Programming heterogeneous, accelerator-based multicore machines: a runtime’s perspective

Raymond Namyst
University of Bordeaux
RUNTIME INRIA group

FGPS 2012
Mons, Belgium, Nov 9th
Understanding the evolution of parallel machines

The end of Moore’s law?

• The end of single thread performance increase
  – Clock rate is no longer increasing
  – Thermal dissipation

• Processor architecture is already very sophisticated
  – Prediction and Prefetching techniques achieve a very high percentage of success
  – Actually, processor complexity is decreasing!

• Question: What kind of circuits should we better add on a chip?
Understanding the evolution of parallel machines

Welcome to the multicore era

- **Answer:** Multicore chips
  - Several cores instead of one processor

- **Back to complex memory hierarchies**
  - Shared caches
    - Organization is vendor-dependent
  - NUMA penalties

- **Clusters can no longer be considered as “flat sets of processors”**
Understanding the evolution of parallel machines

Multicore is a solid trend

- More performance = more cores
  - Toward embarrassingly parallel machines?

- Designing scalable multicore architectures
  - 3D stacked memory
  - Non-coherent cache architectures
    - Intel SCC
    - IBM Cell/BE
Then came heterogeneous computing

And it seems to be a solid trend…

• Accelerators
  – Nvidia & AMD GPUs
    • De facto adoption
    • Concrete success stories
      Speedups > 50
  – Intel MIC
  – Imgtec PowerVR

• Accelerators get more integrated
  – Intel Ivy Bridge
  – AMD APUs

• They all raise the same issues
  – Cost of moving data
    • Even for so-called APUs
  – Programming model
Then came heterogeneous computing

And it seems to be a solid trend…

- “Future processors will be a mix of general purpose and specialized cores” (anonymous source)
  - One interpretation of “Amdalh’s law”
    - Need powerful, general purpose cores to speed up sequential code

- Are we happy with that?
  - No, but it’s probably unavoidable!

Mixed Large and Small Core
What Programming Models for such machines?

Widely used, standard programming models

- **MPI**
  - Communication Interface
  - Scalable implementations exist already
    - Was actually designed with scalability in mind
    - Makes programmers “think” scalable algorithms
  - NUMA awareness?
  - Memory consumption

- **OpenMP**
  - Directive-based, incremental parallelization
  - Shared-memory model
    - Well suited to symmetric machines
  - Portability wrt #cores
  - NUMA awareness?
OpenMP (1997)

A portable approach to shared-memory programming

- Extensions to existing languages
  - C, C++, Fortran
  - Set of programming directives

- Fork/join approach
  - Parallel sections

- Well suited to data-parallel programs
  - Parallel loops

- OpenMP 3.0 introduced tasks
  - Support for irregular parallelism

```c
int matrix[MAX][MAX];
...
#pragma omp parallel for
for (int i; i < 400; i++)
{
    matrix[i][0] += ...
}
```

0 (main)

fork

join
OpenMP (1997)

Multithreading over shared-memory machines
The Quest for Programming Models

Dealing with multicore machines

- Several efforts aim at making MPI and OpenMP multicore-ready
  - MPI
    - NUMA-aware buffer management
    - Efficient collective operations
  - OpenMP
    - Scheduling in a NUMA context (memory affinity, work stealing)
    - Memory management (page migration)
The Quest for Programming Models

Dealing with accelerators

- Software Development Kits and Hardware Specific Languages
  - “Stay close to the hardware and get good performance”
    - Low-level abstractions
  - Compilers generate code for accelerator device

- Examples
  - Nvidia’s CUDA
    - *Compute Unified Device Architecture*
  - IBM Cell SDK
  - OpenCL

Accelerators

Cell SDK
CUDA
OpenCL
The Quest for Programming Models

Are we forced to use such low-level tools?

- Fortunately, well-known kernels are available
  - BLAS routines
    - e.g. CUBLAS
  - FFT kernels

- Implementations are continuously enhanced
  - High Efficiency

- Limitations
  - Data must generally fit accelerators memory
  - Multi-GPU configurations not well supported

- Ongoing efforts
  - Using multi-GPU + multicore
    - PLASMA/MAGMA (ICL/UTK)
Directive-based approaches

Offloading tasks to accelerators

- Idea: use simple directives… and better compilers
  - HMPP (Caps Enterprise), OpenMPC (Purdue University)
  - GPU SuperScalar (Barcelona Supercomputing Center)
  - OpenACC

```c
#pragma omp task inout(C[BS][BS])
void matmul(float *A, float *B, float *C) {
    // regular implementation
}
#pragma omp target device(cuda) implements(matmul)
void matmul cuda (float *A, float *B, float *C) {
    // optimized kernel for cuda
}
```
The Quest for Programming Models

How shall we program clusters of hybrid machines?

- Until the long-awaited new programming model shows up, we must go the hard hybrid way
  - Combine different paradigms
    - MPI/PGAS + OpenMP/TBB + CUDA/OpenCL
  - Performance portability is hard to achieve
    - Work distribution depends on #GPU and #CPU per node…
    - Needs aggressive tuning

- Runtime systems can help
The Quest for Programming Models

How shall we program heterogeneous clusters?

- The uniform way
  - Use a single (or a combination of) high-level programming language to deal with network + multicore + accelerators
  - Increasing number of directive-based languages
    - Use simple directives… and good compilers!
      - XcalableMP
      - PGAS approach
      - HMPP, OpenMPC, OpenMP 4.0
        - Generate CUDA from OpenMP code
      - StarSs
  - Much better potential for *composability*…
    - Runtime systems can help!
The role of runtime systems

Toward “portability of performance”

• Do dynamically what can’t be done statically
  – Load balance
  – React to hardware feedback
  – Autotuning, self-organization

• We need to put more intelligence into the runtime!
Overview of StarPU

A runtime system for heterogeneous architectures

- Rational
  - Dynamically schedule tasks on all processing units
    - See a pool of heterogeneous processing units
  - Avoid unnecessary data transfers between accelerators
    - Software VSM for heterogeneous machines

\[ A = A + B \]
Overview of StarPU

Maximizing PU occupancy, minimizing data transfers

- Accept tasks that may have multiple implementations
  - Together with potential inter-dependencies
    - Leads to a dynamic acyclic graph of tasks
  - Data-flow approach
  - Scheduling hints

- Open, general purpose scheduling platform
  - Scheduling policies = plugins

HPC Applications
Parallel Compilers
Parallel Libraries

StarPU

Drivers (CUDA, OpenCL)

CPU
GPU
MIC

CUDA, OpenCL

(A_{RW}, B_R, C_R)
Tasks scheduling

- When a task is submitted, it first goes into a pool of “frozen tasks” until all dependencies are met
- Then, the task is “pushed” to the scheduler
- Idle processing units actively poll for work (“pop”)
- What happens inside the scheduler is… up to you!
Tasks scheduling

• Queue based scheduler
  - Each worker « pops » task in a specific queue

• Implementing a strategy
  - Easy!
  - Select queue topology
  - Implement « pop » and « push »
    • Priority tasks
    • Work stealing
    • Performance models, …

• Scheduling algorithms testbed
Tasks scheduling

- Queue based scheduler
  - Each worker « pops » task in a specific queue

- Implementing a strategy
  - Easy!
  - Select queue topology
  - Implement « pop » and « push »
    - Priority tasks
    - Work stealing
    - Performance models, …

- Scheduling algorithms testbed
Dealing with heterogeneous architectures

- Task completion time estimation
  - History-based
  - User-defined cost function
  - Parametric cost model

- Can be used to improve scheduling
  - E.g. Heterogeneous Earliest Finish Time
Dealing with heterogeneous architectures

- Data transfer time estimation
  - Sampling based on off-line calibration
- Can be used to
  - Better estimate overall execution time
  - Minimize data movements

Performance prediction
Dealing with heterogeneous architectures

- On the influence of the scheduling policy
  - LU decomposition
    - 8 CPUs (Nehalem) + 3 GPUs (FX5800)
    - 80% of work goes on GPUs, 20% on CPUs

- StarPU exhibits good scalability
  \textit{wrt}:
  - Problem size
  - Number of GPUs
Mixing PLASMA and MAGMA with StarPU
With University of Tennessee & INRIA HiePACS

- Cholesky decomposition
  - 5 CPUs (Nehalem) + 3 GPUs (FX5800)
  - Efficiency > 100%
Mixing PLASMA and MAGMA with StarPU

With University of Tennessee & INRIA HiePACS

- QR decomposition
  - 16 CPUs (AMD) + 4 GPUs (C1060)

![Graph showing performance comparison between different configurations of CPUs and GPUs.](image)

- 4 GPUs + 16 CPUs
- 4 GPUs + 4 CPUs
- 3 GPUs + 3 CPUs
- 2 GPUs + 2 CPUs
- 1 GPUs + 1 CPUs

+12 CPUs
~200 GFlops

(although 12 CPUs alone ~150 Gflops)
Mixing PLASMA and MAGMA with StarPU

« Super-Linear » efficiency in QR?

- Kernel efficiency
  - sgeqrt
    - CPU: 9 Gflops
    - GPU: 30 Gflops
    - Ratio: x3
  - stsqrt
    - CPU: 12 Gflops
    - GPU: 37 Gflops
    - Ratio: x3
  - somqr
    - CPU: 8.5 Gflops
    - GPU: 227 Gflops
    - Ratio: x27
  - Sssmqr
    - CPU: 10 Gflops
    - GPU: 285 Gflops
    - Ratio: x28

- Task distribution observed on StarPU
  - sgeqrt: 20% of tasks on GPUs
  - Sssmqr: 92.5% of tasks on GPUs

- Heterogeneous architectures are cool! 😊
Scheduling for Power Consumption

- Power consumption information can be retrieved from OpenCL devices
  - `clGetEventProfilingInfo`
    - `CL_PROFILING_POWER_CONSUMED`
  - Implemented by Movidius multicore accelerator

- Autotuning of power models similar to performance

- Scheduling heuristic inspired by MCT
  - Actually, a subtle mix with MCT is possible
Ongoing work

• Target for the HMPP compiler, XMP framework (ANR-JST FP3C)
  – Introduce dynamic scheduling in directive-based languages compilers
  – Early experiments with the StarSs framework

• A quest for “ideal” granularity
  – Enabling adaptive granularity
    • Divisible tasks
    • Autotuning

• Porting on Intel MIC
Usability in real applications

SOCL: A StarPU-enabled OpenCL implementation

• Run legacy OpenCL codes on top of StarPU
  – StarPU scheduler involved only if a slight modification is made:
    • OpenCL queue attached to a context, not to a particular device
  – Incremental introduction of “dynamic scheduling” within OpenCL applications
  – Data prefetching also possible
Usability in real applications (cont’d)

I mean real applications…

• Integration into existing MPI programs
  – Because data is spread among multiple memory banks, we have to slightly change the way we designate data to be sent
  • Do not necessarily force data to be flushed back to host memory

• All the magic done by the scheduler (scheduling, prefetching, flushing) makes it harder to understand (bad) performance
  – Where does this disappointing behavior come from?
From nested parallelism to parallel code reuse

Composing parallel libraries

• Integration with other shared-memory programming paradigms
  – We cannot *taskify* the whole world
  – At least provide support for parallel tasks
    • Implemented with threads/OpenMP/…
    • Nested parallelism
  – It raises the composability problem

• It’s all about composability!
  – Probably the biggest upcoming challenge for runtime systems
    • Hybridization will mostly be indirect (linking libraries)

• And with composability come a lot of related issues
  – Need for autotuning / scheduling hints
How will we program exascale machines?

Let’s prepare for serious changes

- Billions of threads will be necessary to occupy exascale machines
  - Exploit every source of (fine-grain) parallelism
    - Not every algorithm/problem resolution can scale that far 😞
  - Multi-scale, Multi-physics applications are welcome!
    - Great opportunity to exploit multiple levels of parallelism

- Is SIMD the only reasonable approach?
  - Are CUDA & OpenCL our future?

- No global, consistent view of node’s state
  - Local algorithms
  - Hierarchical coordination/load balance

- Maybe, this time, we should seriously consider enabling (parallel) code reuse…
Scheduling contexts

- Each context features its own scheduler
- Multiple parallel libraries can run simultaneously
  - Virtualization of resources
  - Scalability workaround
- Contexts may share processing units
  - Avoid underutilized resources
- Contexts may expand and shrink
  - Hypervised approach
  - Maximize overall throughput
  - Use dynamic feedback both from application and runtime
Scheduling contexts

- Dynamic adjustment of context size
  - Based on computation velocity feedback
Toward a common runtime system?

i.e. Unified Software Stack

- There is currently no consensus on a common runtime system
  - Could we agree on a common runtime stack?
  - Technically feasible, but very challenging
Major Challenges are ahead…

We are living exciting times!

more information:
http://runtime.bordeaux.inria.fr