

Par4All: Auto-Parallelizing C and Fortran for the CUDA Architecture

Christopher D. Carothers⁴ Fabien Coelho² Béatrice
Creusillet¹ Laurent Daverio² Stéphanie Even³ Johan
Gall^{1,3} Serge Guelton^{2,3} François Irigoien² **Ronan**
Keryell^{1,3} Pierre Villalon¹

¹HPC Project

²Mines ParisTech/CRI

³Institut TÉLÉCOM/TÉLÉCOM Bretagne/HPCAS

⁴Rensselaer Polytechnic Institute/CS

10/1/2009

<http://www.par4all.org>



-



-

PII

- 

- ▶ 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...)

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
- Unreasonable to begin yet another new compiler project...
- Many academic Open Source projects are available...
- ...But customers need products ☺

Par4All

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

<http://www.par4all.org>



Outline

- 1 Par4All
- 2 CUDA generation
- 3 Results
- 4 Conclusion



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...

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





- Par4All initiative



Current PIPS usage

- Automatic parallelization (C & Fortran to OpenMP)
- Distributed memory computing with OpenMP-to-MPI translation (STEP project)
- Generic vectorization for SIMD instructions (SSE, VMX...) (SAC project)
- Parallelization for embedded systems (SCALOPES)
- Compilation for hardware accelerators (Ter@PIX, SPoC, SIMD, FPGA...)
- High-level hardware accelerators synthesis generation for FPGA
- Reverse engineering & decompiler (reconstruction from binary to C)

Logical next step

GPU! ☺



Current PIPS usage

- Automatic parallelization (C & Fortran to OpenMP)
- Distributed memory computing with OpenMP-to-MPI translation (STEP project)
- Generic vectorization for SIMD instructions (SSE, VMX...) (SAC project)
- Parallelization for embedded systems (SCALOPES)
- Compilation for hardware accelerators (Ter@PIX, SPoC, SIMD, FPGA...)
- High-level hardware accelerators synthesis generation for FPGA
- Reverse engineering & decompiler (reconstruction from binary to C)

Logical next step

GPU! ☺



Outline

- 1 Par4All
- 2 **CUDA generation**
- 3 Results
- 4 Conclusion



Basic CUDA 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
- Copy-out the results from the GPU to the host
- Deallocate the storage on the GPU



Challenges in automatic CUDA generation

- 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)



Automatic parallelization

Most fundamental for a parallel execution

Finding parallelism!

Several parallelization algorithms are available in PIPS

- For example classical Allen & Kennedy use loop distribution more vector-oriented than kernel-oriented  (or need later loop-fusion)
- Coarse grain parallelization based on the independence of array regions used by different loop iterations
 - ▶ Currently used because generates GPU-friendly coarse-grain parallelism
 - ▶ Accept complex control code without *if-conversion*



Parallel code $\rightsquigarrow$ Kernel code on GPU

- Need to extract parallel source code into kernel source code: outlining of parallel loop-nests
- Before:

```
1   for(i = 1; i <= 499; i++)  
    for(j = 1; j <= 499; j++) {  
3       save[i][j] = 0.25*(space[i - 1][j] + space[i + 1][j]  
                           + space[i][j - 1] + space[i][j + 1])  
5   }
```



- After:

```

1 p4a_kernel_launcher_0(space, save);
2 [...]
3 void p4a_kernel_launcher_0(float_t space[SIZE][SIZE],
4                             float_t save[SIZE][SIZE]) {
5     for(i = 1; i <= 499; i += 1)
6         for(j = 1; j <= 499; j += 1)
7             p4a_kernel_0(i, j, save, space);
8 }
9 [...]
10 void p4a_kernel_0(float_t space[SIZE][SIZE],
11                   float_t save[SIZE][SIZE],
12                   int i,
13                   int j) {
14     save[i][j] = 0.25*(space[i-1][j]+space[i+1][j]
15                        +space[i][j-1]+space[i][j+1]);
16 }

```



(1)

- PII**

(11)

```
for(i = 0; i <= n-1; i += 1)
    for(j = i; j <= n-1; j += 1)
        h_A[i][j] = 1;
```

1
2

can be decorated by PIPS with write array regions as:

```

1  // <h_A[PHI1][PHI2]-WEXACT-{0<=PHI1, PHI2+1<=n, PHI1<=PHI2}>
   for(i = 0; i <= n-1; i += 1)
3  // <h_A[PHI1][PHI2]-WEXACT-{PHI1==i, i<=PHI2, PHI2+1<=n, 0<=i}>
   for(j = i; j <= n-1; j += 1)
5  // <h_A[PHI1][PHI2]-WEXACT-{PHI1==i, PHI2==j, 0<=i, i<=j, 1+j<=n}>
   h_A[i][j] = 1;

```

- These read/write regions for a kernel are used to allocate with a `cudaMalloc()` in the host code the memory used inside a kernel and to deallocate it later with a `cudaFree()`



Conservative approach to generate communications

- Associate any GPU memory allocation with a copy-in to keep its value in sync with the host code
- Associate any GPU memory deallocation with a copy-out to keep the host code in sync with the updated values on the GPU
-  But a kernel could initialize an array, or use the initial values without writing into it, or use it as a local (private) array
- (PIPS does have many privatization phases)



```
1  cudaMemcpy(host_address, accel_address,
2             size, cudaMemcpyDeviceToHost)
```



Loop normalization

- Hardware accelerators use fixed iteration space (CUDA: thread index starting from 0...)
- Parallel loops: more general iteration space
- Loop normalization

Before

```

1  for(i = 1; i < SIZE - 1; i++)
2      for(j = 1; j < SIZE - 1; j++) {
3          save[i][j] = 0.25*(space[i - 1][j] + space[i + 1][j]
4                          + space[i][j - 1] + space[i][j + 1]);
5      }

```

After

```

1  for(i = 0; i < SIZE - 2; i++)
2      for(j = 0; j < SIZE - 2; j++) {
3          save[i+1][j+1] = 0.25*(space[i][j + 1] + space[i + 2][j + 1]
4                          + space[i + 1][j] + space[i + 1][j + 2]);
5      }

```

From preconditions to iteration clamping

(1)

- Parallel loop nests are compiled into a CUDA kernel wrapper launch
- The kernel wrapper itself gets its virtual processor index with `SOME blockIdx.x*blockDim.x + threadIdx.x`
- Since only full blocks of threads are executed, if the number of iterations in a given dimension is not a multiple of the `blockDim`, there are incomplete blocks
- An incomplete block means that some index overrun occurs if all the threads of the block are executed



From preconditions to iteration clamping

(II)

- So we need to generate code such as

```

1  void p4a_kernel_wrapper_0(int k, int l,...)
2  {
    k = blockIdx.x*blockDim.x + threadIdx.x;
4   l = blockIdx.y*blockDim.y + threadIdx.y;
    if (k >= 0 && k <= M - 1 && l >= 0 && l <= M - 1)
6     kernel(k, l, ...);
    }
    
```

But how to insert these guards?

- The good news is that PIPS owns *preconditions* that are predicates on integer variables. Preconditions at entry of the kernel are:

```

1  //  P(i, j, k, l) {0<=k, k<=63, 0<=l, l<=63}
    
```

- Guard $\equiv$ directly translation in C of preconditions on loop indices that are GPU thread indices



Complexity analysis

(1)

- Launching a GPU kernel is expensive
 - ▶ so we need to launch only kernels with a significant speed-up (launching overhead, memory CPU-GPU copy overhead...)
- Some systems use `#pragma` to give a go/no-go information to parallel execution

```
1  #pragma omp parallel if(size>100)
```

- $\exists$ phase in PIPS to symbolically estimate complexity of statements
- Based on preconditions
- Use a SuperSparc2 model from the '90s... ☺
- Can be changed, but precise enough to have a coarse go/no-go information
- To be refined: use memory usage complexity to have information about memory reuse (even a big kernel could be more efficient on a CPU if there is a good cache use)



▶

- ▶



Fortran to CUDA

(1)

- Fortran 77 parser available in PIPS
- CUDA is C/C++ with some restrictions on the GPU-executed parts
- Need a Fortran to C translator (f2c...)?
- Only one internal representation is used in PIPS
 - ▶ Use the Fortran parser
 - ▶ Use the C pretty-printer
- But the IO Fortran library is complex to use... and to translate
 - ▶ If you have IO instructions in a Fortran loop-nest, it is not parallelized anyway because of sequential side effects ☺
 - ▶ So keep the Fortran output everywhere but in the parallel CUDA kernels
 - ▶ Apply a memory access transposition phase $a(i, j) \rightsquigarrow a[j-1][i-1]$ inside the kernels to be pretty-printed as C
- We could output CUDA Fortran directly when available



- Fortran 90 and 95 support is quite interesting
 - ▶ Dynamic allocation
 - Avoid nasty user allocation functions in Fortran 77 used to allocate objects in a big array...
 -  Was a parallelism killer...
 - ▶ Derived types (C structures)
 - Avoid using more dimensions to arrays to simulate structure fields in Fortran 77
 -  Was a parallelism killer, used useless dereferencing...
 - ▶ Pointers
 - Useful to navigate through recursive derived type
 -  Was a parallelism killer, used useless dereferencing...
 -  Mmm... Try to avoid this anyway ☺
- Most of these concepts are in C, so are already dealt by PIPS
-  On-going Fortran 95 support with the `gfc` parser from GCC Gfortran



- 

(1)

- 

(11)

(III)



(IV)

49



Outline

- 1 Par4All
- 2 CUDA generation
- 3 Results
- 4 Conclusion

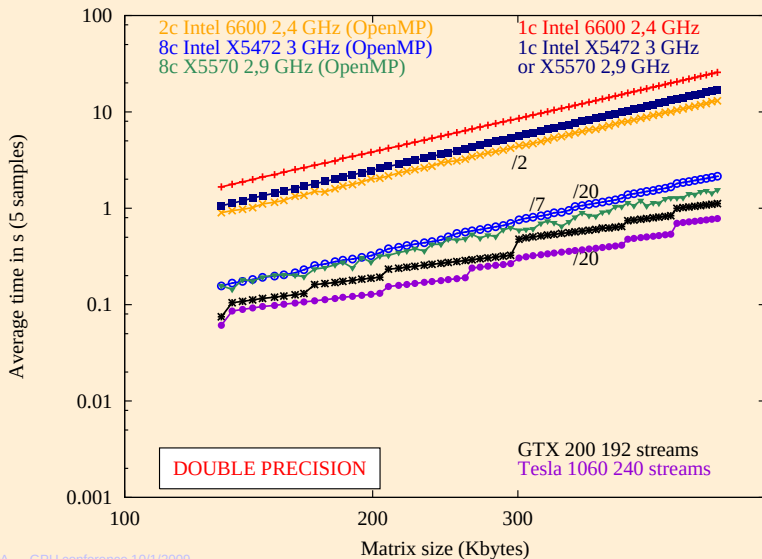


A diagram illustrating a laser beam passing through two rectangular apertures. The laser beam, represented by red arrows, enters from the left and passes through a rectangular aperture labeled n . The beam then passes through a second rectangular aperture labeled m . The beam is shown as a single line passing through both apertures, suggesting a waveguide or a beam splitter configuration. The apertures are labeled n and m in red text.

- 
- 

Comparative performance

(1)

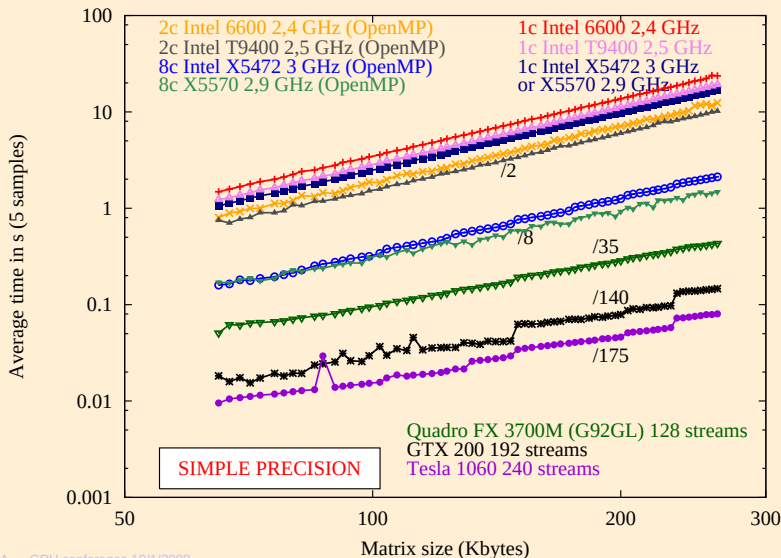


(11)



Keep it simple (precision)

(1)



(11)



Outline

- 1 Par4All
- 2 CUDA generation
- 3 Results
- 4 Conclusion



(1)

- 

▶

Why open source?

- Once upon a time, a research team in Luxembourg decided to experiment program transformations with genetic algorithms...
- Main idea:
 - ▶ Represent the transformations applied to a program as a gene
 - ▶ Make the genes to evolve
 - ▶ Measure the performance
 - ▶ Select and iterate
- They used PIPS to implement this
- Side effects
 - ▶ Now PIPS is scriptable in Python (PYPS)
 - ▶ Algorithm written in Python
 - ▶ We have an automated way to try many different optimizations to generate better CUDA kernels!

Stay tuned... 



Future work

(1)

- Use advanced tiling to take advantage of shared memory and texture cache
- Select optimal transformations and code generation parameter by using genetic algorithm system already in PIPS
- Using complexity information to select the best execution mode (GPU or CPU)
- Mix GPU and CPU execution, with SSEx, OpenMP and/or MPI
- Matlab to CUDA compiler
- OpenCL support via OpenGPU project
- Eclipse (EcliPIPS) integration via OpenGPU project
- If someone spent some time to write `#pragma` information... Use them! ☺
- Generate run-time specialization and speculation when some static information is lacking at compilation time



- 

Conclusion

(1)

- GPU (and other heterogeneous accelerators): impressive peak performances and memory bandwidth, power efficient
- Looks like to the 80's : many accelerators, vectors, parallel computers
 - ▶ Need to port the programs to new architectures
 - ▶ Many start-ups
- ...but some hypothesis changed, hardware constraints ~ we can no longer escape parallelism! No longer a niche market!
- Programming is *quite more* complex... ☹
- Non-functional specifications (speed, time, power, parallelism...) not taken into account by main-stream programming environment and teaching (high-level object programming...)
- And programs are quite bigger now! ☹
- Application source code evolves slowly compared to hardware



- 

- ▶ Open source architecture to collaborate with best research teams
- ▶ Use standards and Open Source for long-term investment
- ▶ Open source to avoid sticking to a one-company solution
- ▶ Hide manual work in preprocessing, tools, code generators...
- ▶ Can accept #pragma from other products as a second-source
- ▶ Come on and participate to the initiative!

You are not a compiler developer and you may ask what to keep from such a talk?

If you **understand better** how a compiler work, you can write code that **fits better** the compiler... 😊

Visit us at the Wild Systems booth #35 for a demo



Par4All is currently supported by...

- HPC Project
- Mines ParisTech
- Institut TÉLÉCOM/TÉLÉCOM Bretagne
- Rensselaer Polytechnic Institute
- European ARTEMIS SCALOPES project
- French NSF (ANR) FREIA project
- French Images and Networks research cluster TransMedi@ project
- French System@TIC research cluster OpenGPU project



4