Targeting heterogeneous architectures through macro data flow

M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Torquati

HPLGPU 2012 – Paris
Talk plot

- Scene 1: improvements
Talk plot

- Scene 1: improvements
- Scene 2: targeting parallel hardware
Talk plot

- Scene 1: improvements
- Scene 2: targeting parallel hardware
- Scene 3: CPU+GPU results
Scene 1: Improvements

Power

Time

Hardware

Programming frameworks
Scene 2: targeting parallel hardware

- Pthreads
- OpenMP
- CUDA
- OpenCL

Parallel design patterns
Algorithmic skeletons
Scene 3: GPU+CPU results

Performance

Hand MGT  Autonomic MGT
**Scene 1**

**Improvements**
Multicores

Currently available

- simplified core design
- shared memory hierarchy
- inter core control lines supporting small clusters (4x)
- cache coherency protocols

Easy design for

- multitasking (desktop)
- small scale parallel programs (shared memory)
- large scale parallel programs
  → via advanced network interconnection
Multicores

Intel i7
- full x86 microarchitecture (SSE up to 4.2)
- up to 6 cores per socket, 2 way hyperthreading, 4 socket interconnect → 48 thread per board

Oracle SPARC T3
- 16 SPARC cores (with FPU and crypto engine)
- 128 thread per CPU, 4 socket interconnect → 512 thread per board
Multicores

Tilera

- 64 or 100 cores per chip
- 2D mesh interconnect (hw routing unit)
- Network interfaces with direct cache injection
- (only) 4 memory interfaces to feed all cores
Programming model(s)

- run standard operating systems (e.g. Linux)
- run “classical” parallel programming frameworks
  - POSIX Threads
  - MPI
  - OpenMP
- seamless transition to multicores
  - additional problems related to memory allocation
    e.g. `numactl --interleave=all`
Multicores

IBM PowerEN

- 16 cores (Power ISA, 4 way SMT, in order 2 way concurrent issue)
- 4 special purpose coprocessors (xml, regex, cypher, compress)
- high speed network interfaces

Programming model(s)

- classical ones
- integrated with libraries handling dedicated co-processors
GPUs

Started as graphic coprocessors $\rightarrow$ GP-GPU

- \{control unit with a number of attached exec units (ALU)\}*
- highly efficient memory design
  $\rightarrow$ striped concurrent access, high bandwidth
- only suitable for data parallel code
  $\rightarrow$ possibly with no data dependencies
- coprocessors $\rightarrow$ explicit data management required
- slightly different code may significantly boost performance
- currently up to 960 cores, mem 406.5 Gb/Sec, 515 double precision GFlops
GPU programming model

Whatever CUDA, OpenCL, Brook+, ...

▶ explicit control of data traffic to/from GPU
▶ explicit control of memory allocation on GPU
▶ explicit control of “kernel” scheduling on GPU
▶ explicit control of multiple GPUs per PE

More recently

▶ directives working much the same as OpenMP directives
▶ programmers relieved of most of the burden relative to data movement, scheduling, etc.
Current programming model issues

- closer to the bare metal than programmers
- deep knowledge of the target architecture needed
- cumbersome, error prone programming effort
- lack of “minimal disruption”
  → poor abstraction level presented to the final user (application programmer)
Scene 2
Targeting parallel HW
Algorithmic skeletons known, reusable, parametric parallelism exploitation patterns provide to the application programmers as pre-defined, composable entities
Alternative programming models

**Algorithmic skeletons** known, reusable, parametric parallelism exploitation patterns provide to the application programmers as pre-defined, composable entities

**Parallel design patterns** Parallel incarnation of the design pattern “recipe” concept → Software engineering backup concepts for algorithmic skeletons
Algorithmic skeletons

- Cole 1988
  - Algorithmic skeletons → common, parametric, reusable parallelism exploitation pattern
  - Directly exposed as constructs, library calls, objects, higher order functions, components, ...
  - Composable
  - Two tier model → Stream parallel skeletons with inner data parallel skeletons
- High level parallel abstractions (HPC community)
  - Hiding most of the technicalities related to parallelism exploitation
  - Directly exposed to application programmers
Typical algorithmic skeletons

Stream parallel
  ▶ pipeline (computation in stages)
  ▶ farm (embarrassingly parallel)

Data parallel
  ▶ map (embarrassingly parallel)
  ▶ stencil (with dependencies)
  ▶ reduce (binary, associative and commutative operators)
  ▶ scan (parallel prefix)

Control parallel
  ▶ loops (determinate, indeterminate)
  ▶ sequential (wrapping of existing code)
  ▶ seqcomposition (in place pipelines)
Skeleton applications

- sequential “function” code $s_1$, $s_2$, $s_3$ provided by application programmer
- along with proper syntax to express the tree
- mapping, scheduling, communication, synchronization, all in charge of the skeleton framework
Key strengths

**Full parallel structure of the application exposed to the skeleton framework**
- optimizations exploit structure knowledge
- support for autonomic non functional concern management

**Framework responsibility for architecture targeting**
- write once, executed everywhere code
- with architecture specific compiler back end tools

**Functional debugging (only) in charge to the application programmer**
- possibility to run skeleton programs through sequential back end
Algorithmic skeleton (assessments)

Separation of concerns
- application programmers → what has to be computed (algorithm)
- system (skeleton) programmers → how things are efficiently computed

Inversion of control
- programmers suggest a possible implementation
- skeleton framework applies known optimizations

Performance
- same as hand written parallel code
- at a fraction of the development time
Parallel design patterns

Software engineering community

▶ introduce concept in early ’00
  Massingill, Mattson, Sanders *Patterns for parallel programming* 2006
▶ parallel “branch” of traditional (seq) design patterns
▶ as defined in the “Gamma book”

Separate communities

▶ algorithmic skeleton results ignored
▶ despite
  ▶ skeletons ≡ pre-programmed *incarnations*
    of a parallel design patterns
Parallel design pattern split in 4 *spaces*

1. **Finding concurrency space** → modelling concurrent (i.e. potentially parallel) activities
2. **Algorithm space** → modelling implementation of parallel algorithms
3. **Supporting structure space** → modelling suitable ways to implement different parallel algorithms
4. **Implementation mechanism space** → *de facto* targeting different architectures
Design pattern space structure

Finding concurrency design space

Algorithm design space

Supporting structure design space

Impl. mechanisms design space

Collapsed in the implementation of algorithmic skeletons
- application programmer → concurrency and algorithm spaces
- skeleton implementation (system programmer) → support structures and implementation mechanisms

Decomposition (task, data), Dependency analysis (group tasks, order tasks, data sharing), Design evaluation

Organize by task (task parallelism, divide & conquer), Organize by data decomposition (geometric decomp, recursive data), Organize by flow of data (pipeline, event based coordination)

Program structure (SPMD, Master/Worker, Loop parallelism, Fork/Join), Data structures (shared data, shared queue, distrib. array)

UE management, Synchronization, Communication
Structured parallel programmer: design patterns
Structured parallel programmer: skeletons

Skeleton library

Problem

Parallel programmer

high level source code

Tools (advanced)

Application code
Structured parallel programmer

- Design patterns
- Skeleton library

Parallel programmer

- Use knowledge
- Instantiate

Problem
Source code
One step further

Currently

- skeletons: parallelism exploitation patterns
- templates: efficient implementation of skeletons on target architectures
- compiler:
  - visits skeleton tree
  - instantiate templates
  - glues together templates

Hw Targeting through

- template optimization +
- compiler algorithms/heuristics
Next step: Intermediate virtual machine

Macro data flow (MDF)

- data flow with “fat” instructions
- instruction ::= HLL function
- same firing rules as in standard data flow

Hw targeting

- compiler \(\rightarrow\) macro data flow graphs
- optimized, distributed macro data flow interpreters
Skeletons to MDF

Pipeline

- linear chain of MDF instructions
- data flow dependencies among stages

\[ f_1 \mid x \mid x_1 \]
\[ f_2 \mid x_1 \mid x_2 \]
\[ f_3 \mid x_2 \mid y \]
Skeletons to MDF

Map - Farm - Reduce

- items from collection/stream or partitions from collection scheduled to worker MDF instructions
- parallel computation results gathered and re-build by another MDF instruction

\[\text{decomp} \mid x \mid y_1 \ldots y_n\]

\[\text{f} \mid y_1 \mid x_1 \quad \ldots \quad \text{f} \mid y_n \mid x_n\]

\[\text{recomp} \mid x_1 \ldots x_n \mid y\]

\[\oplus \mid p_1 \mid x_1 \quad \ldots \quad \oplus \mid p_n \mid x_n\]

\[\oplus \mid x_1 \ldots x_n \mid y\]
Macro Data Flow

- Skeleton source code
- Compiler
- MDF graph
- Output manager
- Task pool
- Input manager
- Interp instance
- Interp instance
- Interp instance
Task pool management

Task pool manager

- waits for requests/results from interpreter instances
  → schedules fireable instructions for execution
  → updates fireable MDF instruction list
    (token updated by interpreter instances)
- logically centralized, physically distributed (if needed, e.g. logical tree of TP managers, interp instances at the leaves)

Scheduling

- pure “on demand”
- affinity may be used in case of multiple fireable instructions
Targeting multicore hardware

“standard” Shared Memory multicores

- Intel, AMD, ...
- comparable performance w.r.t. OpenMP with regular computations
- much better with irregular computations
Sample performance

![Graph showing performance metrics](image-url)
Sample performance

![Graph showing performance and speedup with different parallelism degrees for Generic Graph, Pipeline Graph, and Map Graph.](image)

- Completion time (secs)
- Speed up
- Parallelism degree
- Generic Graph
- Pipeline Graph
- Map Graph
MDF from “well formed code”

Well formed code

- possibly nested loops
- with sequences of calls to math routines (e.g. LAPACK routines)
  → with associated input/output parameter semantics
HLL → MDF graphs

Compile

\[ A[k][k] := DSYRK(A[k][n], A[k][k]) \]

- function \(\rightarrow\) DSYRK
- input tokens \(\rightarrow\) \(A[k][n], A[k][k]\)
- output tokens \(\rightarrow\) \(A[k][k]\)
MDF from “well formed code”

Cholesky factorization

FOR $k = 0..TILES-1$
  FOR $n = 0..k-1$
    $A[k][k] := DSYRK(A[k][n], A[k][k])$
    $A[k][k] := DPOTRF(A[k][k])$
  FOR $m = k+1..TILES-1$
    FOR $n = 0..k-1$
      $A[m][k] :=$
        $DGEMM(A[k][n], A[m][n], A[m][k])$
      $A[m][k] := DTRSM(A[k][k], A[m][k])$
Cholesky factorization compiled

$3 \times 3$ tile matrix

Diagram:

```
DPOTRF
  └── DTRSM
      └── DSYRK
          └── DPOTRF

DPOTRF
  └── DTRSM
      └── DSyrK
          └── DPOTRF

DPOTRF
  └── DTRSM
      └── DGEMM
          └── DSyrK
              └── DPOTRF
```
Static optimizations

Compile time

- recognize multiple MDF instructions with same function
- group to augment the grain
Dynamic optimizations

Affinity scheduling

- large data structures passed by pointer (as usual)
  - cached locally
  - copied on cache miss
- MDF instruction reusing large portions of data
  → scheduled to the same remote interpreter
Target driven optimizations

On target $PE_i$

- apply proper compiling to MDF instruction code
  e.g. SSE/ AVX
  OpenMP
  OpenCL
- Source to source transformation (refactoring) possible because:
  - full parallel pattern structure exposed
  - full access to the MDF instruction source code
Scene 3
CPU + GPU results
MDF interpreter upgraded

TP management
- thread(s) added to take care of GPU(s)
- concur with core feeding threads
- (uniform) on demand task scheduling

Static/dynamic optimization
- collective data parallel subgraphs intercepted
- *on-the-fly* processed as single MDF instructions
- directed to the GPU
- after suitable re-factoring
MDF interpreter upgraded

Source (skeleton) -> Compiler -> Source (HLNA)

Inputs -> MDF graph

Input manager

Task pool

Parallel MDF interpreter

\[ mdfi_1 \]  
\[ core_1 \]

\[ mdfi_2 \]  
\[ core_2 \]

...  
...  

\[ mdfi_{i-1} \]  
\[ core_{i-1} \]

\[ mdfi_i \]  
\[ gpu_1 \]

...  
...  

\[ mdfi_n \]  
\[ gpu_k \]
Preliminary experiments

Hardware

- 2x AMD 6176 (24 cores) with
- nVidia Tesla C2050 (490 cores)

Strategy

- self scheduling of fireable MDF instructions
- grouping map workers (lower level)
Copy optimization

- from HLL code $\rightarrow$ data reused $A$
- copy $A$ to GPU memory once
- use pointers instead of per MDF instruction data transfers
## Load balancing

<table>
<thead>
<tr>
<th>#cores</th>
<th>Avg task/core</th>
<th>GPU tasks</th>
<th>GPU %</th>
<th>Avg task/core</th>
<th>GPU tasks</th>
<th>GPU %</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>53</td>
<td>459</td>
<td>89%</td>
<td>43</td>
<td>469</td>
<td>91%</td>
</tr>
<tr>
<td>2</td>
<td>53</td>
<td>406</td>
<td>79%</td>
<td>37</td>
<td>438</td>
<td>85%</td>
</tr>
<tr>
<td>4</td>
<td>49</td>
<td>317</td>
<td>61%</td>
<td>34</td>
<td>377</td>
<td>73%</td>
</tr>
<tr>
<td>8</td>
<td>40</td>
<td>193</td>
<td>37%</td>
<td>30</td>
<td>272</td>
<td>53%</td>
</tr>
<tr>
<td>16</td>
<td>30</td>
<td>42</td>
<td>8%</td>
<td>23</td>
<td>141</td>
<td>27%</td>
</tr>
</tbody>
</table>
Performance improvement

![Graph showing performance improvement with varying number of cores. The graph plots % diff Tc against #cores.]
Conclusions (ParaPhrase perspective)

- Preliminary experiments
  - demonstrate feasibility
  - demonstrate performance
- Methodology designed
  - separate general purpose optimizations
    - typical of parallel patterns
    - to be implemented at compile time
  - from target specific optimizations
    - typical of target architecture
    - to be implemented at run time
    - exploiting re-factoring
THANK YOU

marcod@di.unipi.it

http://www.di.unipi.it/~marcod
http://www.paraphrase-ict.eu