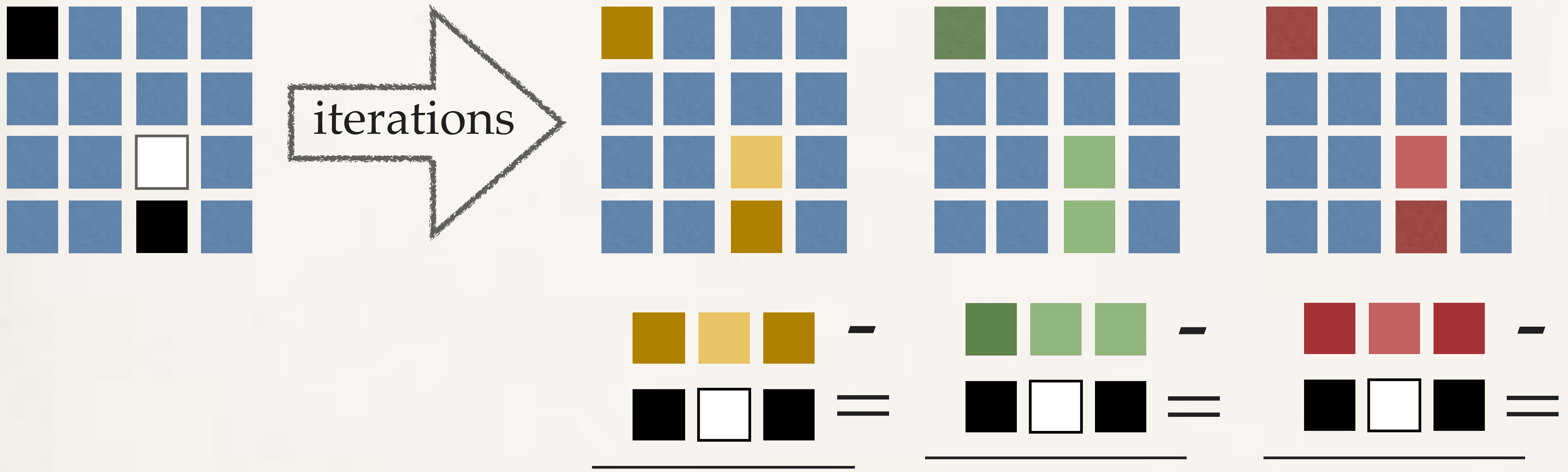
$$S_1 = \sum_{(m,n)\in V_{i,j}\cap N} 2 \cdot \varphi(u_{i,j} - d_{m,n}) \qquad S_2 = \sum_{(m,n)\in V_{i,j}\cap N^c} \varphi(u_{i,j} - u_{m,n})$$

$\varphi(t) = |t|^\alpha$ with $1 < \alpha \leq 2$ for Salt&Pepper

- R. Chan, C. Ho, and M. Nikolova, Salt-and-pepper noise removal by median-type noise detectors and detail-preserving regularization. IEEE Trans. on Image Processing, vol. 14, 2005.
- M. Aldinucci et al. A parallel edge preserving algorithm for salt and pepper image denoising. In Intl. Conference on Image Processing Theory Tools and Applications (IPTA), 2012. IEEE.

Convergence can't be evaluated with a reduce (involves three iterations, i.e. memory)

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

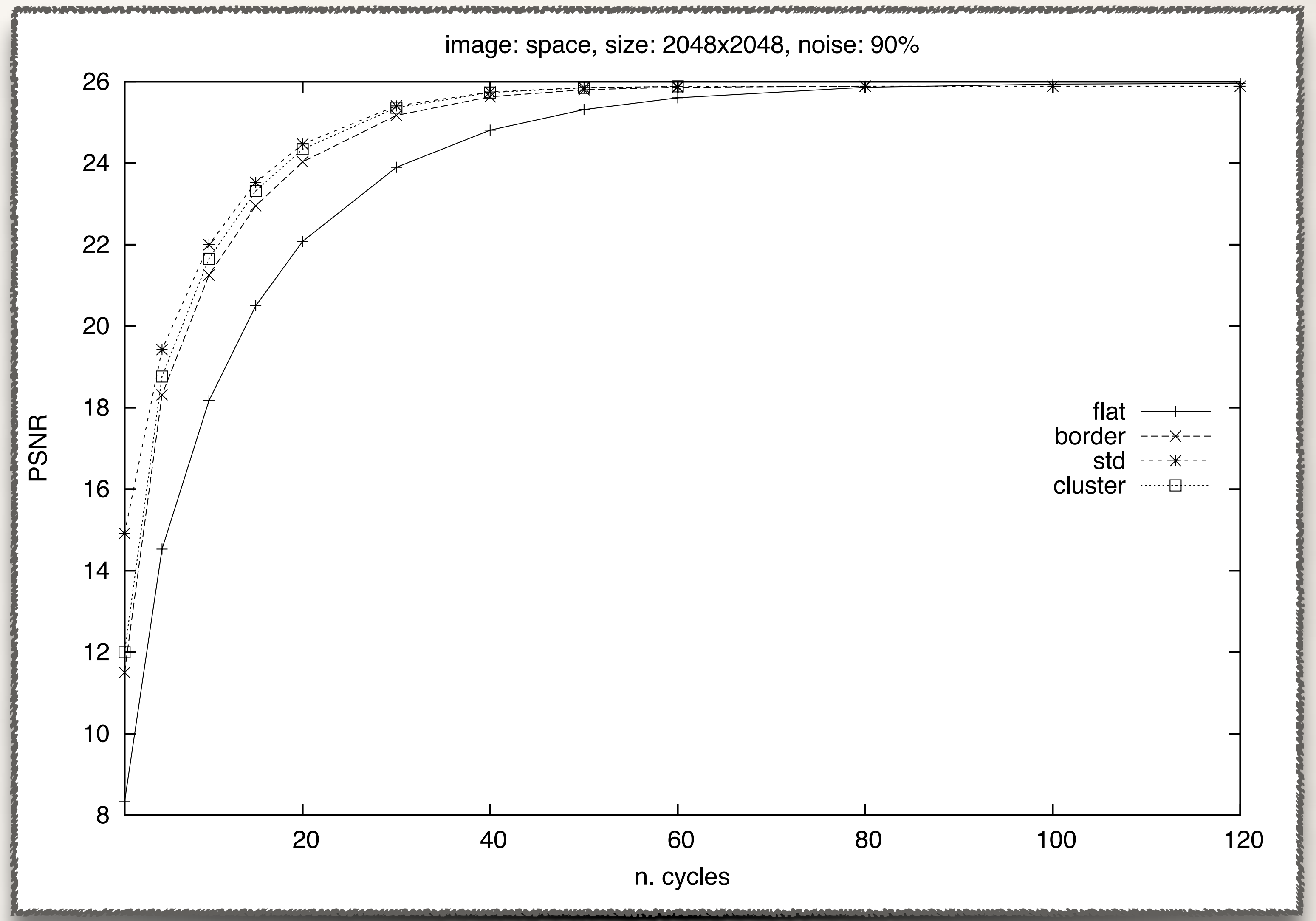


Residuals: 2 4 2 · 5 8 5 · 6 7 6

2 4 2 · 5 8 5

Diff of residuals: 3 4 3 · 1 1 1

Reduce of diffs: $\sum \Delta^{(k)} = 10$ · $\sum \Delta^{(k+1)} = 3$



quality →

time →

terminate if

$$\frac{\left|\left|\sum \Delta^{(k)}\right|\right| - \left|\sum \Delta^{(k+1)}\right|}{\left|\sum \Delta^{(k)}\right|} < \epsilon \in \mathcal{R}$$

M. Aldinucci, M. Drocco et al.
IJHPCA submitted

Quality results




Original
Baboon standard
test image
1024x1024

10% impulsive noise

50% impulsive noise

90% impulsive noise

Restored



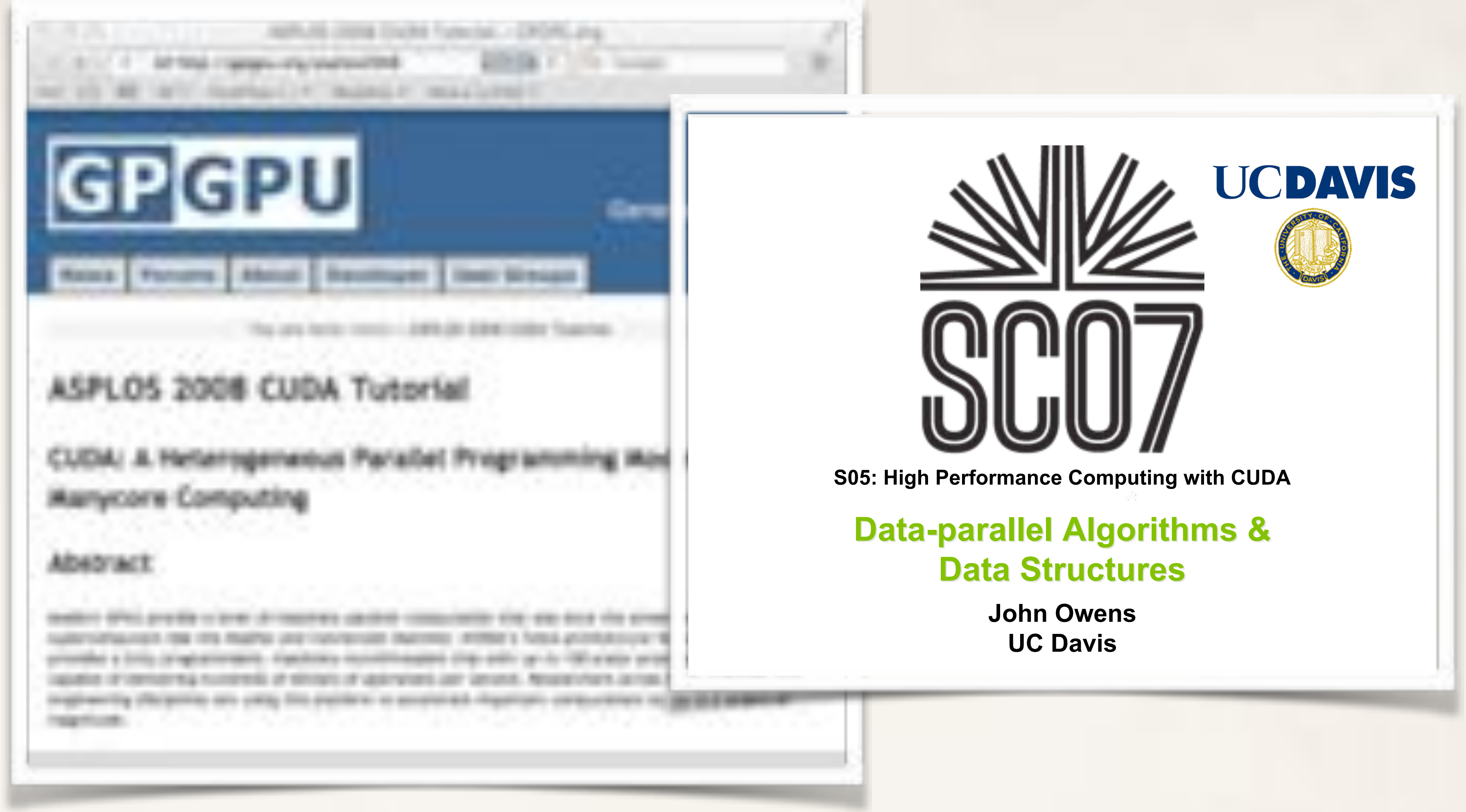
PNSR 43.29dB MAE 0.35

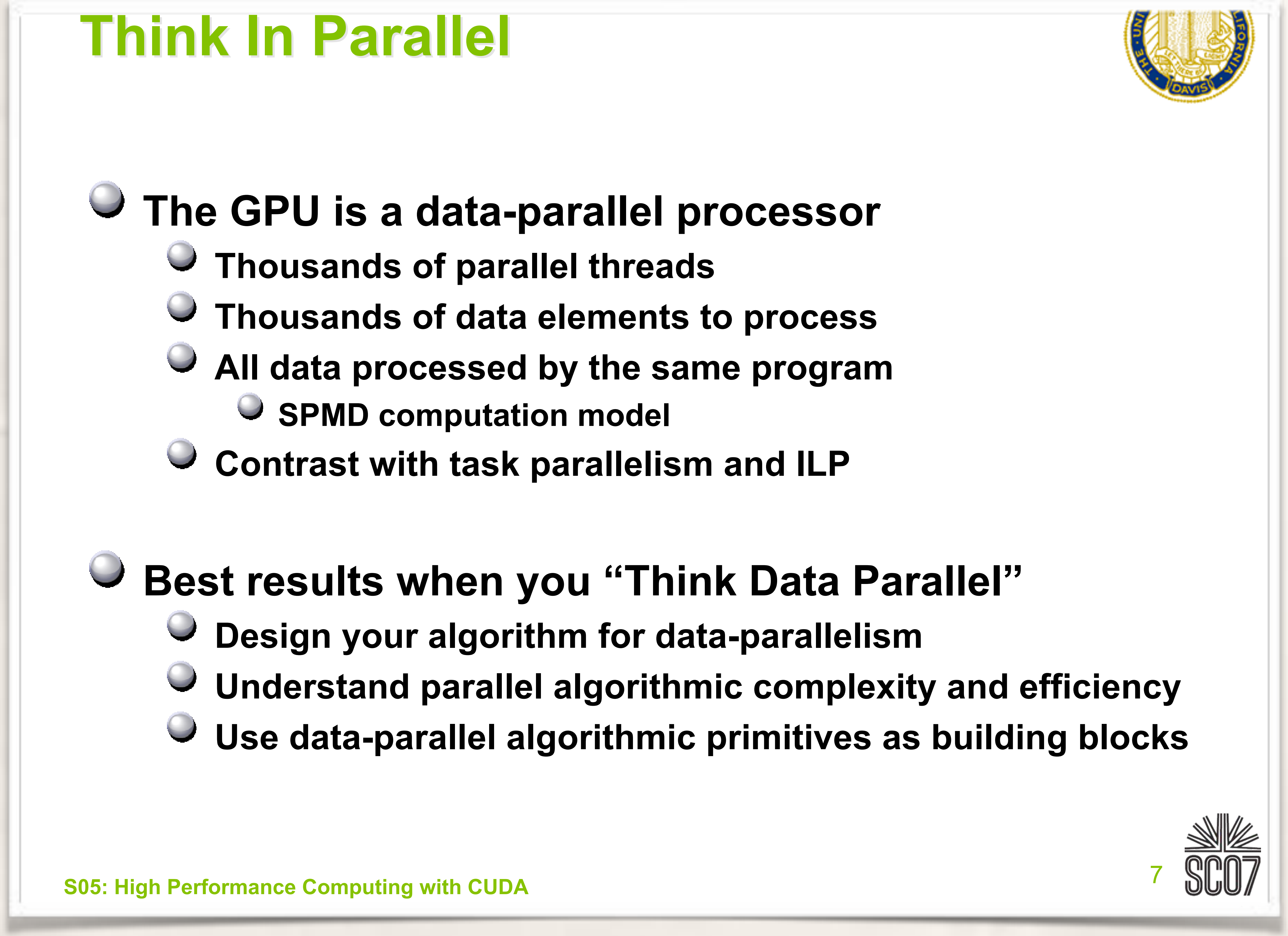
PNSR 32.75dB MAE 2.67

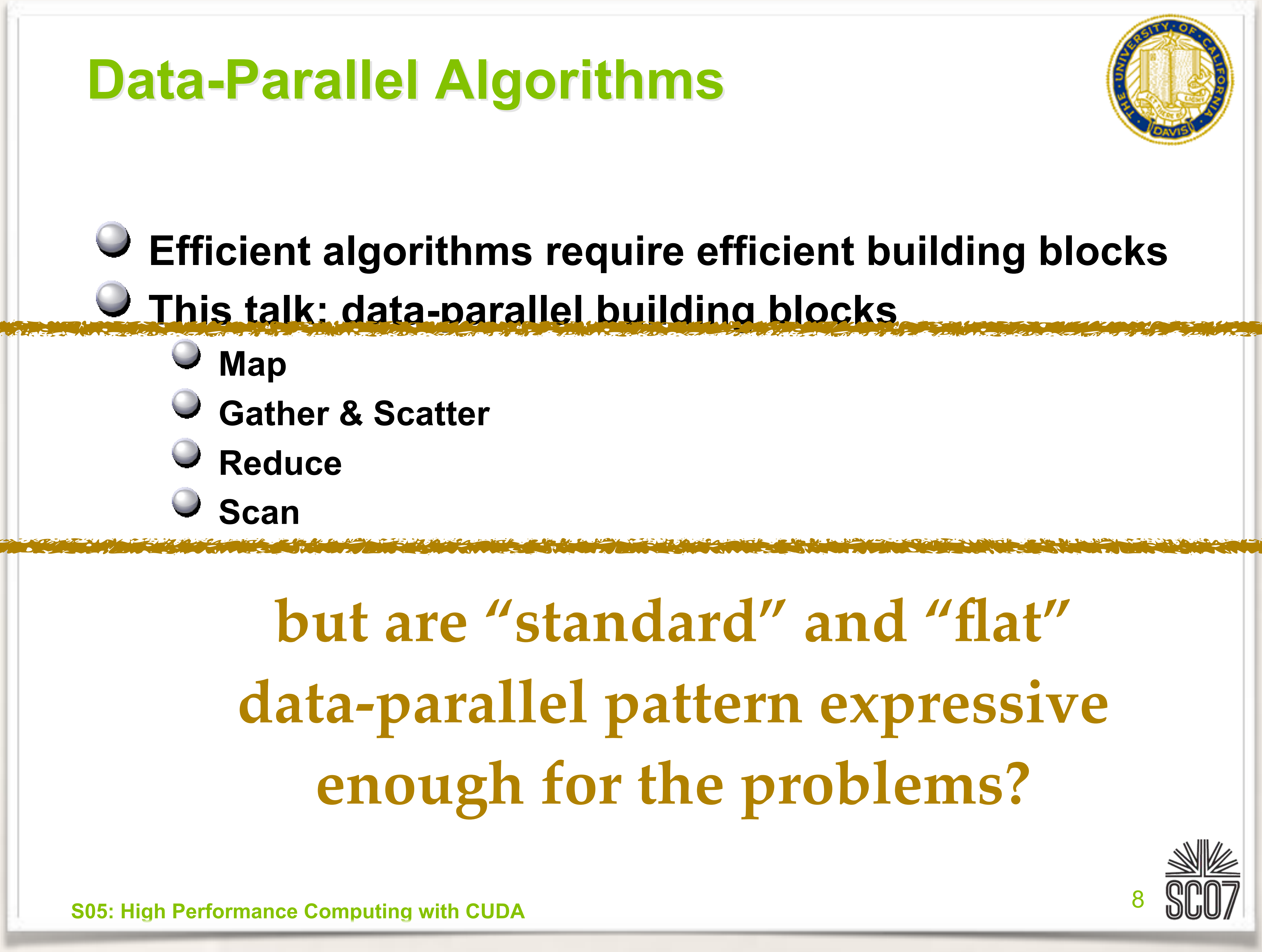
PNSR 23.4 MAE 11.21

Patterns are a natural approach in GPGPUs

… and this well-known from long time







but are "standard" and "flat" data-parallel pattern expressive enough for the problems?

Rationale: patterns are there but are not simply map or reduce

- Detect-Denoise can be naturally pipelined
- Denoise is a (sort of) map with a stencil
  - Where, x = < x1, x2, …, xn >, map f x = < f(x1), f(x2), …, f(xn) >
  - Can be written as a map, but is neither natural nor easy
- Convergence evaluation is map across three iterations and reduce
  - Even more complex to write it as a MapReduce (if not impossible)
- Cholesky LU or C4.5 tree pruning with map, reduce or MapReduce?

stencilReduce

- a (low-level) powerful pattern
- it capture most of the interesting data parallel GPGPUs computations
- 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, …

```
loop
   before (…)
   stencil<stencilK,reduceK> (data[i], env)
   reduce op data
   after (…)
```

stencilReduce

- a (low-level) powerful pattern
- it capture most of the interesting data parallel GPGPUs computations
- 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, …

Compute on host
possibly in parallel on CPU cores

Unified Memory
greatly simplify this part

```
loop
    before (…)
    stencil<stencilK,reduceK> (data[i], env)
    reduce op data
    after (…)
```

CUDA code

Compute on host
possibly in parallel on CPU cores

Low-level approaches = lot of freedom (threads, CUDA, OpenCL, MPI, …)

you can design your algorithms as you want

… as a car, you can drive it where you want

car

Val D'Orcia, Tuscany, Italy

Low-level approaches = lot of freedom (threads, CUDA, OpenCL, MPI, …)

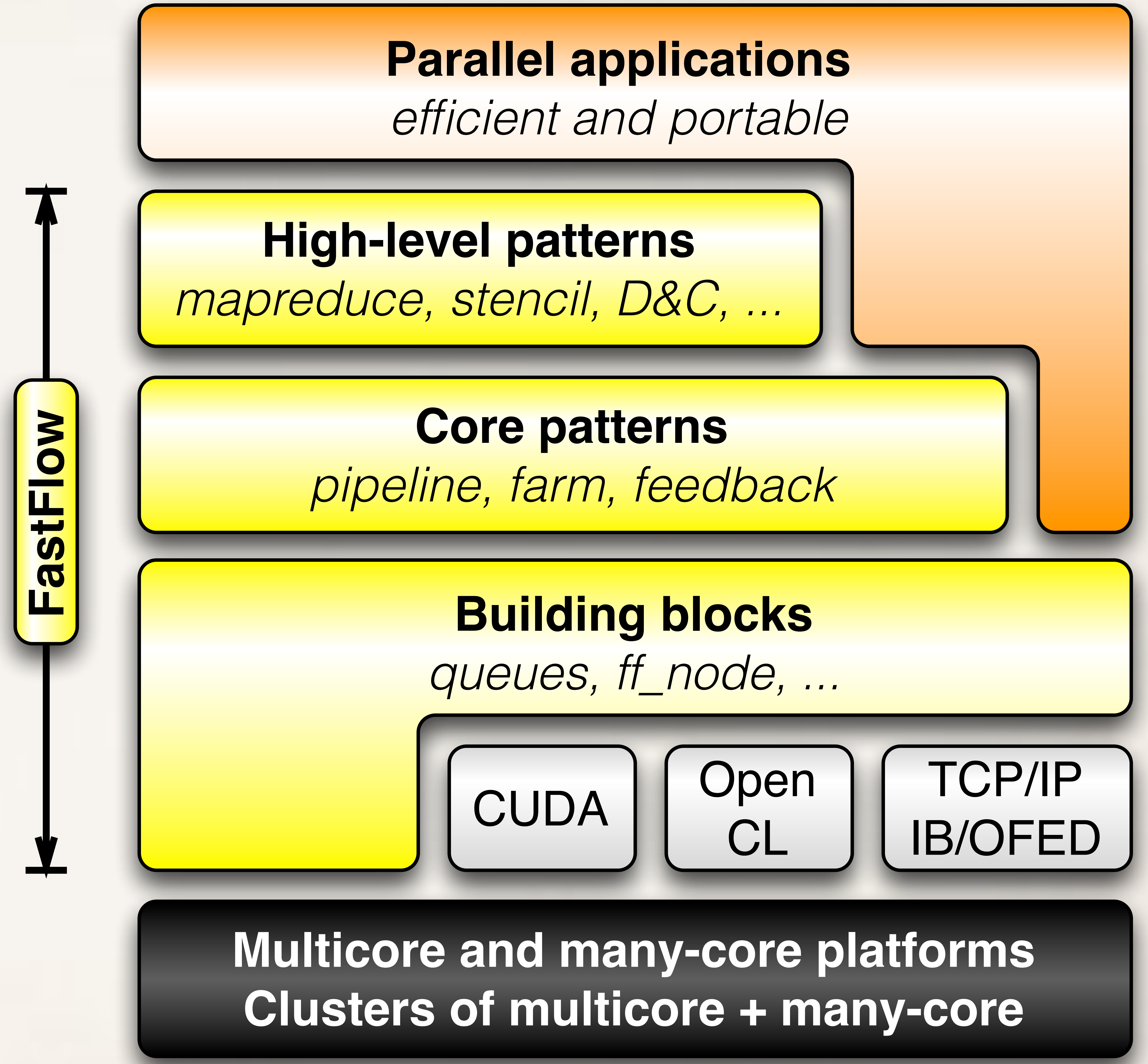
- Think in parallel & high-level
  - Efficiency, portability, time-to-market
- High-level parallel patterns
  - Describing collective behavior
  - Can be: expressive, efficient, compositional
  - Targeting multicore, GPGPUs, distributed with an unifying vision
  - On various CPUs/GPGPUs and OSes

you can design your algorithms as you want
… as a car, you can drive it where you want

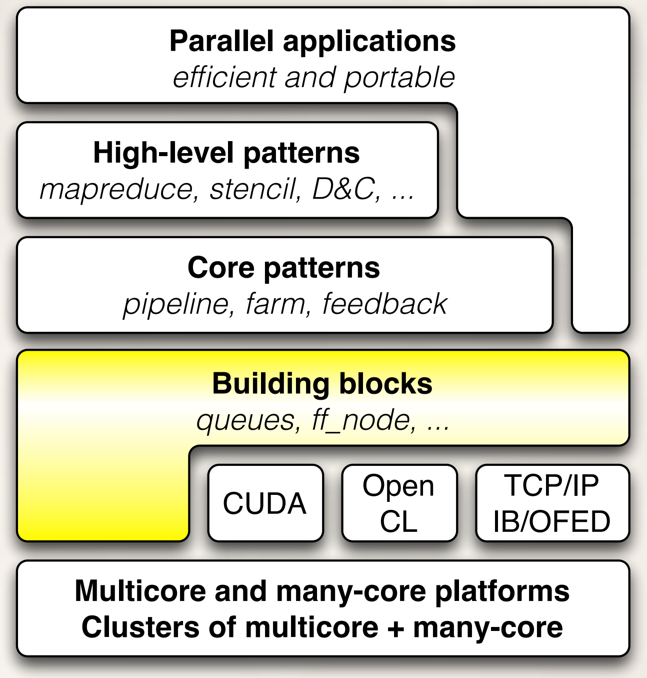
Sao Paulo, Brasil

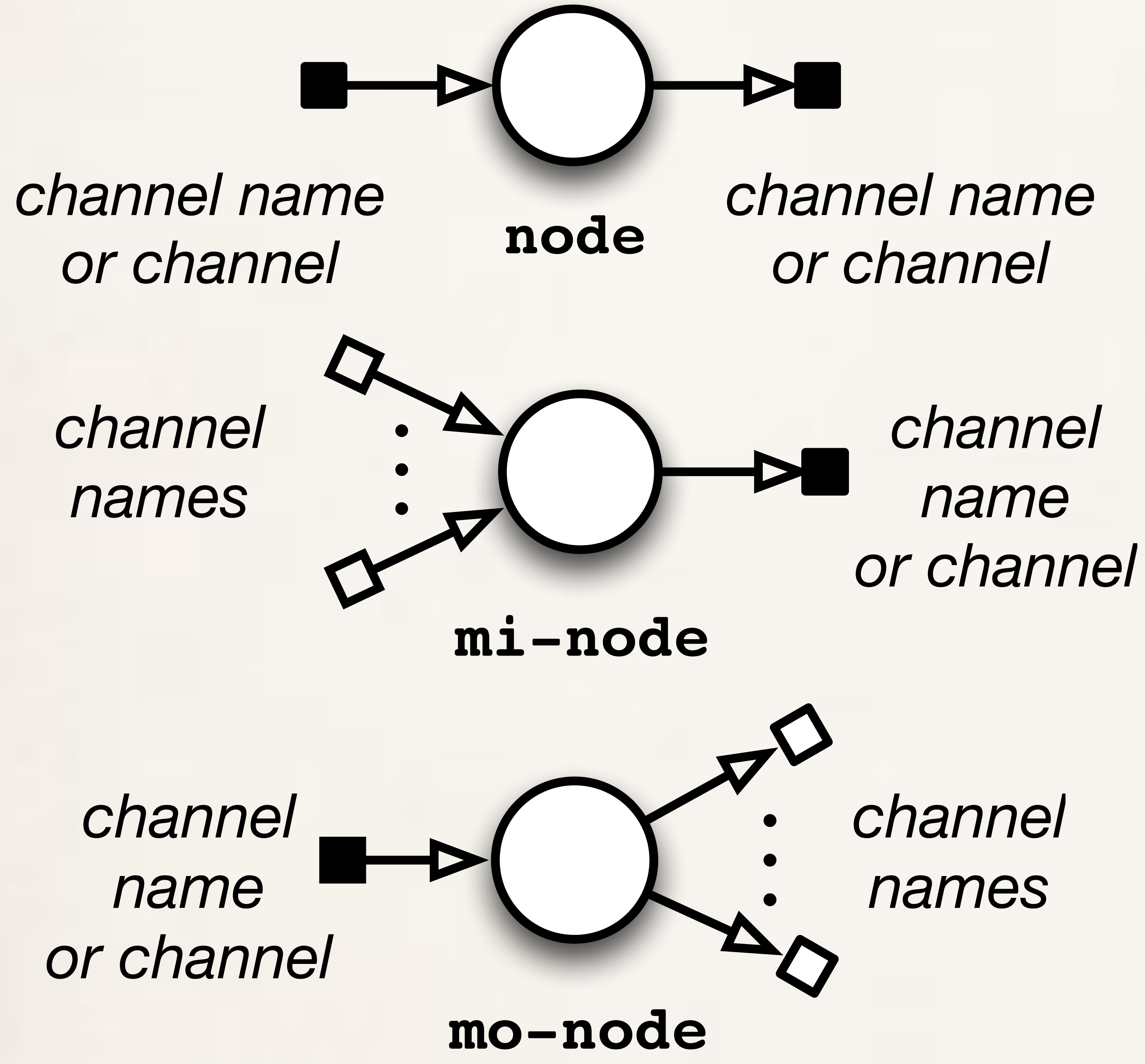
FastFlow (FF)

- C++ header-only library
  - Portable everywhere exists a C++ compiler
  - Originally designed for high-frequency streaming
- Provides stream-oriented and data-parallel patterns
  - compositional, efficient
- Accommodate diversity
  - if you need a different pattern, do it extending a C++ class
- Multi-core, GPGPUs, distributed
- https://sourceforge.net/projects/mc-fastflow



FF building blocks: nodes and channels





threads or processes
threads are non-blocking
(can be suspended using
a native protocol)

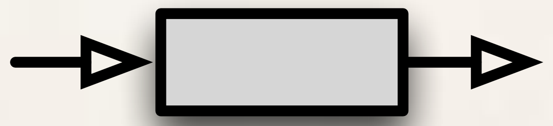
FF bound shmem FIFO channel

Single-Producer-Single-Consumer lock-free fence-free queue

FF unbound shmem FIFO channel

Single-Producer-Single-Consumer lock-free fence-free queue

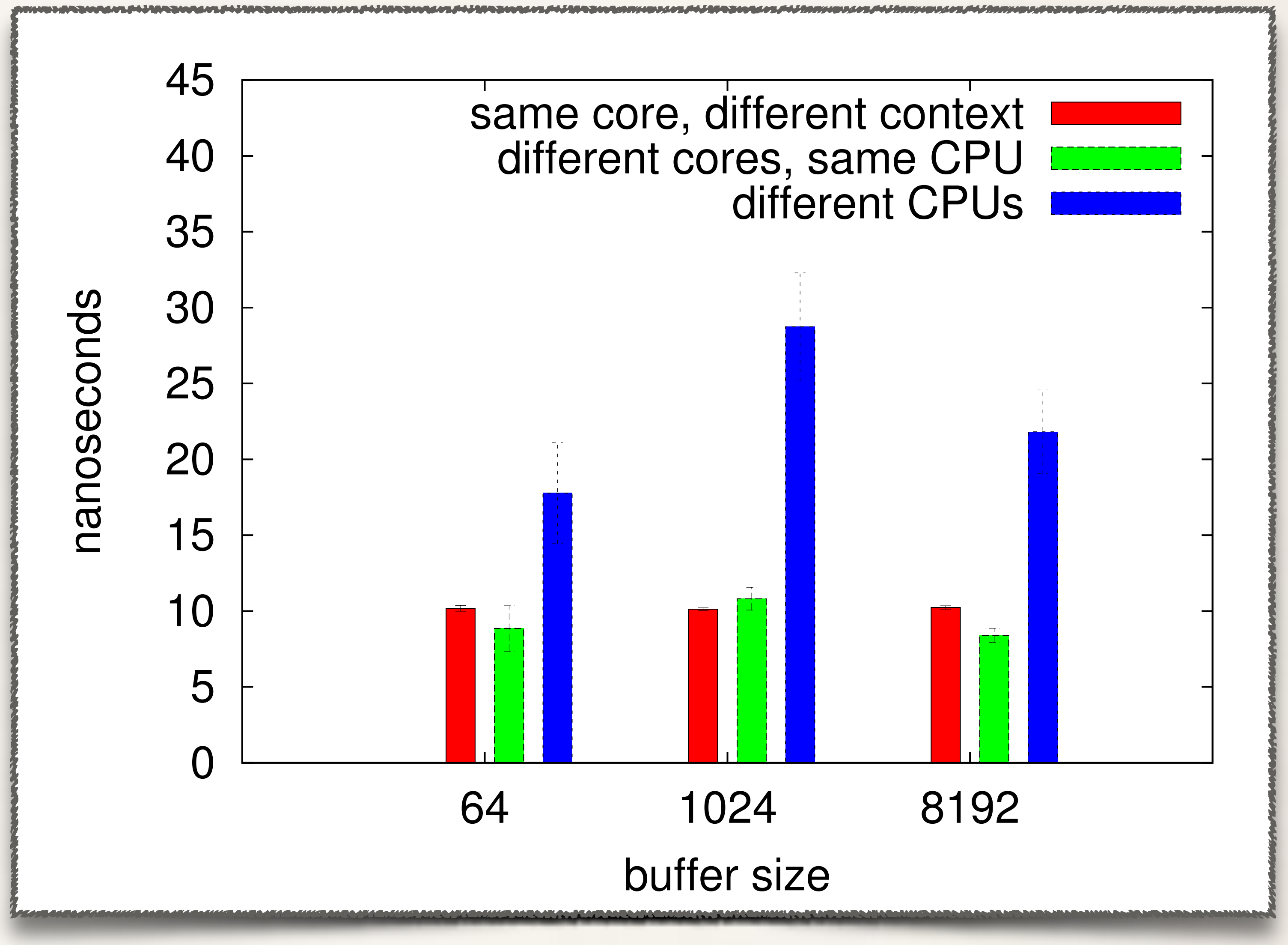
shmem channels communicate pointers in a message passing style



Distributed zero-copy channel

0MQ/TCP or native IB/OFED

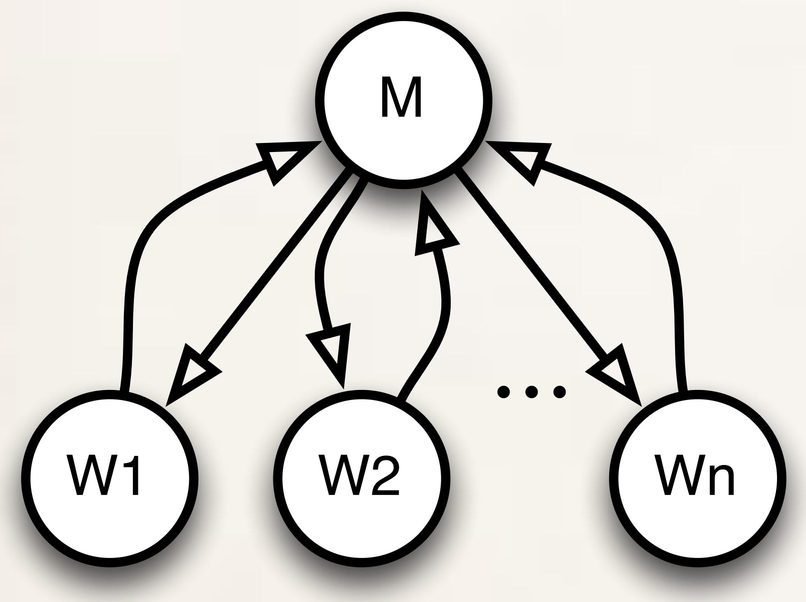
Xeon E7-4820 @2.0GHz Sandy Bridge

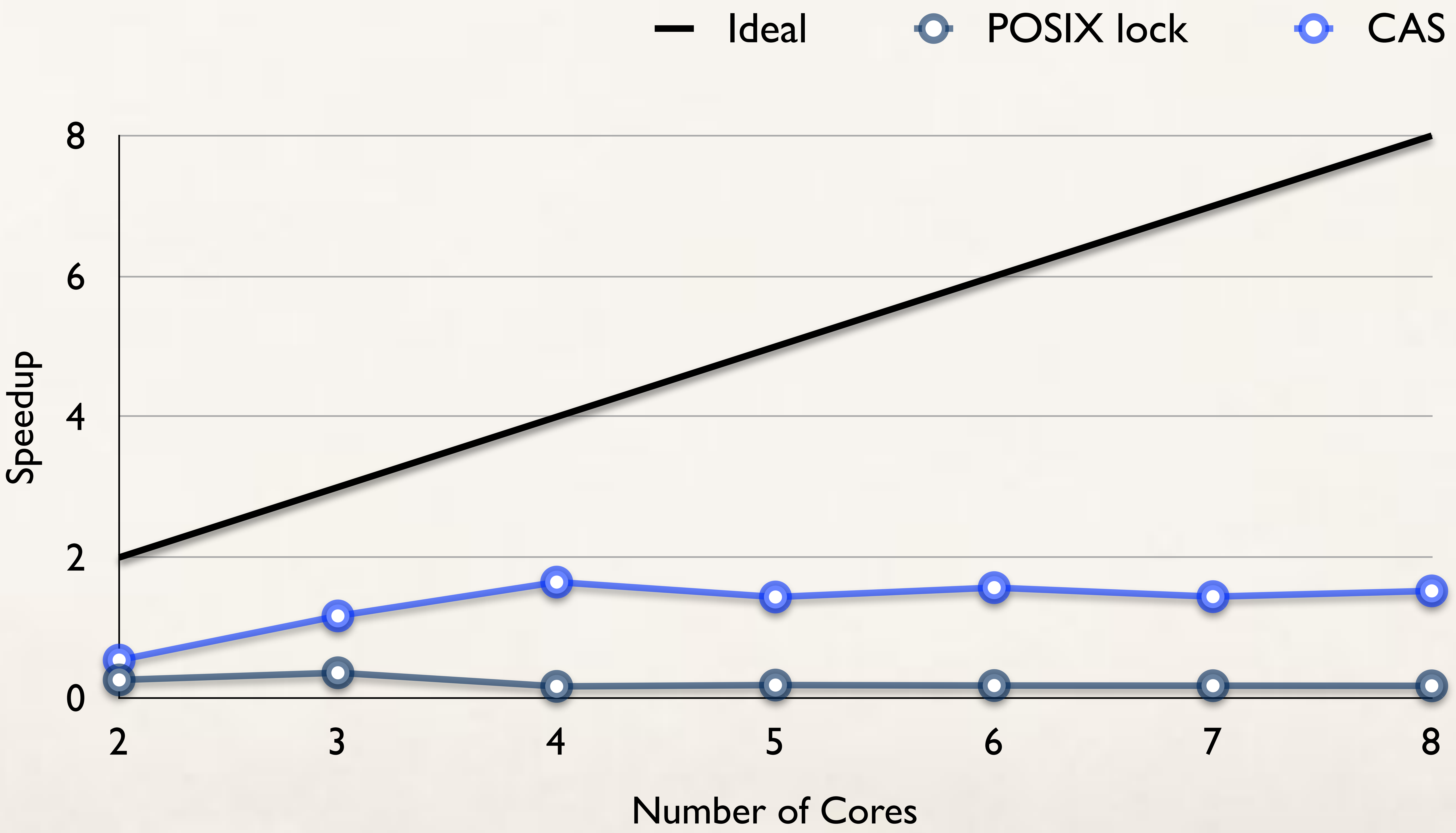


- MVAPICH ~ 190ns
- faster and more scalable than CAS/test-and-set implement.

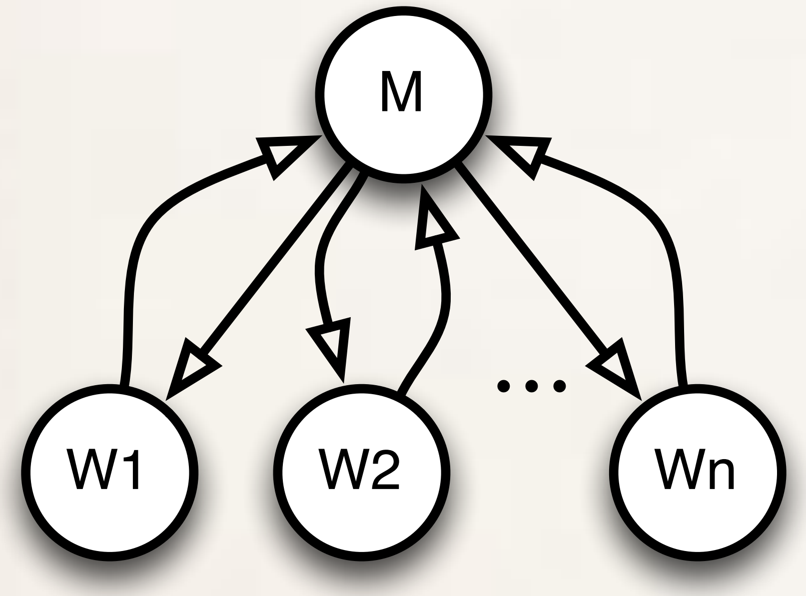
M. Aldinucci and M. Danelutto and P. Kilpatrick and M Meneghin. An Efficient Synchronisation Mechanism for Multi-Core Systems. Euro-Par 2012. LNCS.

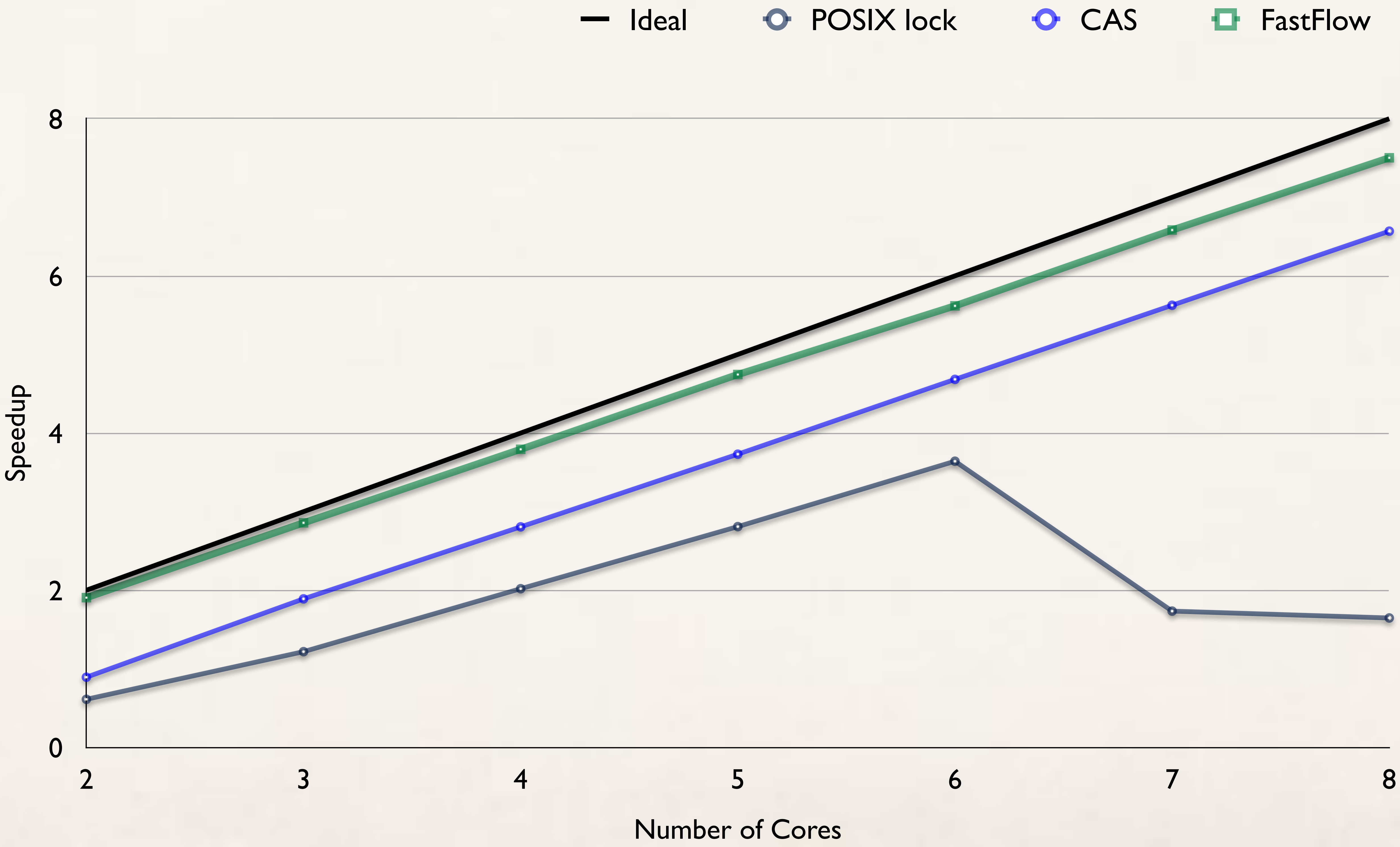
Master-workers (0.5 µS workload)



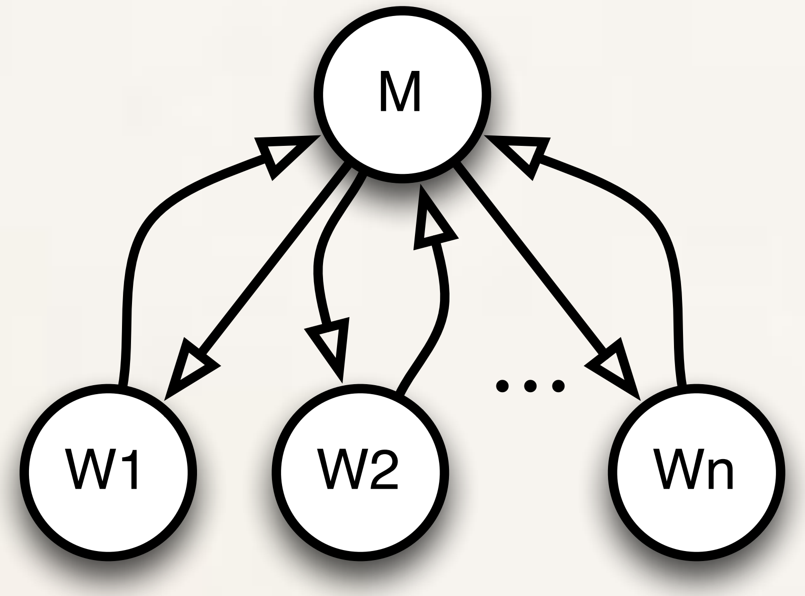


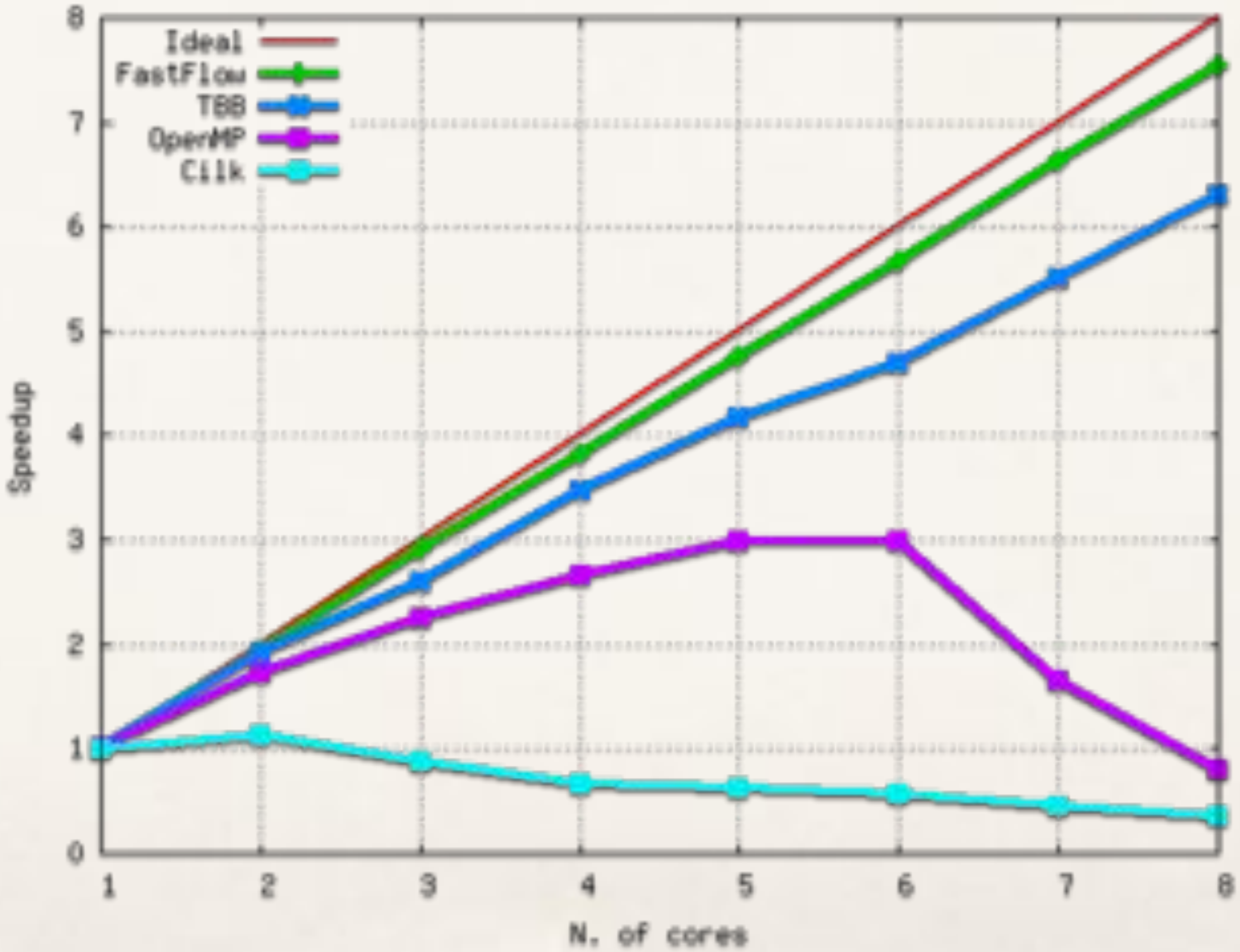
Master-workers (5 μS workload)



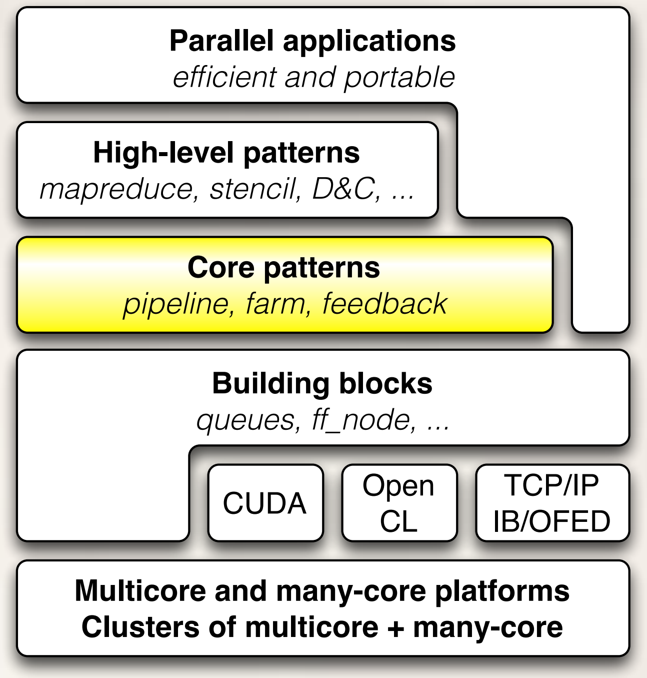


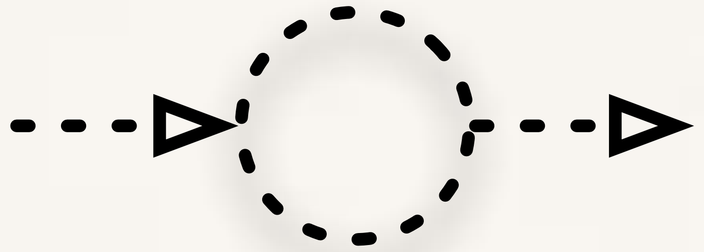
Master-workers (5 μS workload)





Semantics of the node: dataflow activation



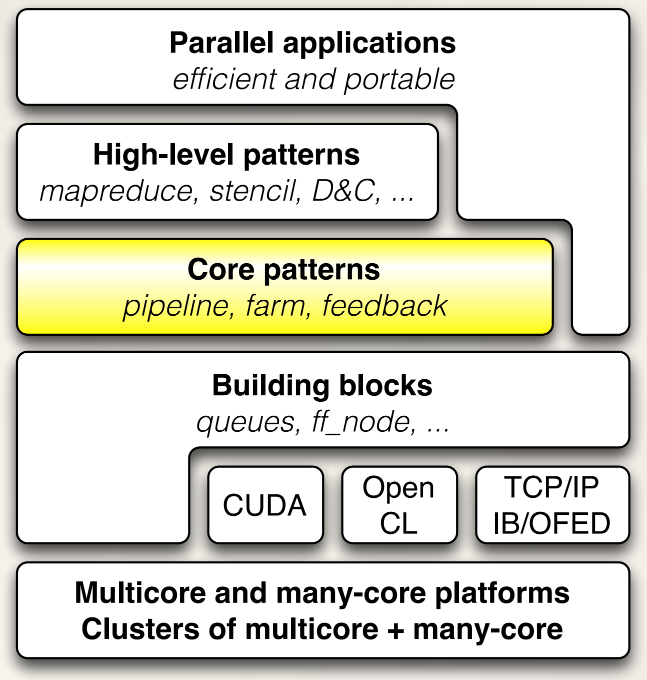


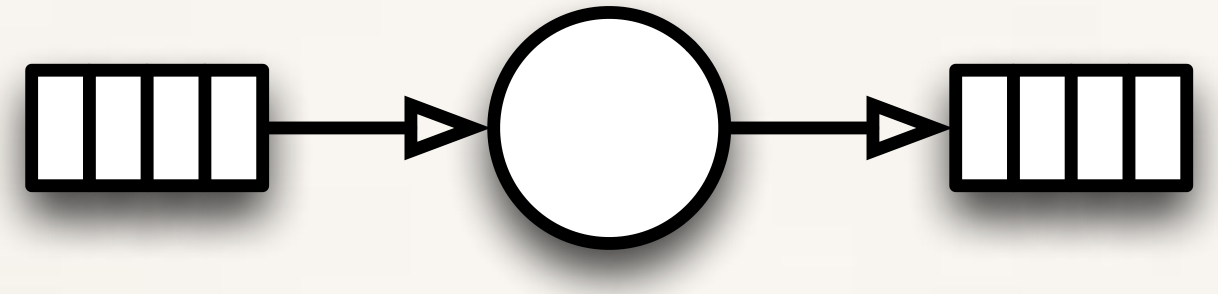
ff_node

mynode is created as a standard C++ class extending ff_node

```
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





ff_node

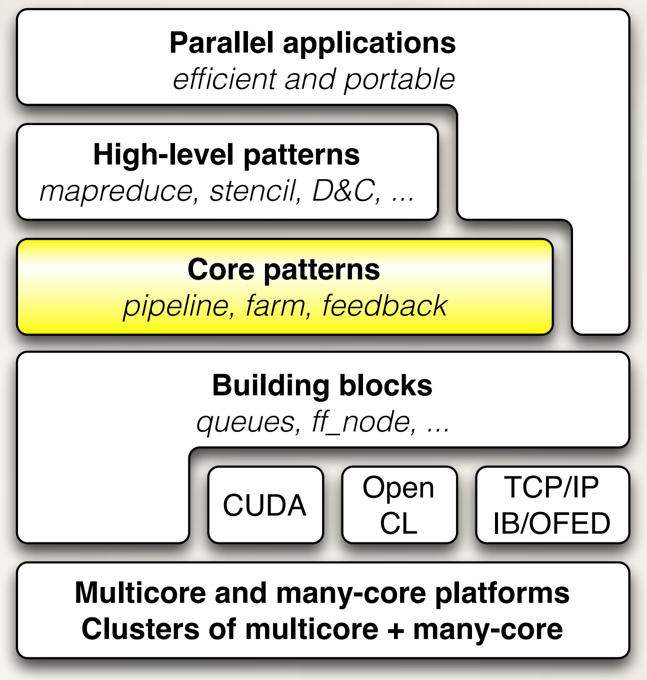
After class construction
mynode is turn into a thread

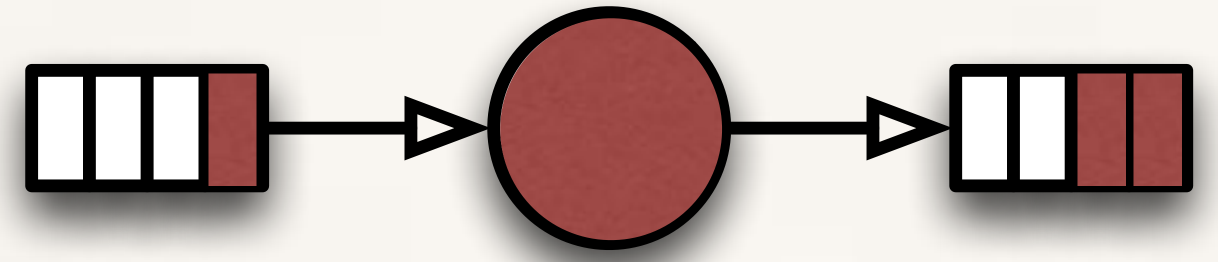
Nodes are not tasks (as in TBB), they are executors

The svc_init() method is executed

```
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





ff_node

the node enters in a infinite loop

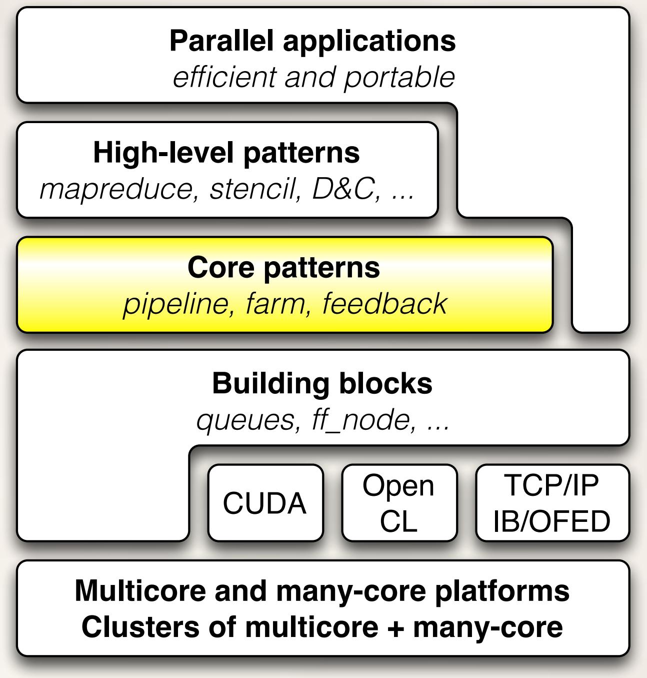
1. get a task from input channel (i.e. a pointer)
2. execute svc method
3. put a task the output channel (i.e. a pointer)

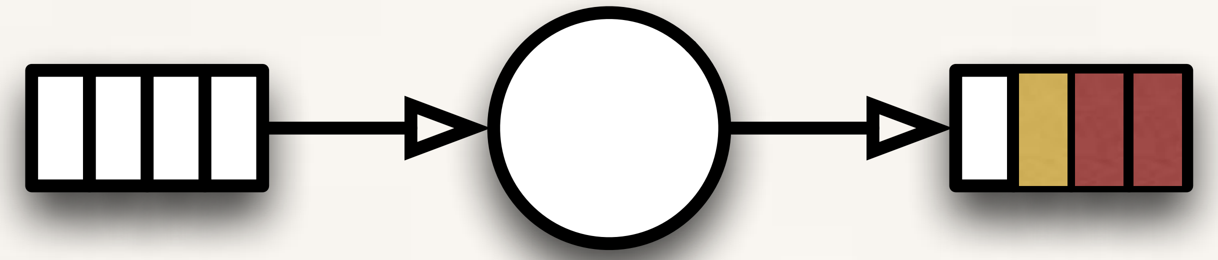
svc() might output more tasks via ff_send_out call (not shown)

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 */
    }
};
```

Semantics of the node: dataflow activation





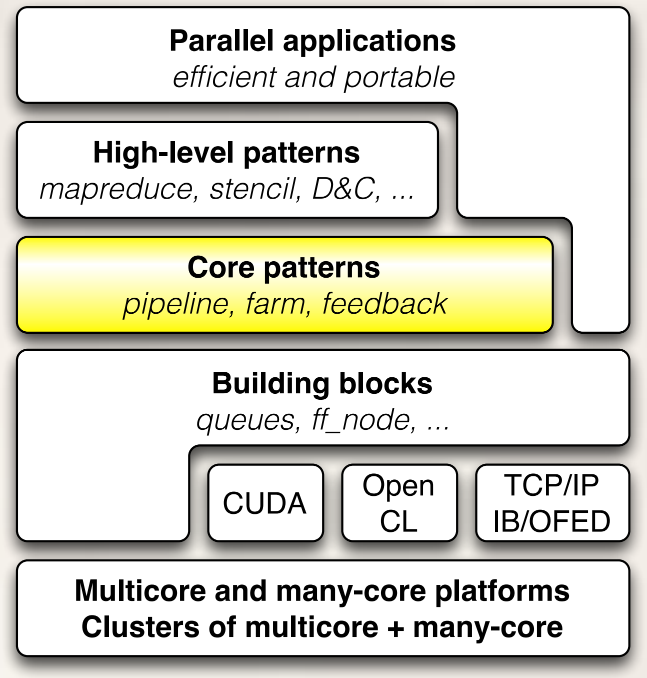
ff_node

svc_end is executed before class destruction

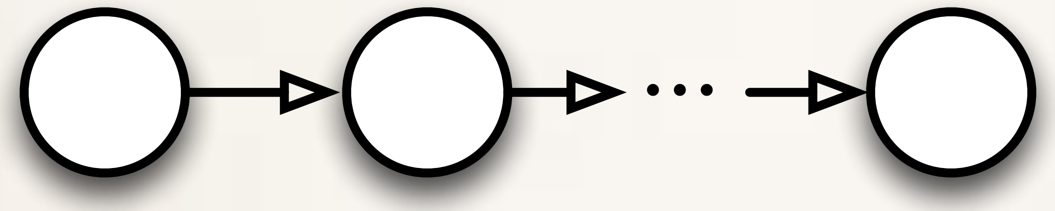
termination token is propagated to the next node

```
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 */
    }
};
```

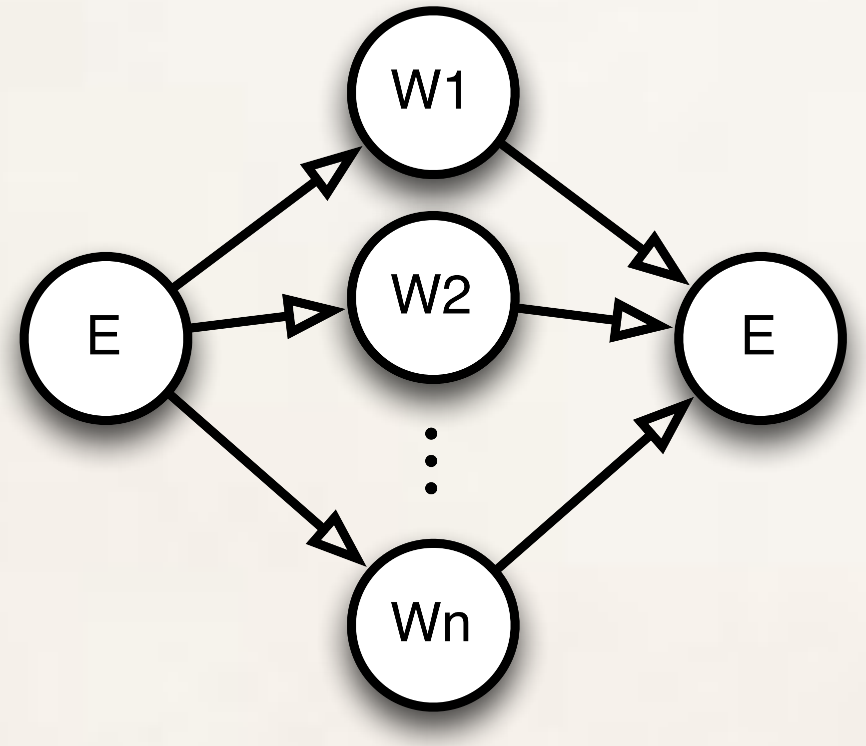
FF core patterns: pipe, farm, feedback
they are streaming networks, not task graphs



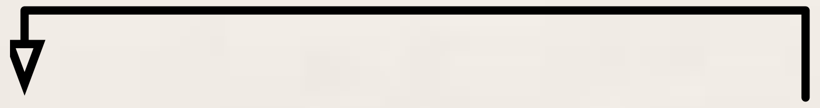
pipeline



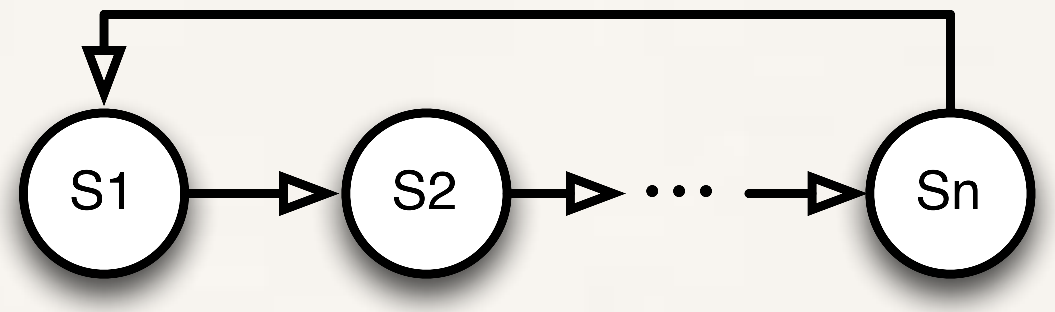
farm



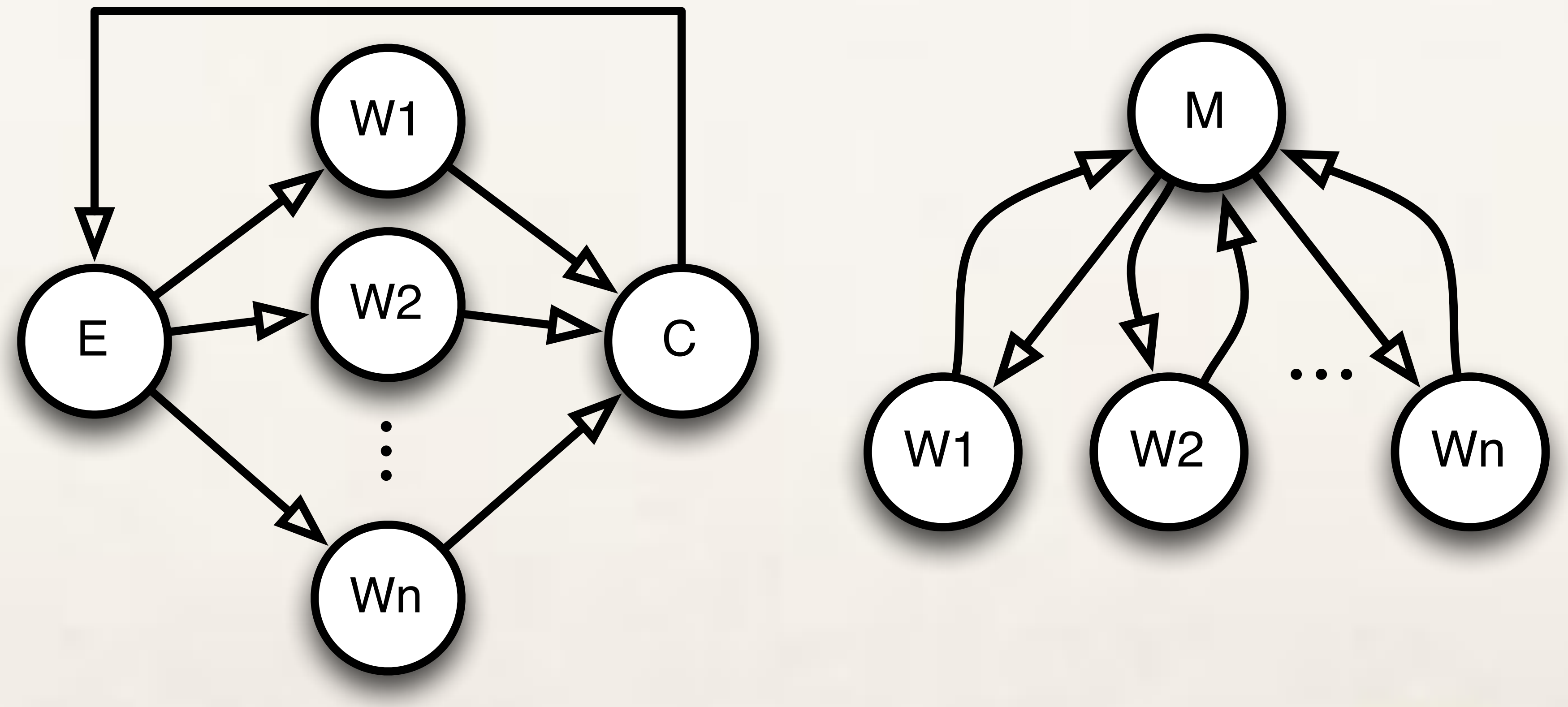
feedback



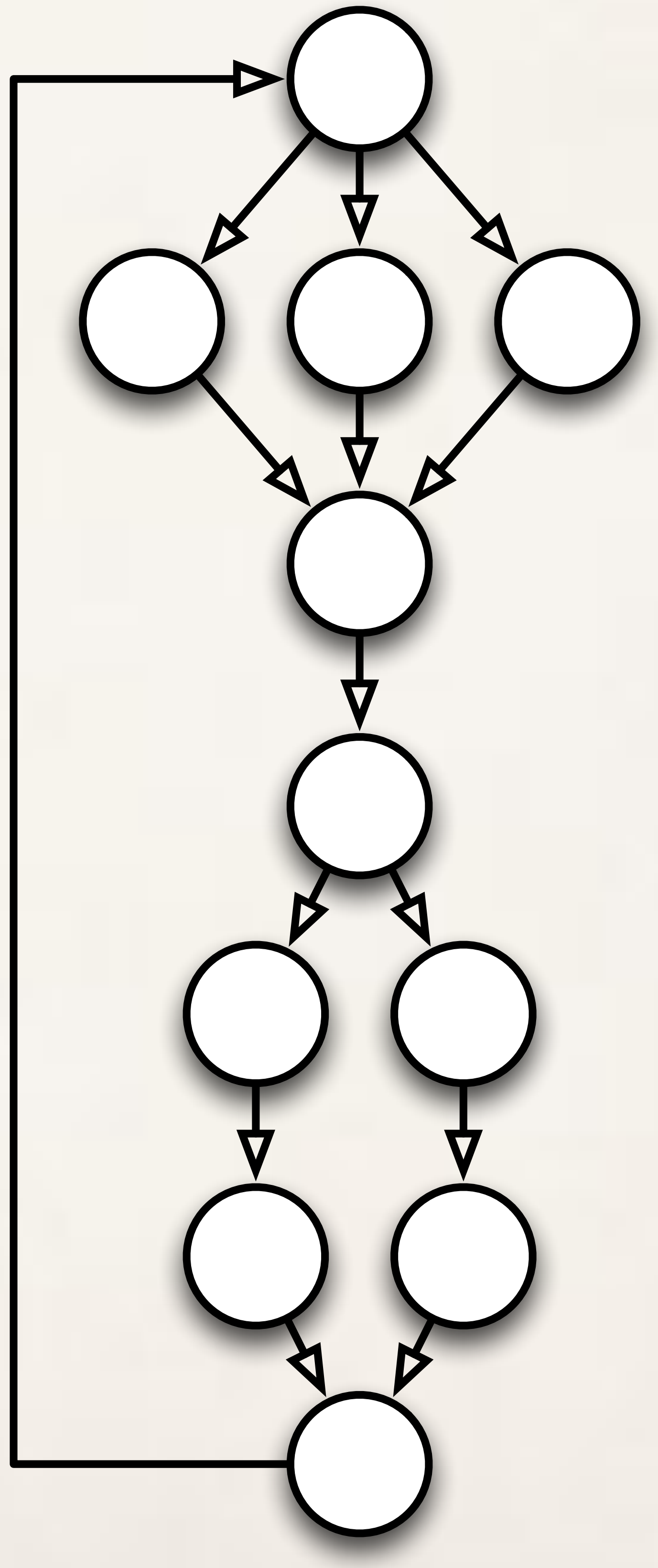
pipeline with feedback



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



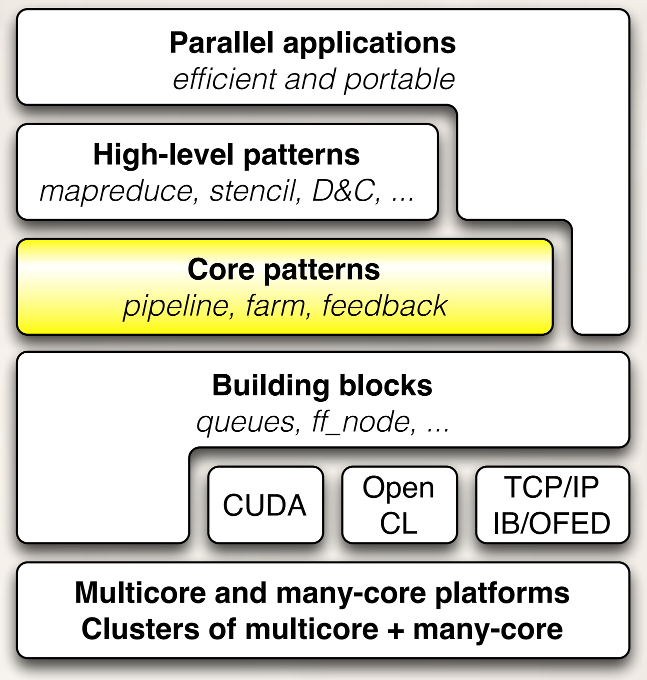
or any composition of them



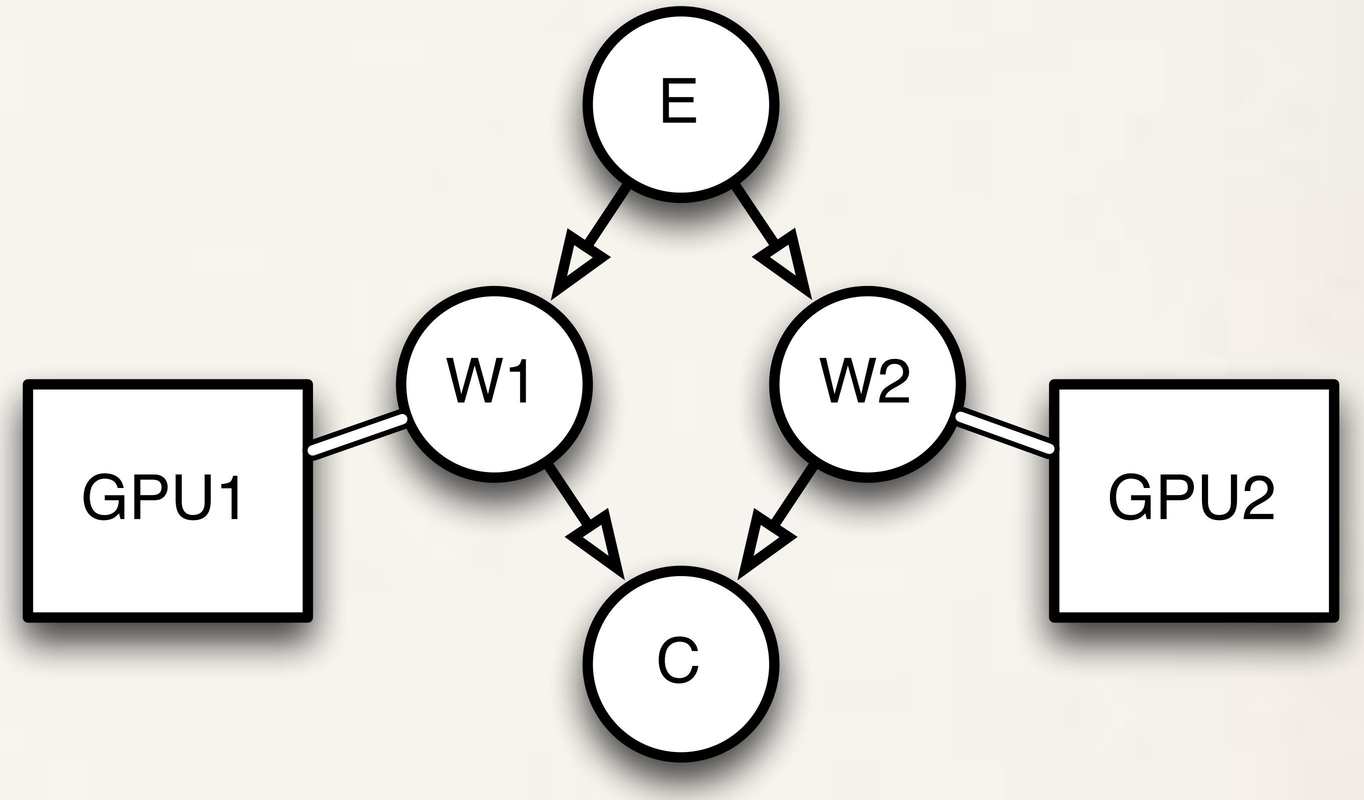
```cpp
#include <vector>
#include <iostream>
#include <ff/farm.hpp>


using namespace ff;

// generic worker
class Worker: public ff_node {
public:
    void * svc(void * task) {
        int * t = (int *)task;
        std::cout << "Worker " << ff_node::get_my_id()
                  << " received task " << *t << "\n";
        return task;
    }
    // I don't need the following functions for this test
    //int   svc_init() { return 0; }
    //void  svc_end() {}

};

// the gatherer filter
class Collector: public ff_node {
public:
    void * svc(void * task) {
        int * t = (int *)task;
        if (*t == -1) return NULL;
        return task;
    }
};

// the load-balancer filter
class Emitter: public ff_node {
public:
    Emitter(int max_task):ntask(max_task) {};

    void * svc(void *) {
        int * task = new int(ntask);
        --ntask;
        if (ntask<0) return NULL;
        return task;
    }
private:
    int ntask;
};
```

```cpp
int main(int argc, char * argv[]) {

    if (argc<3) {
        std::cerr << "use: "
                  << argv[0]
                  << " nworkers streamlen\n";
        return -1;
    }

    int nworkers=atoi(argv[1]);
    int streamlen=atoi(argv[2]);

    if (!nworkers || !streamlen) {
        std::cerr << "Wrong parameters values\n";
        return -1;
    }

    ff_farm<> farm; // farm object

    Emitter E(streamlen);
    farm.add_emitter(&E);

    std::vector<ff_node *> w;
    for(int i=0;i<nworkers;++i) w.push_back(new Worker);
    farm.add_workers(w); // add all workers to the farm

    Collector C;
    farm.add_collector(&C);

    if (farm.run_and_wait_end()<0) {
        error("running farm\n");
        return -1;
    }
    std::cerr << "DONE, time= " << farm.ffTime() << " (ms)\n";
    farm.ffStats(std::cerr);

    return 0;
}
```

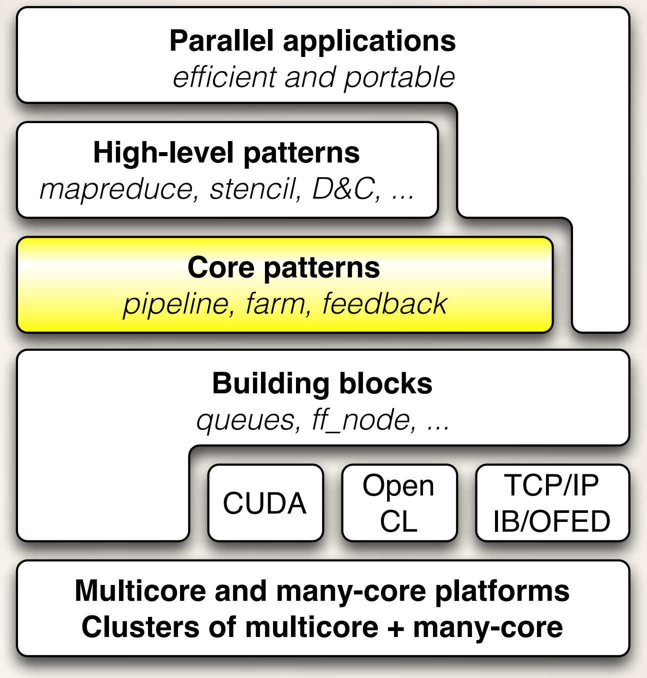
GPGPUs



- 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

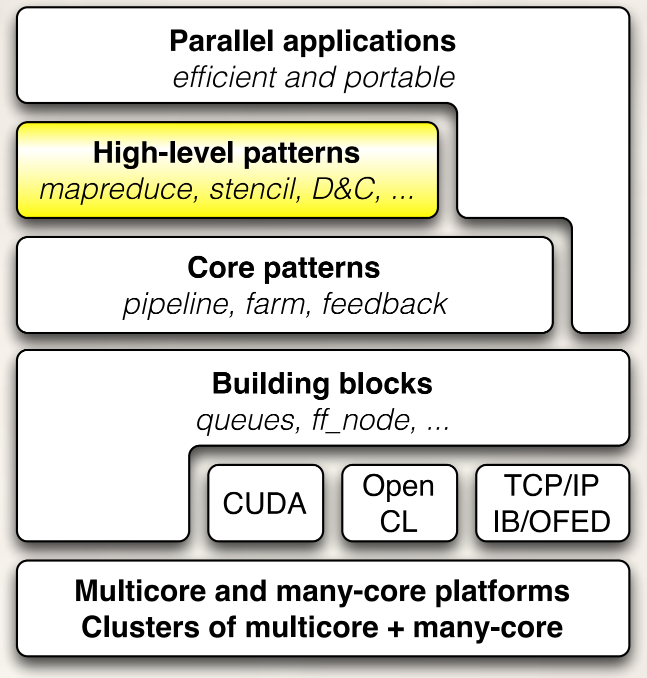


FF core patterns: rationale



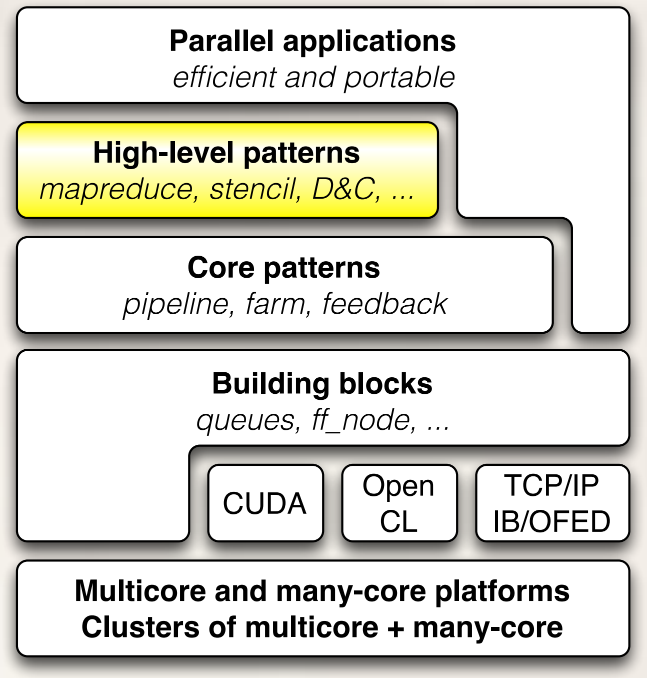
- farm: process in parallel independent tasks (e.g. C++ objects)
  - true dependencies are enforced only along edges of the graph
  - workers might synchronise (e.g. w locks/atomics), synchronisation in the business code
- farm, pipeline and feedback (to build cyclic networks) are enough to write all 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



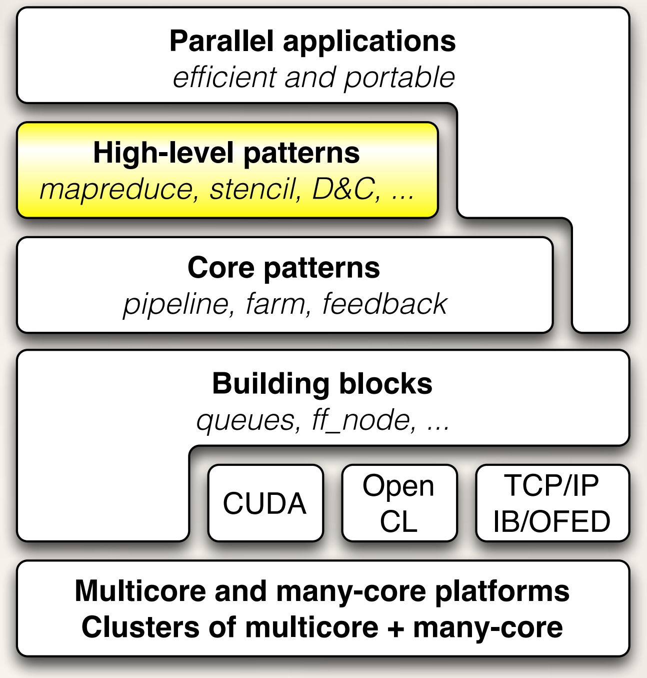
- Proposed as code annotations
  - Similarly to openMP, openacc, …
  - used to generate a graph at the core pattern level
- Examples
  - parallel_for
  - map, reduce, MapReduce, … (targeting GPGPUs)
  - and as many as you want: developing a new pattern is just developing a new class

Example: map (derived from stencilReduce)



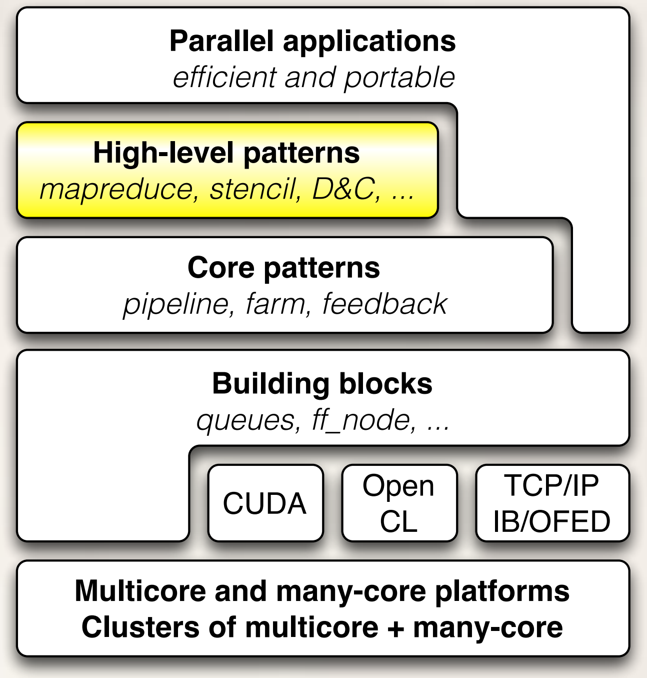
- 2 or more GPGPUs on the same platform
  - nest a stencilReduce, map, reduce … into a (host) farm with 2 workers
  - (future: we need to understand how to use NVLINK)
- offload code onto distributed GPGPUs
  - nest a stencilReduce, map, reduce … into a (host) distributed farm with 2 workers
  - data 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)




```
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
…
}
```

Example: map (derived from stencilReduce)




```
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)

GPGPU code: Rationale

- Is it worth abstract even more?
  - In particular, the the CUDA code
- No, we believe
  - Often not needed: CUDA code if often C++ code
  - Access to thread_id, always needed
  - Programmers would like to super-optimize their code
    - using all CUDA features
  - CUDA evolves too rapidly

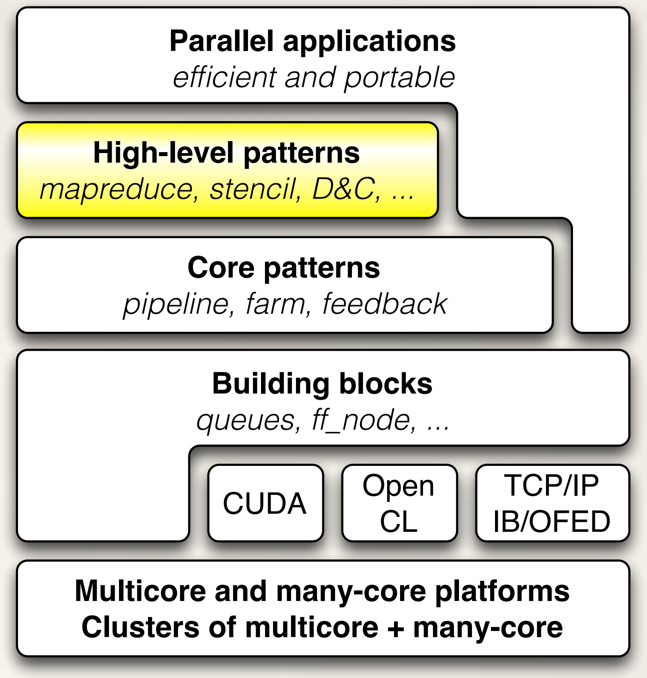
“Demo”

map, farm(map), …

“Demo”

denoiser

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)

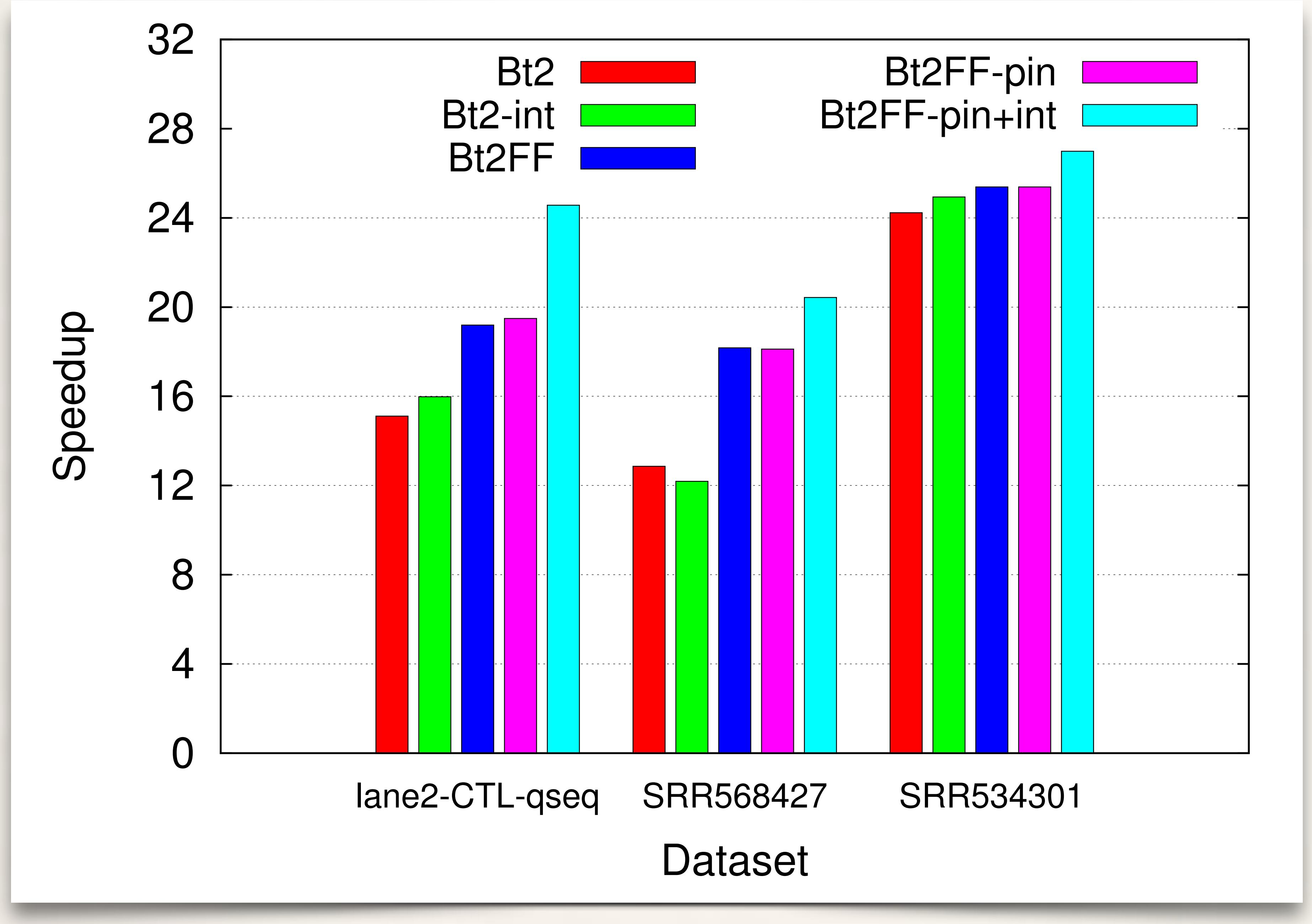
```
…
const int Limit = 4;
bool allBlack = true;
if (restart) break;
if (abort)   return;

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”

mandelbrot

Performance (multicore)

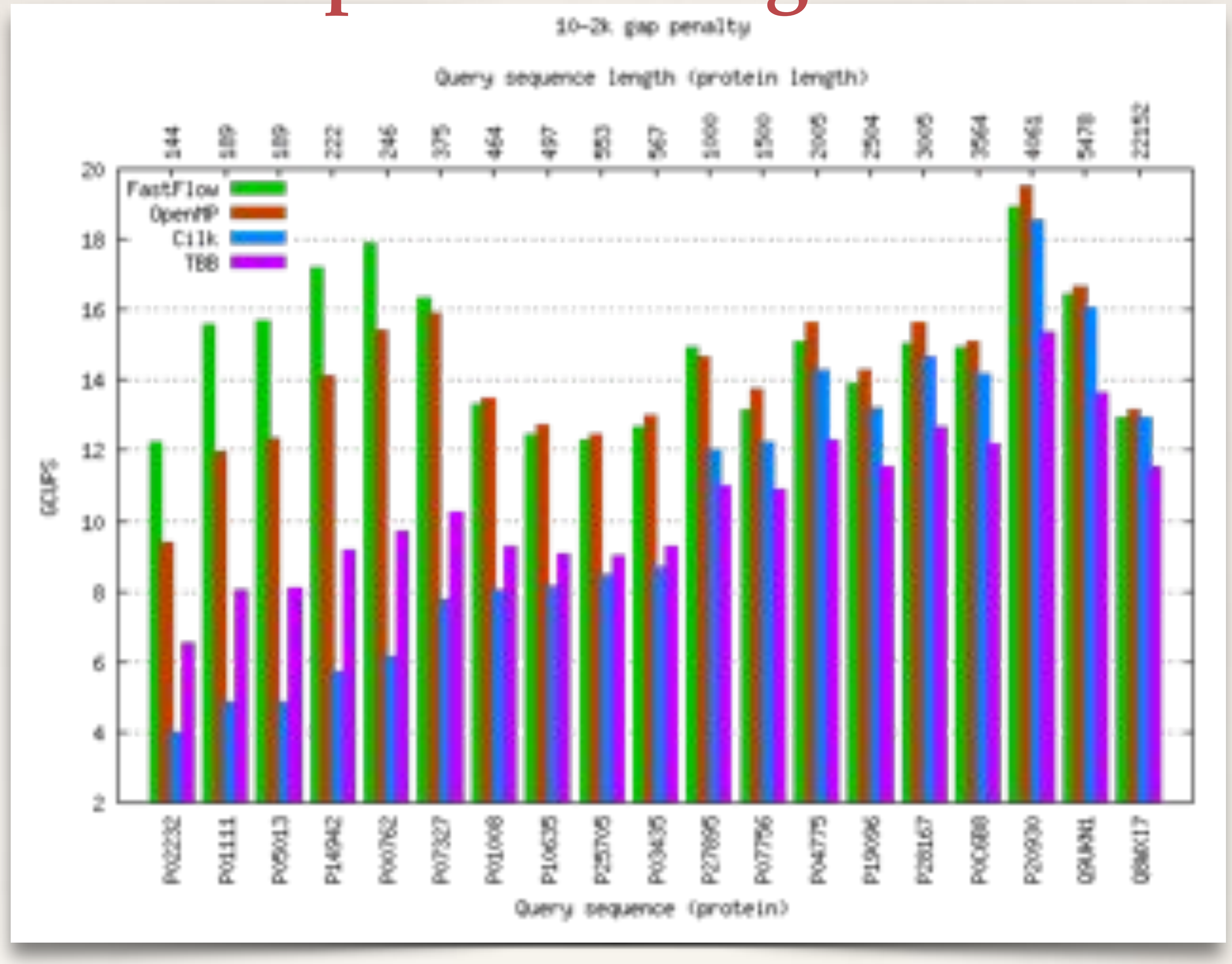
Bowtie2 tool (DNA mapping)



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)

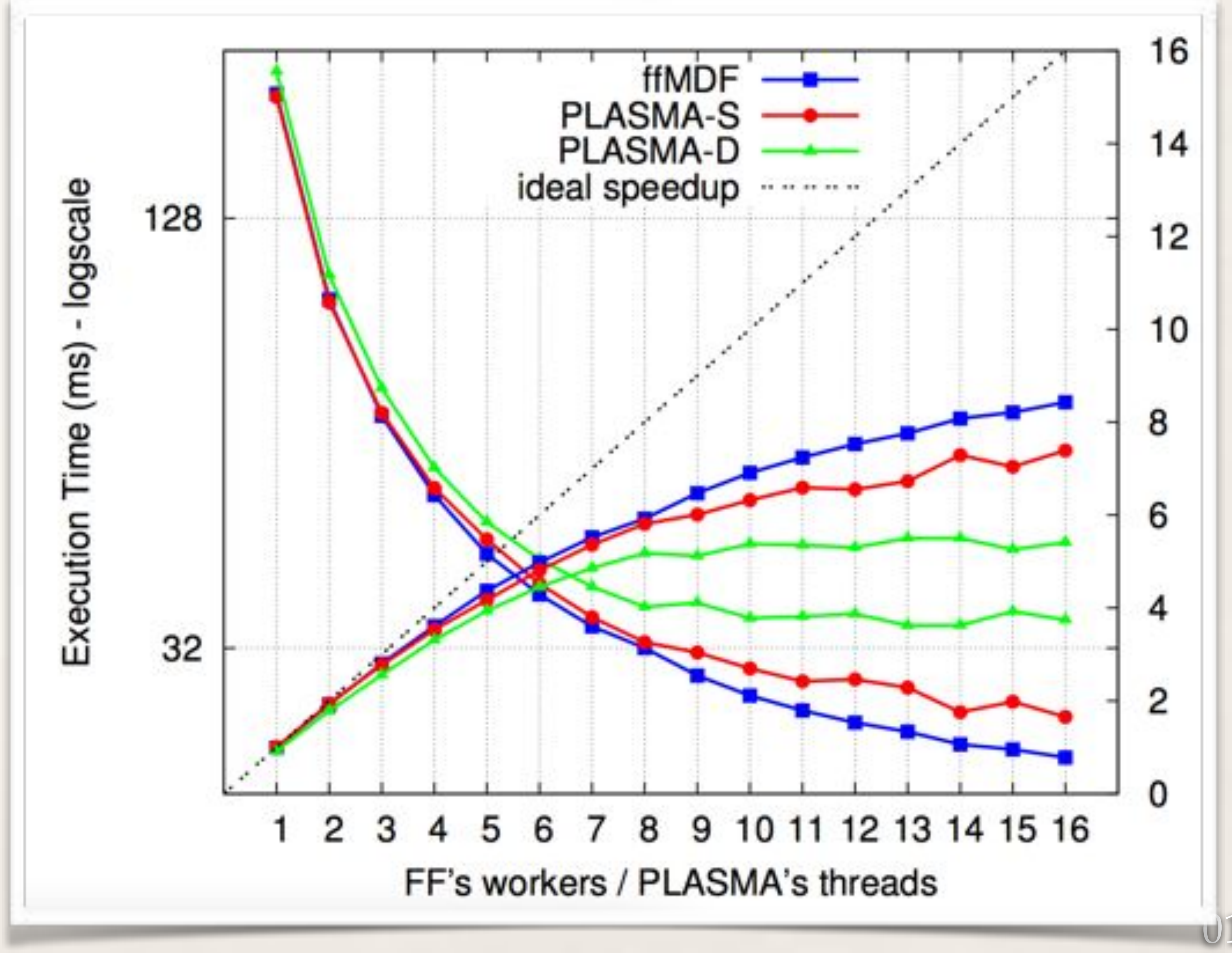
proteine length →

speed →



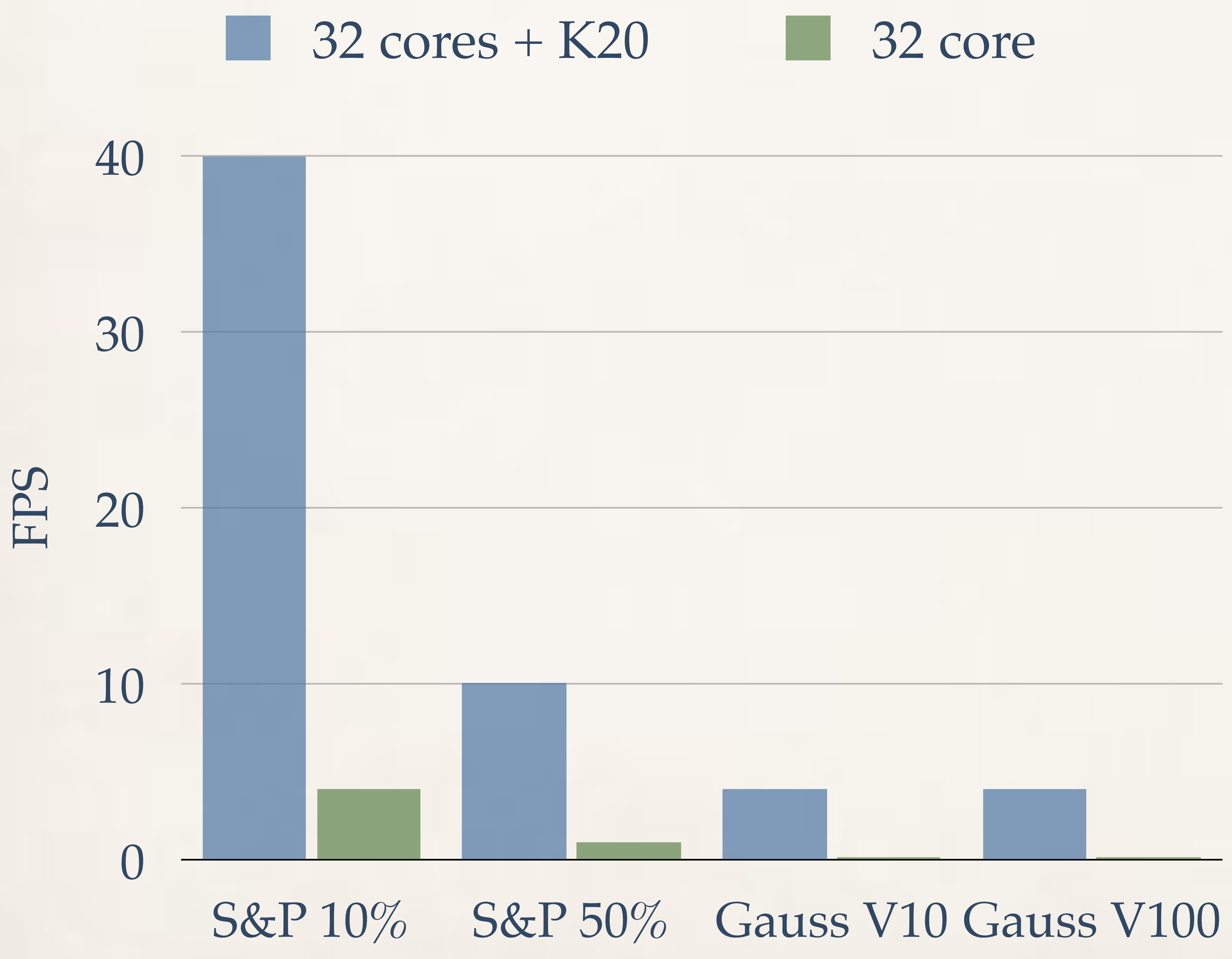
Smith-Waterman (SSE2) against OpenMP, Cilk, TBB

Cholesky LU against PLASMA

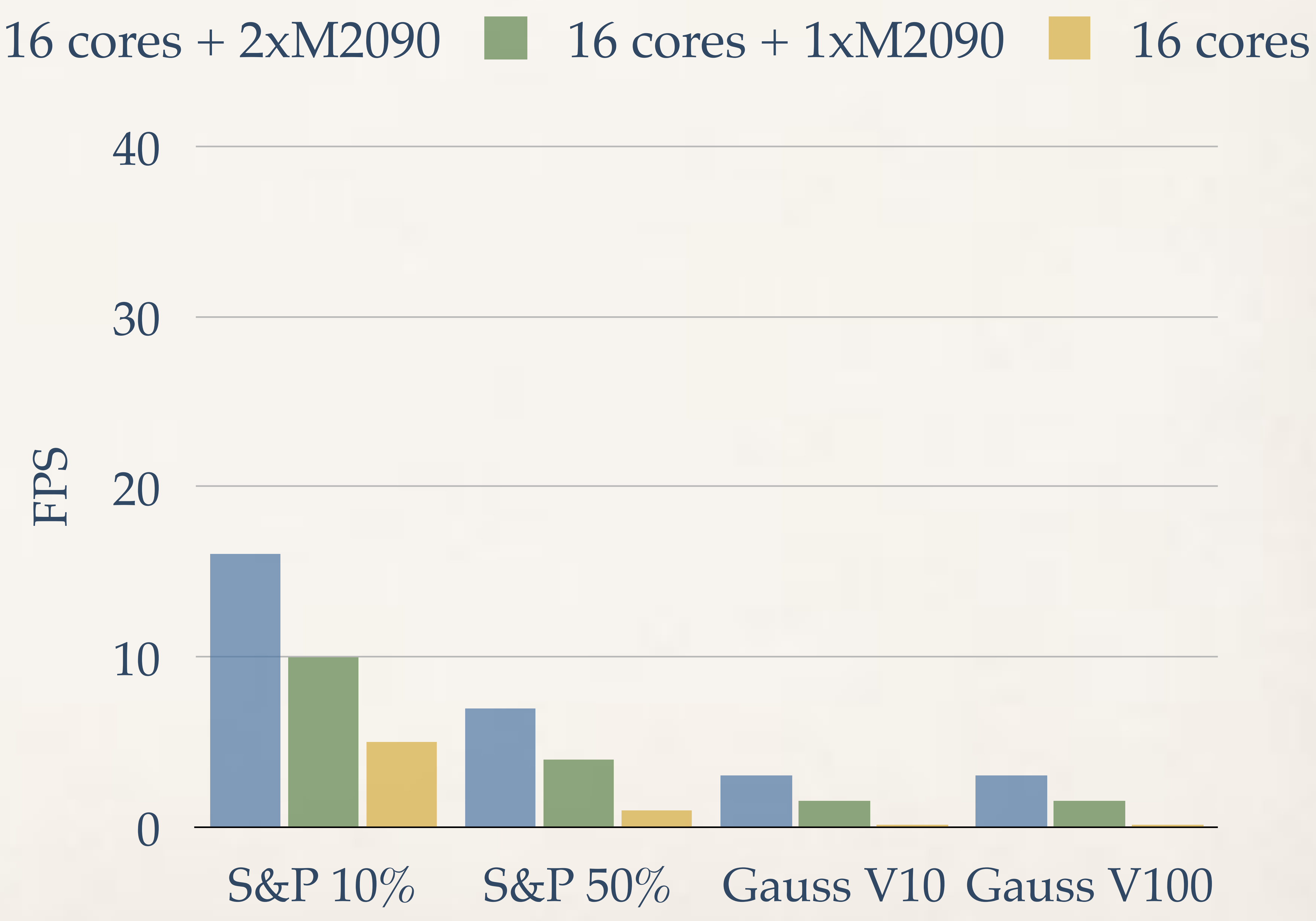


Performance (CPUs + GPGPUs) Video frames 768x512

Nehalem 32 cores + 1 K20



SandyBridge 16 cores + 2 Tesla M2090



no difference w.r.t. hand-written CUDA code

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, reduced development time, 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
- 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

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, reduced development time, 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
- 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

Thanks

University of Turin


M. Aldinucci


G. Peretti


A. Secco


F. Tordini


M. Drocco


C. Misale

University of Pisa


M. Torquati


M. Danelutto

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

NVIDIA CUDA RESEARCH CENTER

PARAPHRASE
EU-FP7 - 3.5M€

IMPACT
Unito - 440K€

REPARA
EU-FP7 - 3.7M€

HiPEAC
EU-FP7 Network of Excellence