Par4All

—

Automatic parallelization for CPU & GPU

Mehdi Amini Antoine Bernardeau Béatrice Creusillet Amaury Darsch Stéphanie Even Serge Guelton Vincent Guguin Ronan Keryell Onil Nazra Persada Goubier Janice Onanian McMahon François-Xavier Pasquier Grégoire Péan Thierry Porcher Claire Seguin Mickaël Thiévent Pierre Villalon + *many interns*

HPC Project & Institut TÉLÉCOM/TÉLÉCOM Bretagne/HPCAS

2010/10/25

http://www.par4all.org

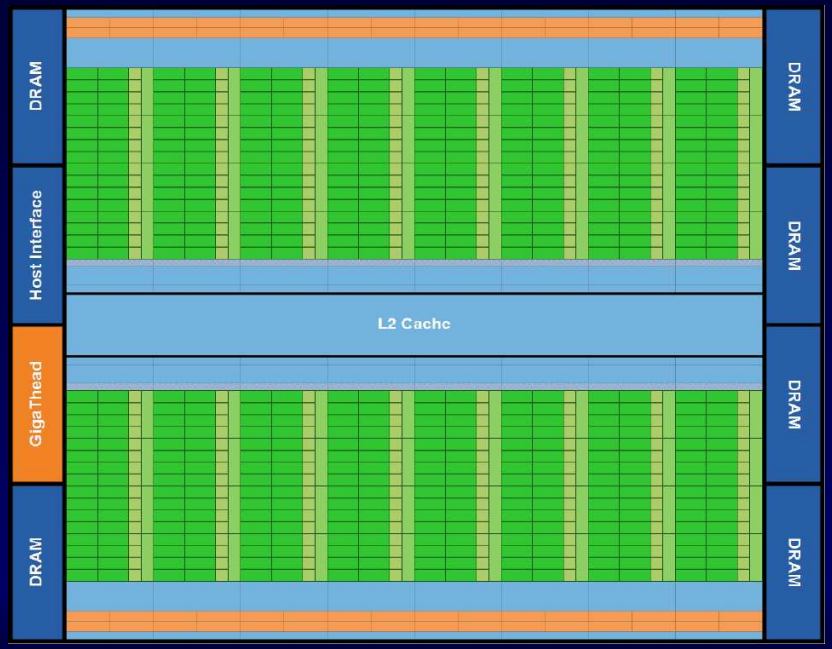
TechNoCloCgy shrinking

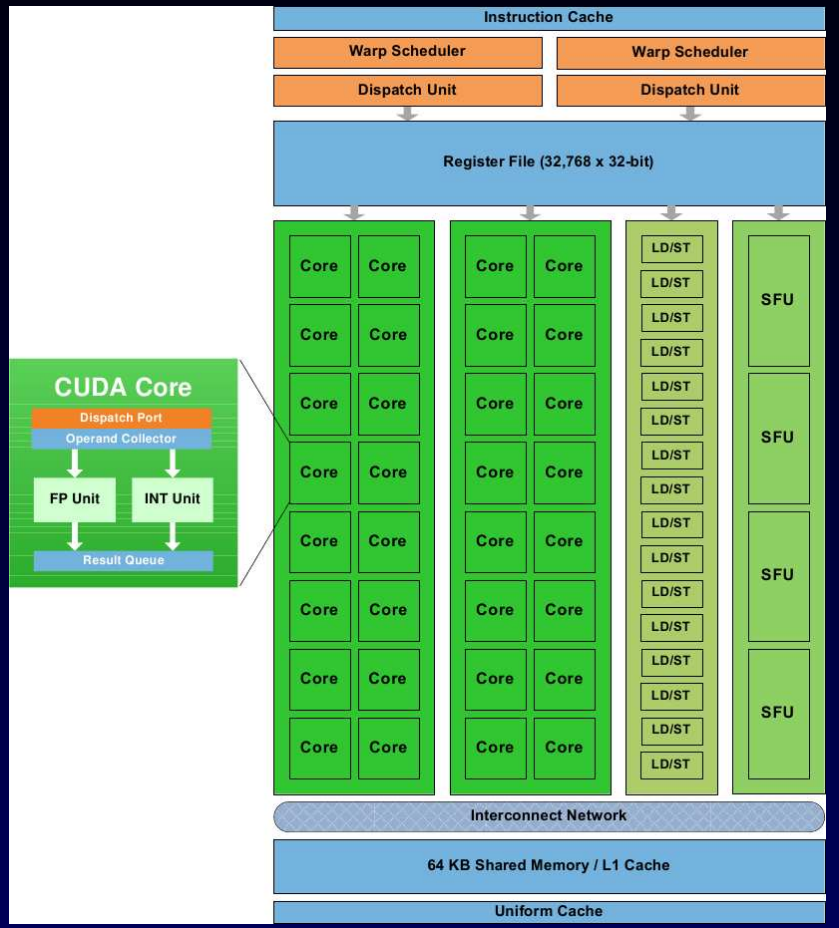
MOORE's law revisited *à la* Berkeley:

- System... $\rightsquigarrow$ on Chip (SoC)
- Multi-Processor System... $\rightsquigarrow$ on Chip (MP-SoC)
- Network... $\rightsquigarrow$ on Chip (NoC)
- Data center... $\rightsquigarrow$ on Chip
- Heat... $\rightsquigarrow$ on Chip ☹

Off-the-shelf nVidia Tesla Fermi

- 3 billion 40nm transistors
- 448 thread processors @ 1150 MHz, 1 TFLOPS SP, 0.5 TFLOPS DP
- + External 6 GB GDDR5 memory 3 Gt/s, 144 GB/s. Less if using ECC





- 247 W on board PCI Express 2.1 x16 bus interface
- OpenGL, OpenCL, CUDA

Off-the-shelf AMD/ATI Radeon HD 5870 GPU

- 2.15 billion 40nm transistors
- 1600 stream processors @ 850 MHz, 2.7 TFLOPS SP, 0.5 TFLOPS DP
- + External 1 GB GDDR5 memory 4.8 Gt/s, 153.6 GB/s
- 188 W on board (27 idle), PCI Express 2.1 x16 bus interface
- OpenGL, OpenCL

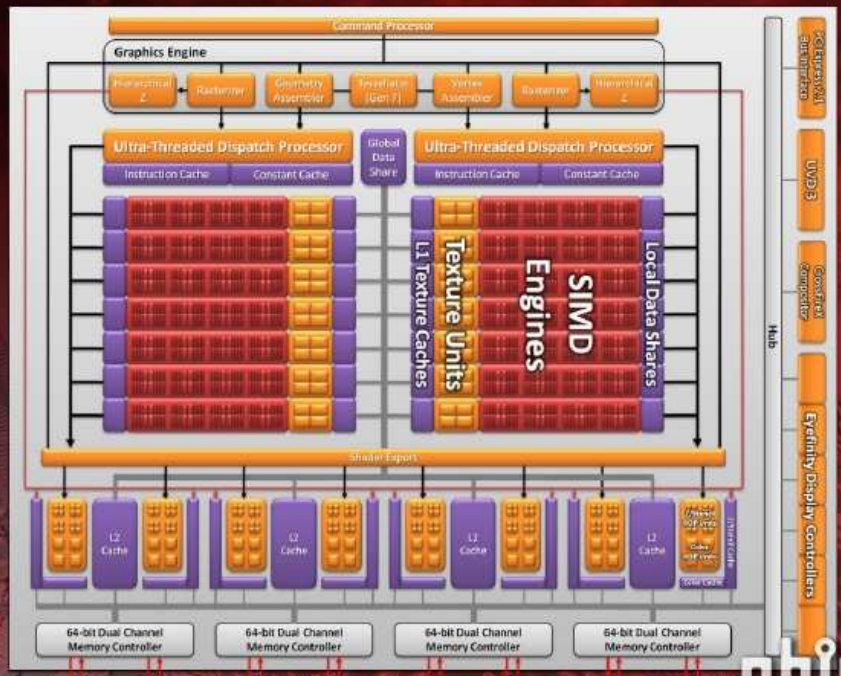
New stuff (estimations...):

- Radeon HD 6870 1.8Mtr (40nm), 960 SP, 134 MB/s 256b GDDR5, 2 TFLOPS, 151 (19) W (2010/10/22)
- Radeon HD 6970, 3Mtr (40nm), 1920 SP, 230 MB/s 384b GDDR5, 3.2 TFLOPS, 285 (50) W (2010/11 ?)
- Llano APU (FUSION Accelerated Processing Unit) : x86 multicore + GPU 32nm, OpenCL (1H 2011)

Radeon HD 6870 ?

AMD Radeon™ HD 6800 Series

- Reconfigured core design
  - Up to 2.0 TeraFLOPS
  - Over 24 Gigapixels/sec
- Enhanced tessellation unit
- Dual rasterizers
- 12-14 SIMD engines
- 256-bit GDDR5 memory interface
- Better performance than the ATI Radeon™ HD 5850, 25% less silicon[1]



3 | AMD Radeon™ HD 6800 Series | October 2010 | AMD Confidential - NDA Required

* See slide 19 for additional information

More performances: nothing but parallelism!

2 previous start-ups in //ism:

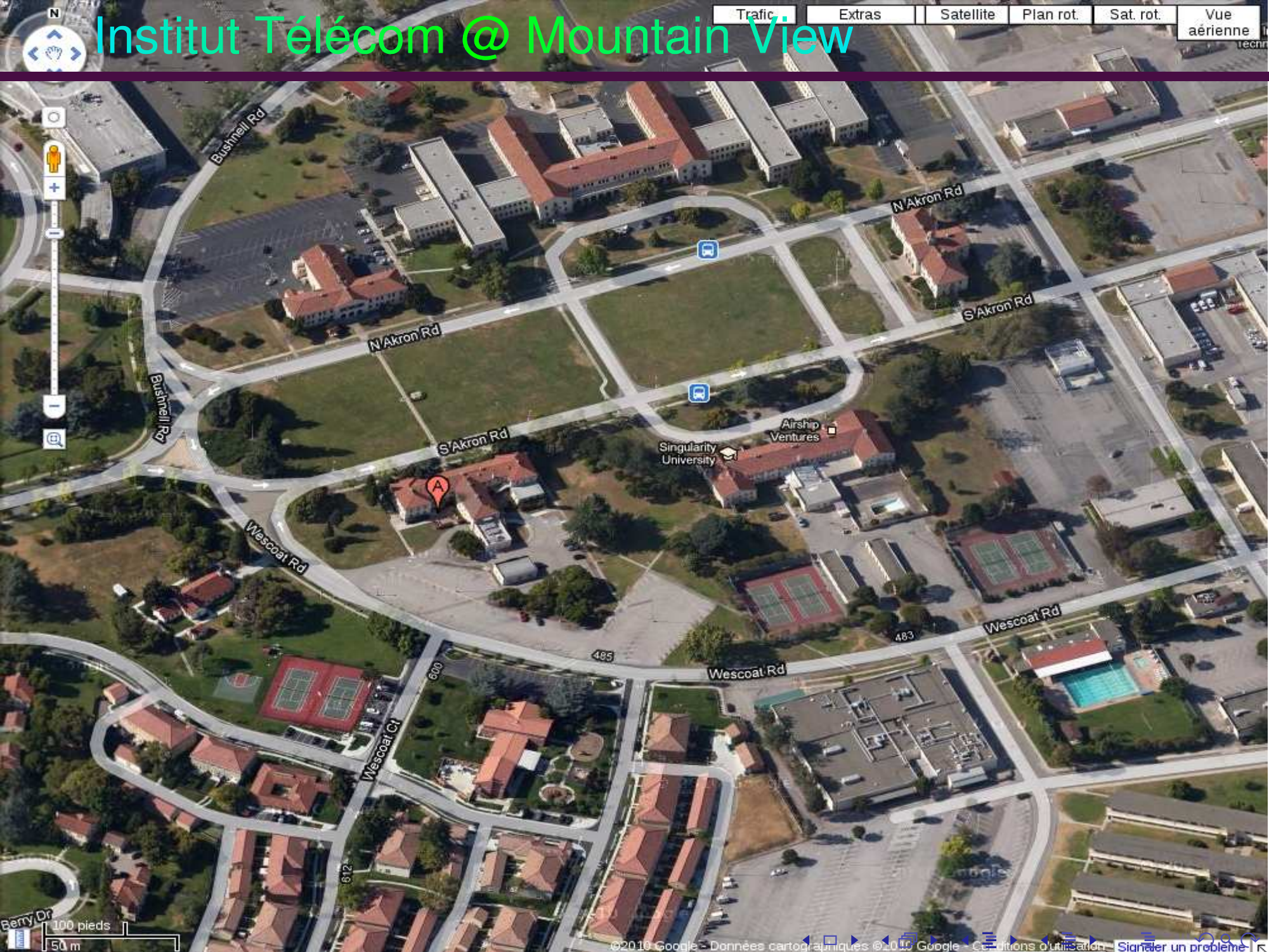
- 1986: Minitel servers with clusters of Atari 1040ST (128 users/Atari!), MIDI LAN ☺, PC+X25 cards as front-end
- 1992: HyperParallel Technologies (Alpha processors, 3D-torus, HyperC language)

⇝ 2006: Time to be back in parallelism!

Yet another start-up... ☺

- People that met ≈ 1990 at SEH/ETCA: researchers in Computer Science, CINES director, venture capital and more: ex-CEO of Thales Computer, HP marketing...
- ≈ 25 colleagues in Montpellier, Meudon, Montréal & Mountain View

Institut Télécom @ Mountain View



HPC Project hardware: WildNode from Wild Systems

Through its Wild Systems subsidiary company

- WildNode hardware desktop accelerator
  - Low noise for in-office operation
  - *x*86 manycore
  - nVidia Tesla GPU Computing
  - Linux & Windows

- WildHive
  - Aggregate 2-4 nodes with 2 possible memory views
    - Distributed memory with Ethernet or InfiniBand
    - Virtual shared memory through Linux Kerrighed for single-image system

`http://www.wild-systems.com`

HPC Project software and services

- Parallelize and optimize customer applications, co-branded as a bundle product in a WildNode (e.g. Presagis Stage battle-field simulator)
- Acceleration software for the WildNode
  - GPU-accelerated libraries for Scilab/Matlab/Octave/R
  - Transparent execution on the WildNode
- Remote display software for Windows on the WildNode

HPC consulting

- Optimization and parallelization of applications
- *High Performance?...* not only TOP500-class systems: power-efficiency, embedded systems, green computing...
- $\rightsquigarrow$ Embedded system and application design
- Training in parallel programming (OpenMP, MPI, TBB, CUDA, OpenCL...)

Extracting parallelism in applications...

- The implicit attitude
  - Hardware: massively superscalars processors
  - Software: auto-parallelizing compilers
- The ($\pm$) explicit attitude
  - Languages ($\pm$ extensions): OpenMP, UPC, HPF, Co-array Fortran (F- -), Fortran 2008, X10, Chapel, Fortress, Matlab, SciLab, Octave, Mapple, LabView, nVidia CUDA, AMD/ATI Stream (Brook+, Cal), OpenCL, HMPP, *insert your language here* ...
  - Libraries: application-oriented (mathematics, coupling...), parallelism (MPI, concurrency *pthreads*, SPE/MFC on Cell...), Multicore Association MCAPI, objects (parallel STL, TBB, Ct...)

... but multidimensional heterogeneity!

Welcome into Parallel Hard-Core Real Life 2.0!

- Heterogeneous execution models
  - Multicore SMP ± coupled by caches
  - SIMD instructions in processors (Neon, VMX, SSE4.2, 3DNow!, LRBni...)
  - Hardware accelerators (MIMD, MISD, SIMD, SIMT, FPGA...)
- New heterogeneous memory hierarchies
  - Classic caches/physical memory/disks
  - Flash SSD is a new-comer to play with
  - NUMA (*Non Uniform Memory Access*) : sockets-attached memory banks, remote nodes...
  - Peripherals attached to sockets : NUPA (*Non Uniform Peripheral Access*). GPU on PCIe ×16 in this case...
  - If non-shared memory: remote memory, remote disks...
  - Inside GPU : registers, local memory, shared memory, constant memory, texture cache, processor grouping, locked physical pages, host memory access...
- Heterogeneous communications
  - Anisotropic networks
  - Various protocols

Several dimensions to cope with at the same time ☹

Outline

We need software tools

- HPC Project needs tools for its hardware accelerators (*Wild Nodes* from *Wild Systems*) and to parallelize, port & optimize customer applications
- Application development: long-term business $\rightsquigarrow$ long-term commitment in a tool that needs to survive to (too fast) technology change

Use the Source, Luke...

Hardware is moving quite (too) fast but...

What has survived for 50+ years?

Fortran programs...

What has survived for 30+ years?

C programs, Unix...

- A lot of legacy code could be pushed onto parallel hardware (accelerators) with automatic tools...
- Need automatic tools for source-to-source transformation to leverage existing software tools for a given hardware
- Not as efficient as hand-tuned programs, but quick production phase

Not reinventing the wheel

Want to create your own tool?

- House-keeping and infrastructure in a compiler is a huge task
- Unreasonable to begin yet another new compiler project...
- Many academic Open Source projects are available...
- ...But customers need products ☺
- ⇝ Integrate your ideas and developments in existing project
- ...or buy one if you can afford (ST with PGI...) ☺
- Some projects to consider
  - Old projects: gcc, PIPS... and many dead ones (SUIF...)
  - But new ones appear too: LLVM, RoseCompiler, Cetus...

Par4All

- ⇝ Funding an initiative to industrialize Open Source tools
- PIPS is the first project to enter the Par4All initiative

`http://www.par4all.org`

PIPS (I)

- PIPS (Interprocedural Parallelizer of Scientific Programs): Open Source project from Mines ParisTech... 22-year old!
- Funded by many people (French DoD, Industry & Research Departments, University, CEA, IFP, Onera, ANR (French NSF), European projects, regional research clusters...)
- One of the project that coined polytope model-based compilation
- $\approx$ 456 KLOC according to David A. Wheeler's `SLOCCount`
- ... but modular and sensible approach to pass through the years
  - $\approx$300 phases (parsers, analyzers, transformations, optimizers, parallelizers, code generators, pretty-printers...) that can be combined for the right purpose
  - Polytope lattice (sparse linear algebra) used for semantics analysis, transformations, code generation... to deal with big programs, not only loop-nests

PIPS (II)

- NewGen object description language for language-agnostic automatic generation of methods, persistence, object introspection, visitors, accessors, constructors, XML marshaling for interfacing with external tools...
- Interprocedural *à la* `make` engine to chain the phases as needed. Lazy construction of resources
- On-going efforts to extend the semantics analysis for C

- Around 15 programmers currently developing in PIPS (Mines ParisTech, HPC Project, IT SudParis, TÉLÉCOM Bretagne, RPI) with public `svn`, Trac, `git`, mailing lists, IRC, Plone, Skype... and use it for many projects
- But still...
  - Huge need of documentation (even if PIPS uses literate programming...)
  - Need of industrialization
  - Need further communication to increase community size

Current PIPS usage

- Automatic parallelization (Par4All C & Fortran to OpenMP)
- Distributed memory computing with OpenMP-to-MPI translation [STEP project]
- Generic vectorization for SIMD instructions (SSE, VMX, Neon...) (SAC project) [SCALOPES]
- Parallelization for embedded systems [SCALOPES]
- Compilation for hardware accelerators (Ter@PIX, SPoC, SIMD, FPGA...) [FREIA, SCALOPES]
- High-level hardware accelerators synthesis generation for FPGA [PHRASE, CoMap]
- Reverse engineering & decompiler (reconstruction from binary to C)
- Genetic algorithm-based optimization [Luxembourg university]
- Code instrumentation for performance measures
- GPU with CUDA & OpenCL [TransMedi@, FREIA, OpenGPU]

Current targets in development in Par4All

- Fortran and C to OpenMP
- Fortran and C to CUDA
- Fortran and C to OpenMP/Larrabee
- SIMD code generation (SSE, Neon, CUDA, OpenCL...)
- Compilation for various MP-SoC (ScaleoChip) and hardware accelerators (SCMP @ CEA) (close to STEP @ ITSP?)
- Aggregation of Simulink components [Aggregation]
- Scilab & Matlab compilation to OpenMP & GPU

What is in Par4All?

- Par4All Python and shell scripts for running, installation, development, validation...
- Documentation (automatically built from execution if possible: anti-literate programming ☺)
- Examples
- Runtime for GPU
- PIPS! ☺
- Polylib
- GCC 4.4 GFC for Fortran 95 parser

What is around Par4All?

- Debian and Ubuntu distribution with a repository, releases + nightly builds
- Work also on RedHat and should work on any Unix flavour (thanks to AutoTools! ☺)
- `par4all.org`
- `#pips` IRC channel
- Support mailing list `support@par4all.org`
- Validation infrastructure (PIPS + Par4All + Benchmarks on CPU, CPU...)
- Git infrastructure

`git` infrastructure (I)

- Use an efficient, robust, branch-enabled and versatile VCS for git people like us
- Nice WWW interface `https://git.hpc-project.com/cgit`
- Use HPC Project LDAP infrastructure for access rights, mailing lists...
- Public `git://git.hpc-project.com/git/par4all.git` for perfect tractability with many branches (since branches are a fundamental concept in `git`!)
  - `p4a` is the usable full infrastructure
  - `p4a-own` for what is really specific to Par4All.  This is where you should merge your work when you are happy with it. Not in `master` branch or   worse into `p4a`
  - `p4a-packages` assembles everything coming from outside, eventually modified for Par4All, from various branches
    - `p4a-linear`
    - `p4a-newgen`

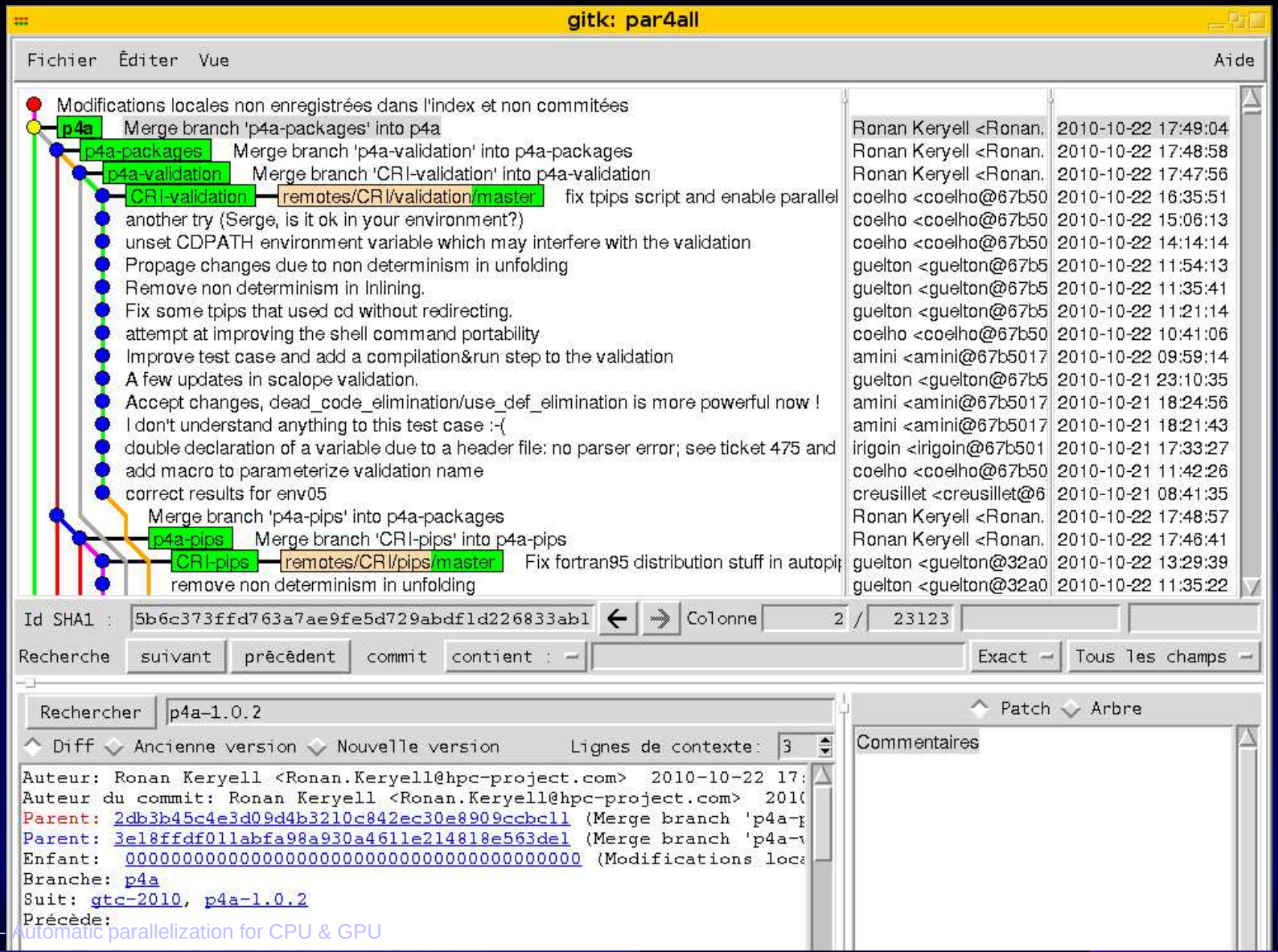
`git` infrastructure (II)

- `p4a-nlpmake`
- `p4a-pips`
- `p4a-validation`
- `p4a-polylib`
- `p4a-gcc-gfc`

- Some branches on the real external components, from other `git` or `git svn`: `CRI-linear`, `CRI-newgen`, `CRI-nlpmake`, `CRI-pips`, `CRI-validation`, `ICPS-polylib`, `p4a-gcc-gfc-4.4.3`
- Some tags for the releases
- `p4a_git` script deals with all these automatically ☺

- `git://git.hpc-project.com/git/par4all-priv.git`
  - Private to Par4All contributors (HPC Project, MINES, TB, interns...)
  - Par4All validation with benchmarks
- Other repositories for really secret stuff

`gitk` is your friend for Par4All ACID history



Outline

Basic GPU execution model

A sequential program on a host launches computational-intensive kernels on a GPU

- Allocate storage on the GPU
- Copy-in data from the host to the GPU
- Launch the kernel on the GPU
- The host waits...
- Copy-out the results from the GPU to the host
- Deallocate the storage on the GPU

Challenges in automatic GPU code generation

Mehdi AMINI's PhD thesis! ☺

- Find parallel kernels
- Improve data reuse inside kernels to have better compute intensity (even if the memory bandwidth is quite higher than on a CPU...)
- Access the memory in a GPU-friendly way (to coalesce memory accesses)
- Take advantage of complex memory hierarchy that make the GPU fast (shared memory, cached texture memory, registers...)
- Reduce the copy-in and copy-out transfers that pile up on the PCIe
- Reduce memory usage in the GPU (no swap there, yet...)
- Limit inter-block synchronizations
- Overlap computations and GPU-CPU transfers (via streams)

Par4All accel runtime (I)

- CUDA or OpenCL can not be directly represented in the internal representation (IR, abstract syntax tree) such as `__device__` or `<<< >>>`
- PIPS motto: keep the IR as simple as possible
- Use some calls to intrinsics functions that can be represented directly
- Intrinsics functions are implemented with (macro-)functions
  - `p4a_accel.h` has indeed currently 2 implementations
    - `p4a_accel-CUDA.h` than can be compiled with CUDA for nVidia GPU execution or emulation on CPU
    - `p4a_accel-OpenMP.h` that can be compiled with an OpenMP compiler for simulation on a (multicore) CPU

Par4All ≡ PyPS scripting in the backstage (I)

- Up to now PIPS was scripted with a special shell-like language: `tpips`
- PIPS was designed as an interactive tool...
- Not enough powerful (not a programming language)
- ⇝ A real language
- Avoid the “yet another language” syndrome...
- Add a SWIG Python interface to PIPS phases and interface
  - All the power of a wide-spread real language
  - Add introspection in the compiling phase
  - Easy to add any glue, pre-/post-processing to generate target code

⇝ `p4a` script as simple as

- `p4a -openmp toto.c -o toto`
- `p4a -cuda toto.c -o toto`

PyPS is the future of PIPS!

Par4All ≡ PyPS scripting in the backstage (II)

p4a sample code

```
1  def parallelize(self, fine = False, filter_select = None, filter_exclude = None):
       all_modules = self.filter_modules(filter_select, filter_exclude)
3
       # Try to privatize all the scalar variables in loops:
5      all_modules.privatize_module()

7      if fine:
           # Use a fine-grain parallelization à la Allen & Kennedy:
9          all_modules.internalize_parallel_code()
       else:
11         # Use a coarse-grain parallelization with regions:
           all_modules.coarse_grain_parallelization()
13
   def gpuify(self, filter_select = None, filter_exclude = None):
15     all_modules = self.filter_modules(filter_select, filter_exclude)

17     # First, only generate the launchers to work on them later. They are
       # generated by outlining all the parallel loops:
19     all_modules.gpu_ify(GPU_USE_WRAPPER = False,
                           GPU_USE_KERNEL = False,
21                         concurrent=True)

23     # Select kernel launchers by using the fact that all the generated
       # functions have their names beginning with "p4a_kernel_launcher":
25     kernel_launcher_filter_re = re.compile("p4a_kernel_launcher_.*[^!]$")
       kernel_launchers = self.workspace.filter(lambda m: kernel_launcher_filter_re.match(m.name))
27
       # Normalize all loops in kernels to suit hardware iteration spaces:
29     kernel_launchers.loop_normalize(
           # Loop normalize for the C language and GPU friendly
```

Par4All ≡ PyPS scripting in the backstage (III)

```
        LOOP_NORMALIZE_ONE_INCREMENT = True,
        # Arrays start at 0 in C, so the iteration loops:
        LOOP_NORMALIZE_LOWER_BOUND = 0,
        # It is legal in the following by construction (..Hmmm to verify)
        LOOP_NORMALIZE_SKIP_INDEX_SIDE_EFFECT = True,
        concurrent=True)

    # Unfortunately the information about parallelization and
    # privatization is lost by the current outliner, so rebuild
    # it... :-( But anyway, since we've normalized the code, we
    # changed it so it is to be parallelized again...
    #kernel_launchers.privatize_module()
    kernel_launchers.capply("privatize_module")
    #kernel_launchers.coarse_grain_parallelization()
    kernel_launchers.capply("coarse_grain_parallelization")

    # In CUDA there is a limitation on 2D grids of thread blocks, in
    # OpenCL there is a 3D limitation, so limit parallelism at 2D
    # top-level loops inside parallel loop nests:
    kernel_launchers.limit_nested_parallelism(NESTED_PARALLELISM_THRESHOLD = 2, concurrent=True)

    #kernel_launchers.localize_declaration()
    # Does not work:
    #kernel_launchers.omp_merge_pragma()


    # Add iteration space decorations and insert iteration clamping
    # into the launchers onto the outer parallel loop nests:
    kernel_launchers.gpu_loop_nest_annotate(concurrent=True)

    # End to generate the wrappers and kernel contents, but not the
    # launchers that have already been generated:
```

Par4All ≡ PyPS scripting in the backstage *(IV)*

```
63      kernel_launchers.gpu_ify(GPU_USE_LAUNCHER = False,
                                 concurrent=True)
65
        # Add communication around all the call site of the kernels:
67      kernel_launchers.kernel_load_store(concurrent=True)

69      # Select kernels by using the fact that all the generated kernels
        # have their names of this form:
71      kernel_filter_re = re.compile("p4a_kernel_\\d+$")
        kernels = self.workspace.filter(lambda m: kernel_filter_re.match(m.name))
73
        # Unfortunately CUDA 3.0 does not accept C99 array declarations
75      # with sizes also passed as parameters in kernels. So we degrade
        # the quality of the generated code by generating array
77      # declarations as pointers and by accessing them as
        # array[linearized expression]:
79      kernels.array_to_pointer(ARRAY_TO_POINTER_FLATTEN_ONLY = True,
                                 ARRAY_TO_POINTER_CONVERT_PARAMETERS = "POINTER")
```

Outline

Hyantes (I)

- Geographical application: library to compute neighbourhood population potential with scale control
- Example given in `par4all.org` distribution
- WildNode with 2 Intel Xeon X5670 @ 2.93GHz (12 cores) and a nVidia Tesla C2050 (Fermi), Linux/Ubuntu 10.04, gcc 4.4.3, CUDA 3.1
  - Sequential execution time on CPU: 30.355s
  - OpenMP parallel execution time on CPUs: 3.859s, speed-up: 7.87
  - CUDA parallel execution time on GPU: 0.441s, speed-up: 68.8
- With single precision on a HP EliteBook 8730w laptop (with an Intel Core2 Extreme Q9300 @ 2.53GHz (4 cores) and a nVidia GPU Quadro FX 3700M, 16 multiprocessors, 128 cores, architecture 1.1) with Linux/Debian/sid, gcc 4.4.3, CUDA 3.1:
  - Sequential execution time on CPU: 38s
  - OpenMP parallel execution time on CPUs: 18.9s, speed-up: 2.01
  - CUDA parallel execution time on GPU: 1.57s, speed-up: 24.2

Hyantes (II)

Original main C kernel:

```
1  void run(data_t xmin, data_t ymin, data_t xmax, data_t ymax, data_t step, data_t range,
2    town pt[rangex][rangey], town t[nb])
{
4      size_t i,j,k;

6      fprintf(stderr,"begin computation ...\n");

8      for(i=0;i<rangex;i++)
           for(j=0;j<rangey;j++) {
10             pt[i][j].latitude =(xmin+step*i)*180/M_PI;
               pt[i][j].longitude =(ymin+step*j)*180/M_PI;
12             pt[i][j].stock =0.;
               for(k=0;k<nb;k++) {
14                 data_t tmp = 6368.* acos(cos(xmin+step*i)*cos( t[k].latitude )
                       * cos((ymin+step*j)-t[k].longitude)
16                     + sin(xmin+step*i)*sin(t[k].latitude));
                   if( tmp < range )
18                     pt[i][j].stock += t[k].stock  / (1 + tmp) ;
               }
20         }
       fprintf(stderr,"end computation ...\n");
22 }
```

Hyantes (III)

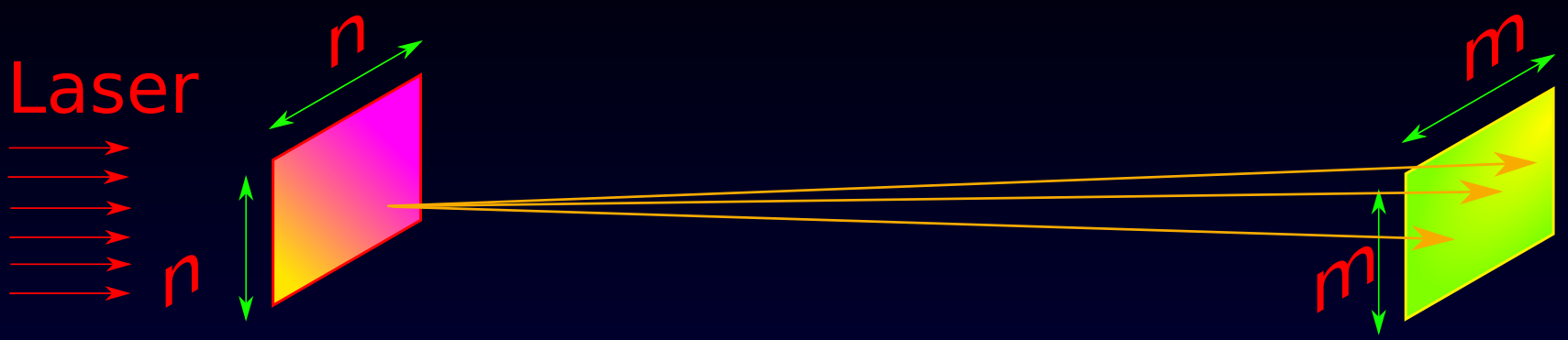
Generated GPU code:

```
1  void run(data_t xmin, data_t ymin, data_t xmax, data_t ymax, data_t step, data_t range,
2    town pt[290][299], town t[2878])
{
4     size_t i, j, k;
     //PIPS generated variable
6     town (*P_0)[2878] = (town (*)[2878]) 0, (*P_1)[290][299] = (town (*)[290][299]) 0;

8     fprintf(stderr, "begin computation ...\n");
     P4A_accel_malloc(&P_1, sizeof(town[290][299])-1+1);
10    P4A_accel_malloc(&P_0, sizeof(town[2878])-1+1);
     P4A_copy_to_accel(pt, *P_1, sizeof(town[290][299])-1+1);
12    P4A_copy_to_accel(t, *P_0, sizeof(town[2878])-1+1);

14    p4a_kernel_launcher_0(*P_1, range, step, *P_0, xmin, ymin);
     P4A_copy_from_accel(pt, *P_1, sizeof(town[290][299])-1+1);
16    P4A_accel_free(*P_1);
     P4A_accel_free(*P_0);
18    fprintf(stderr, "end computation ...\n");
}
20
void p4a_kernel_launcher_0(town pt[290][299], data_t range, data_t step, town t[2878],
22    data_t xmin, data_t ymin)
{
24    //PIPS generated variable
     size_t i, j, k;
26    P4A_call_accel_kernel_2d(p4a_kernel_wrapper_0, 290,299, i, j, pt, range,
                              step, t, xmin, ymin);
28 }

30 P4A_accel_kernel_wrapper void p4a_kernel_wrapper_0(size_t i, size_t j, town pt[290][299],
```

Hyantes (IV)

```
  data_t range, data_t step, town t[2878], data_t xmin, data_t ymin)
{
   // Index has been replaced by P4A_vp_0:
   i = P4A_vp_0;
   // Index has been replaced by P4A_vp_1:
   j = P4A_vp_1;
   // Loop nest P4A end
   p4a_kernel_0(i, j, &pt[0][0], range, step, &t[0], xmin, ymin);
}

P4A_accel_kernel void p4a_kernel_0(size_t i, size_t j, town *pt, data_t range,
  data_t step, town *t, data_t xmin, data_t ymin)
{
   //PIPS generated variable
   size_t k;
   // Loop nest P4A end
   if (i<=289&&j<=298) {
      pt[299*i+j].latitude = (xmin+step*i)*180/3.14159265358979323846;
      pt[299*i+j].longitude = (ymin+step*j)*180/3.14159265358979323846;
      pt[299*i+j].stock = 0.;
      for(k = 0; k <= 2877; k += 1) {
         data_t tmp = 6368.*acos(cos(xmin+step*i)*cos((*(t+k)).latitude)*cos(ymin+step*j
               -(*(t+k)).longitude)+sin(xmin+step*i)*sin((*(t+k)).latitude));
         if (tmp<range)
            pt[299*i+j].stock += t[k].stock/(1+tmp);
      }
   }
}
```

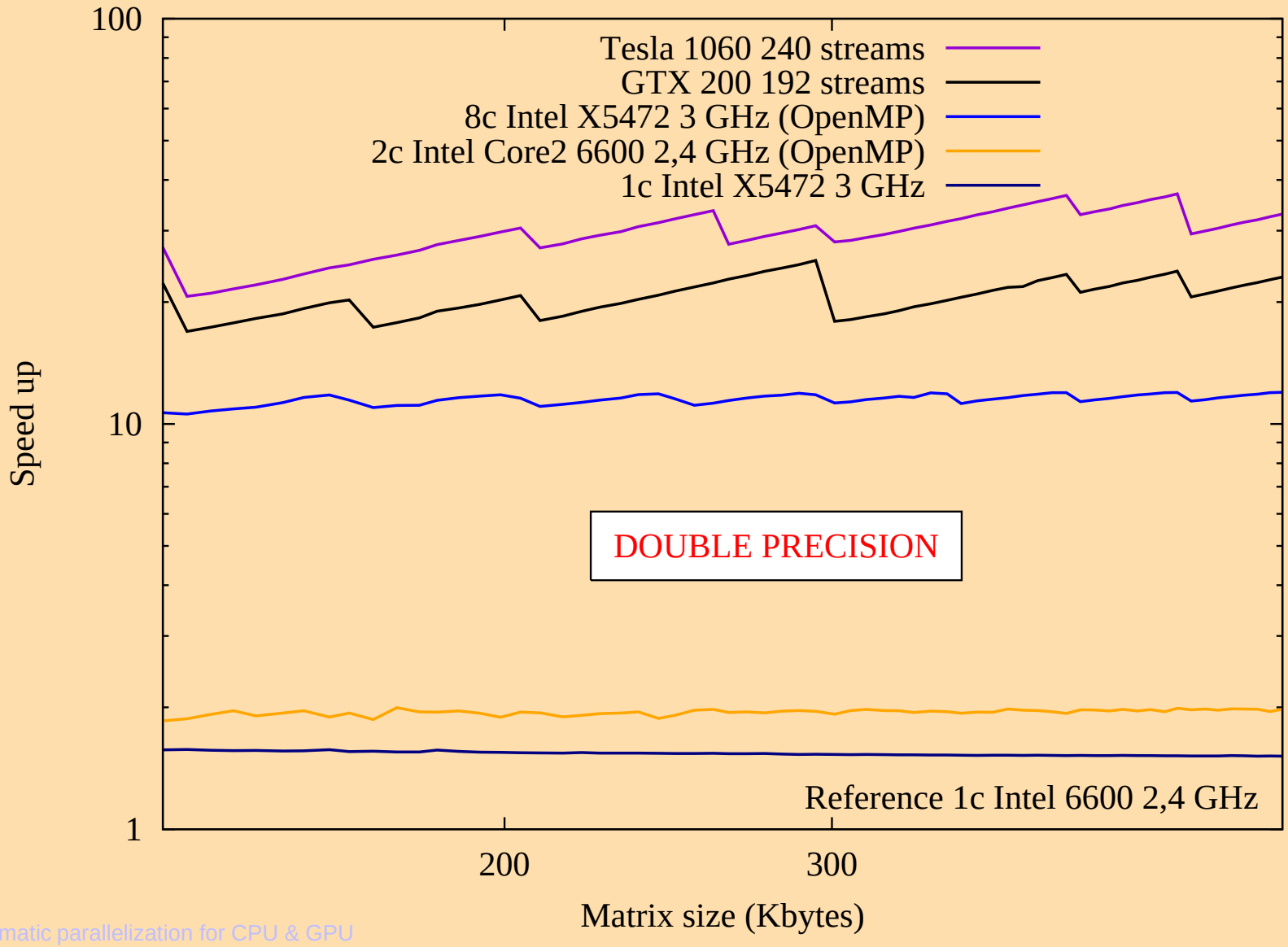
Results on a customer application



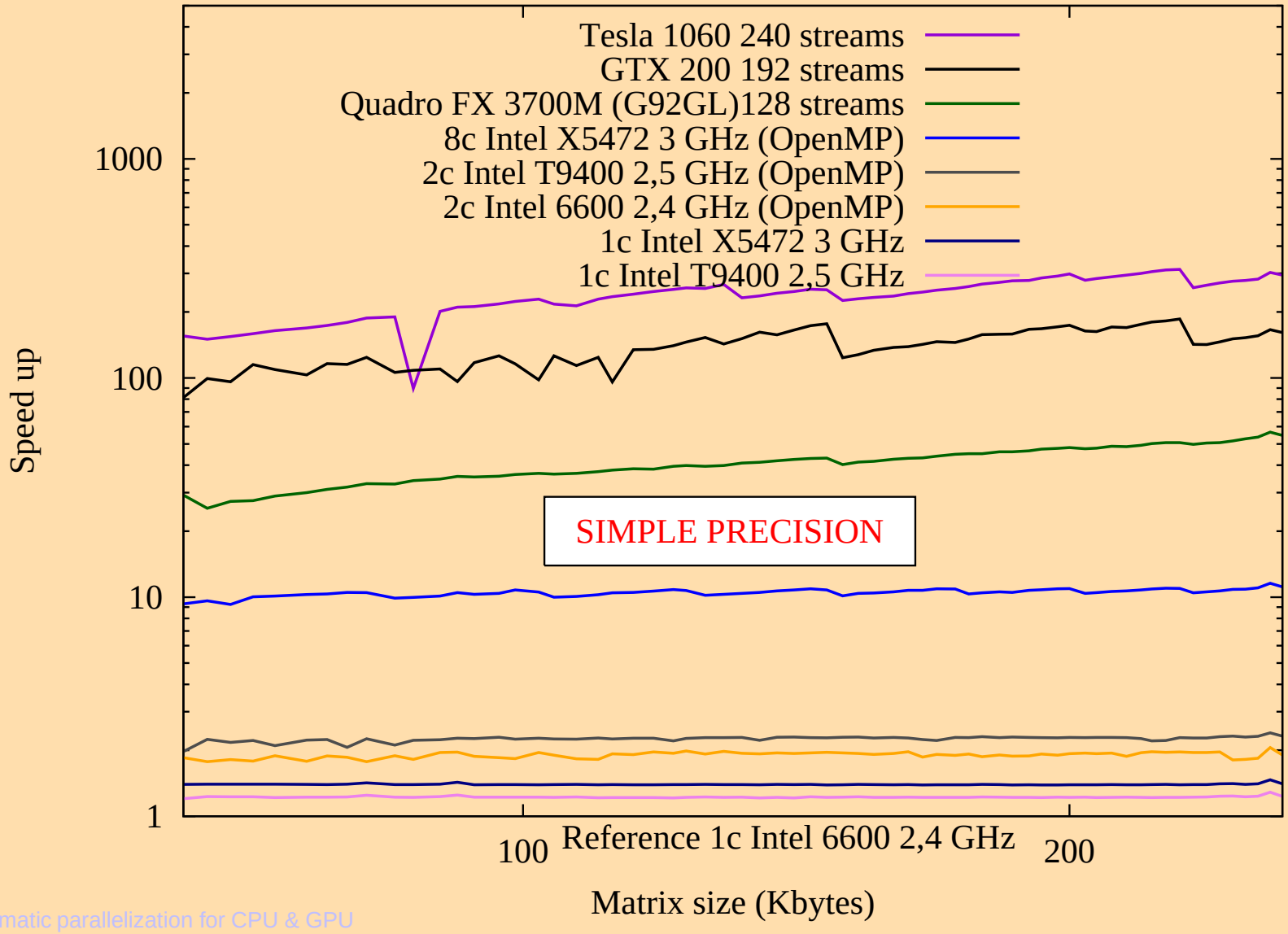
- Holotetrix's primary activities are the design, fabrication and commercialization of prototype diffractive optical elements (DOE) and micro-optics for diverse industrial applications such as LED illumination, laser beam shaping, wavefront analyzers, etc.
- Hologram verification with direct Fresnel simulation
- Program in C
- Parallelized with
  - Par4All CUDA and CUDA 2.3, Linux Ubuntu x86-64
  - Par4All OpenMP, gcc 4.3, Linux Ubuntu x86-64
- Reference: Intel Core2 6600 @ 2.40GHz

`http://www.holotetrix.com`

Comparative performance



Keep it simple (precision)



Outline

Scilab language

- Interpreted scientific language widely used like Matlab
- Free software
- Roots in free version of Matlab from the 80's
- Dynamic typing (scalars, vectors, (hyper)matrices, strings...)
- Many scientific functions, graphics...
- Double precision everywhere, even for loop indices (now)
- Slow because everything decided at runtime, garbage collecting
  - Implicit loops around each vector expression
    - Huge memory bandwidth used
    - Cache thrashing
    - Redundant control flow
- Strong commitment to develop Scilab through Scilab Enterprise, backed by a big user community, INRIA...
- HPC Project WildNode appliance with Scilab parallelization
- Reuse Par4All infrastructure to parallelize the code

Scilab2C (I)

- Try to recycle existing tools to avoid reinventing the wheel
  - Need a Scilab parser...
  - There are many many functions to understand and translate
- ∃ C generator! ☺, produced during European and other projects
- If you liked the ToolPack STF restructurer of Fortran programs written in... Fortran in the 1980's, you will love Scilab2C written in... Scilab! ☺
- Still a work in progress
  - Line-by-line translation
  - Use 1 temporary (vector, matrix...) variable per expression result allocated on the heap
  - Lot of scalar computation duplications
  - Many pointer arithmetics, linearization and casts because not using C99 niceties
- Use annotations in the entry code to specify some types and object sizes

Scilab2C (II)

- Difficult to imagine developing further advanced compilation phases in the current infrastructure that represents already a huge great work of Scilab code...
  - `sloccount scilab2c` shows these line numbers

ansic	61346	(51.08%)
fortran (BLAS & LAPACK)	48166	(40.11%)
sh	10081	(8.39%)
xml	496	(0.41%)

  - `find . -name '*.sc*' -print0 | xargs -0 wc` shows 121307 lines of Scilab code

  ⇝ Scilab2C is almost half of the size of PIPS...

☞ Try to work on the C output instead of reimplementing the Scilab side...

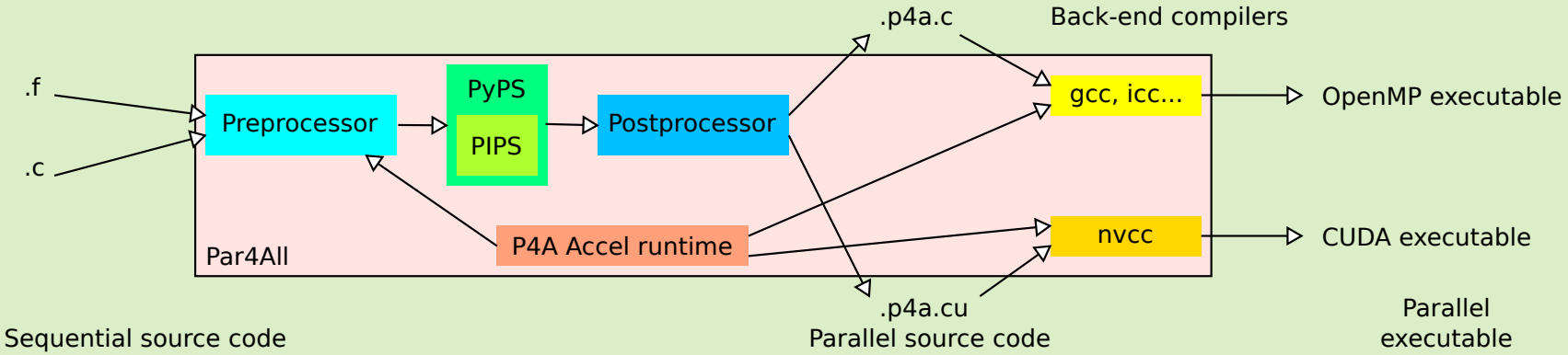
Small Scilab for GPU prototype *(I)*

- Rely on Scilab2C : may parallelize any Scilab code, not only rewritten library parts (Jacket, GPU implementation...) or special variable types
- Common subexpression elimination
- Inlining
- Code motion based on topological sort
- Privatization
- Scalarization
- Loop fusion to finish
- Good promising results since we start from interpreted language!
- But the C from Scilab2C is not good enough (but demo @ GTC2010!)
- Starting a new Scilab2C project...

Outline

Intestinal view of p4a

Overview



p4a architecture for advanced users

Main workflow in p4a.main from p4a.py

- Loading PyPS
- Normalization of user source file names
- Call to p4a_process to do the code transformation with PyPS
- Generate Cmake configuration files if asked
- Build the final executable with a p4a_builder

p4a extensions for bleeding edge users (I)

Code injection with:

```
p4a --execute p4a.execute_some_python_code_in_process='"import p4a_process_inject_class"'
```

of module:

```
# Import the code we want to tweak:
import p4a_process

class my_p4a_processor(p4a_process.p4a_processor):
    """Change the PyPS transit of p4a.

    Since it is done in the p4a_process.p4a_processor, we inherit of this
    class
    """
    def __init__(self, **args):
        "Change the init method just to warn about this new version"

        print "This is now done by class", self.__class__
        # Just call the normal constructors with all the named parameters:
        super(my_p4a_processor, self).__init__(**args)

    # Define a new method instead:
    def parallelize(self, **args):
        "Parallelize the code"

        # Apply for example a partial evaluation on all the functions of the
        # programm at the beginning of the parallelization
        self.workspace.all_functions.partial_eval()

        # Go on by calling the original method with the same parameters:
```

p4a extensions for bleeding edge users (II)

```
27          # Compatible super() invocation with pre-3 Python
            super(my_p4a_processor, self).parallelize(**args)

29  # Overide the old class definition by our new one:[language=Python]
    p4a_process.p4a_processor = my_p4a_processor
```

Outline

Future milestones

- SuperComputing 2010
  - Scilab ⇝ OpenMP & GPU prototype
  - Fortran & C ⇝ OpenMP & GPU
- 12/2010
  - Fortran 95
  - GPU
    - Simple Communication optimizations
    - Simple shared memory use case

Future work in Par4All & PIPS (I)

- PyPS: polishing Python scripting with SWIG (⇝ easy extension to other religions ☺)
- Mix heterogenous //: GPU and CPU execution, with SSE*x*, OpenMP and/or MPI...
- OpenCL output [OpenGPU project]
- Use PIPS SAC vectorization to generate OpenCL vector types
- If someone spent some time to write `#pragma` information... Use them! ☺
- Eclipse (EcliPIPS) integration to apply transformations and get easy feedback [OpenGPU project]
- Use advanced tiling to take advantage of various weird memory hierarchy: cache/shared/non-coherent/shared memory/texture cache/...

Future work in Par4All & PIPS *(II)*

- Select optimal transformations and code generation parameter by using genetic algorithm system
- Using complexity information to select the best execution mode (GPU or CPU)
- Matlab/SciLab/Octave/R/... to GPU compiler
- Generate run-time specialization and speculation when some static information is lacking at compilation time
- Target the Saint Cloud!

Conclusion (I)

- GPU (and other heterogeneous accelerators): impressive peak performances and memory bandwidth, power efficient
- Real codes are often not well written to be parallelized... even by human being ☹
- At least writing clean C99 & Fortran code should be a prerequisite
- Take a positive attitude... Parallelization is a good opportunity for deep cleaning (refactoring, modernization...) ⇝ improve also the original code
- Open standards to avoid sticking to some architectures
- Need software tools and environments that will last through business plans or companies
- Open implementations are a warranty for long time support for a technology (cf. current tendency in military and national security projects)

Conclusion (II)

- Python wrapping can really revive old software!
- Source-to-source compilation can be used in a wide range of domains
- Par4All allows to surf on the better back-ends with good results
- Many programmers use dynamic languages: think to them too!
-  Entry cost
-  Exit cost! ☹

Par4All is currently supported by...

- HPC Project
- Institut TÉLÉCOM/TÉLÉCOM Bretagne
- MINES ParisTech
- European ARTEMIS SCALOPES project
- European ARTEMIS SMECY project (to finance)
- French NSF (ANR) FREIA project
- French NSF (ANR) MediaGPU project
- French Images and Networks research cluster TransMedi@ project (finished)
- French System@TIC research cluster OpenGPU project
- French System@TIC research cluster SIMILAN project (to start)

Good opportunity for academic & industry high-tech collaboration!
And the revolution in research & teaching in some French Grandes Écoles... ☺