A compiler toolkit for array-based languages targeting mixed CPU/GPU systems

Rahul Garg
PhD student, McGill
Supervisor: Prof Laurie Hendren
Two important trends:
- Emergence of general purpose GPUs (GPGPUs)
- Popularity of array-based languages

Enable development of compilers that bring the two trends together

Challenges

Proposal: Reusable, shared infrastructure including compiler, library and runtime
General Purpose GPUs (GPGPUs)

5x FP peak compared to latest server CPU

<table>
<thead>
<tr>
<th></th>
<th>Xeon E5</th>
<th>FirePro W9000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cores</td>
<td>8</td>
<td>32</td>
</tr>
<tr>
<td>Threads</td>
<td>16</td>
<td>Thousands</td>
</tr>
<tr>
<td>Peak FP64 perf (Gflops/s)</td>
<td>~200</td>
<td>~1000</td>
</tr>
</tbody>
</table>
CPU + (GPU/Many-core) everywhere

- Supercomputers
- Workstations
- Laptops
- Tablets
Programming GPGPUs

• Two dominant APIs: CUDA and OpenCL
• Both of them are low-level:
  – Require management of GPU resources
  – Require GPU-specific optimization
• Nvidia CUDA: Mature but proprietary
• OpenCL is an industry standard
Dynamic array-based languages

a = zeros(1000);
b = zeros(1000);
c = a(1,:) * b(:,1);
Typical dynamic array-based languages

- No explicit type declarations
- Built-in high-level array operators
- Very flexible indexing schemes
- Both vectorized operations as well as explicit loops
- Interpreter + JIT compiler for parts of programs
- Language runtime with automatic memory management
Using GPUs

• Approach 1: Provide library:
  – D = gpu_mult(A,B)

• Approach 2: Mark GPU sections. Ask language implementation for assistance

  gpu_begin()
  D = A*B
  for ....
  gpu_end()
Scenario:

• You have an existing CPU-based language implementation for an array-based language.

• Evil boss heard about GPUs. Comes up with GPU sections.

• Now boss has asked you to write a GPU backend for GPU sections
Compiling for CPUs

Source
- Frontend
- Type inference
- More analysis

CPU codegen
- CPU code

Language Runtime
Compiling for CPUs + GPUs

Source + GPU annotations

- Frontend
- Type inference
- More analysis

CPU codegen

- CPU code

GPU codegen

- GPU code
  - GPU numerics
  - GPU runtime

Language Runtime
Not everything runs on GPUs

Programs can be broadly classified into numerical and non-numerical parts

Some of the numerical parts will run on GPU

Complex data structures, file IO etc. still on CPU

Hence, a GPU compiler need only deal with numerical things, mostly involving arrays
Proposed overall design

Source + GPU annotations

Frontend + Analysis

Identify numerical sections

CPU codegen

New IR

Velociraptor

Non-numeric CPU code

Glue

Language Runtime
Proposed overall design

Source + GPU annotations
Frontend + Analysis
Identify numerical sections
CPU codegen

Non-numeric CPU code

New IR
Velociraptor

CPU + GPU codegen
CPU code
GPU code

GPU runtime
Glue

Language Runtime
• VRIR (Velociraptor IR) is the input representation for Velociraptor
• Typed attributed abstract syntax tree (AST)
• Flexible built-in array operators and indexing
• Flexible array layout schemes
• Optionally indicate which statements to execute on GPU
• Not tied to any one source language
Velociraptor: Codegen

- First implementation is now done
- Generates LLVM + pthreads for CPUs
  - Tested on x86-64, plans to port to ARM

- Generates OpenCL for GPUs
- Tested on AMD and Nvidia GPU targets
**GPU Runtime: VRuntime**

- Handles data transfers between CPU & GPU
- Tries to avoid unneeded data transfers
- Tries to perform data transfers in parallel with computation

- Abstract out OpenCL API
- Provides a dispatch queue for all GPU kernel calls
- Non-blocking
- CPU and GPU can work in parallel

---

**CPU + GPU codegen**

- CPU code
- GPU code
- GPU numerics
- GPU runtime
• GPU architectures are quite diverse
• Not all vendors provide OpenCL libraries
• Thus, I wrote an autotuning library (RaijinCL)

• Search parameters such as tile size, SIMD length, loop unrolling, work group size etc.
• Implements operations such as matrix multiplication, trigonometric functions, reductions
Integrating VelociRaptor

• In your code generator
  – Identify and outline numerical sections
  – Compile numerical sections to VRIR
  – Either provide VRIR as XML, or use C++ APIs

• Provide glue code for language runtime
  – Tell VelociRaptor the structure of your array objects
  – Routines to do object allocation, integrate with memory management
  – Integrate with error reporting
Integration with multiple languages

• McVM:
  – A virtual machine for MATLAB built at our lab
  – Integrating Velociraptor only for parfor loops and GPU sections

• Python:
  – Proof-of-concept compiler for a numeric subset of Python+NumPy
  – Requires manual type annotations
  – All codegen being done by Velociraptor
RaijinCL DGEMM: AMD Radeon 7970
RaijinCL DGEMM: Nvidia Tesla C2050
Benchmarks

- Matrix add
- Matrix mult

Speedups

- CPU parallel
- GPU

23
Future work

• Finish described stuff (about 99% done)
• Loop optimization
• Scheduling for optimal use of CPU+ multiple GPUs
• Automatically identifying parts which should be executed on GPU
• Look into CUDA support
• Graduate. Make money.
Thanks!

• Group website: http://www.sable.mcgill.ca/mclab
• Email: rahul.garg@mail.mcgill.ca
• Compiler writer? Alpha builds available end of November
• Hardware vendor? We want to test on your hardware!
• MATLAB/Python user? We want your benchmarks!