Jet:
A Language and Heterogeneous Compiler for Fluid Simulations

dan bailey

double negative visual effects
Double Negative

- Largest Visual Effects studio in Europe
- Offices in London and Singapore
- Large and growing R & D team
Squirt

- Proprietary Fluid Solver
- Plausibility over Accuracy
- Focus on Parallel Research
GPU Acceleration

- Computational Bottleneck
- Introduced New GPU Poisson Solver
- Over 70% of Projections Accelerated
Offloading Some Computation to GPU

- Succumb to Amdahl’s Law
- Memory bandwidth limited
- GPU needs to do everything
GPU Issues

- GPU development very low-level
- Require dual GPU / CPU codebases
- Can only use NVidia hardware
Jet Language + Compiler

- Performance
- Productivity
- Portability
Jet Language

- DSL for simulation
- Expressive, high-level
- Restrictive, ‘atomic’ primitives:
Jet Compiler

- Uses LLVM Infrastructure
- Re-usable Transformations
- Target Agnostic
- Voxel primitive is for tackling 3D grids
- Kernel invoked for each voxel in grid
- This example shows one such invocation
grid: (0,0,0)

- Term “focus” used for target voxel
- Colon-bracket operator introduced
- Each grid cell calculated independently
- Focus shown in red, neighbours in yellow
- All voxel offsets determined relatively
- Simplifies accessing cells and neighbours
- Borrow heavily from functional programming
- All voxels must return their new value
- Only output parameter is mutable
Weighted_Blur : Voxel<Box> (output, input)
{
  limit(1, 1, 1, 1, 1, 1, 1);

  value = input:(0, 0, 0) * 4
  + input:(-1, 0, 0) + input:(1, 0, 0)
  + input:(0, -1, 0) + input:(0, 1, 0)
  + input:(0, 0, -1) + input:(0, 0, 1);

  return value / 10;
}
- Reduce primitive can also handle grids
- This example sums the voxels marked “Interior”
- Trivial calculation when done sequentially
Reduce

- Need an initial value to start the reduction
- Identity introduced for input operations
- Identity is zero and marked in green
- Now examine handling reduction in parallel
- First stage is an input reduction on all values
- Use identity in performing this first operation
- Main reduction operations now performed
- Multiple stages of parallel reduction required
- Each stage sums the previous reduction pairs
```c
#define INTERIOR 0
#define VALUE 0

int Interior_Count (int reduction, int value) {
    int identity(0);

    if (is_input()) {
        return reduction + (value == INTERIOR);
    }

    return reduction + value;
}
```
Jet compiler has a frontend that produces LLVM IR
IR produced is “parallel-aware”
A valid form of the IR used in assisting transformation
• Declared, but undefined Jet intrinsics used
• Custom passes handle translation of Jet intrinsics
• Some passes generic, others specific to target / data layout
- Standard optimisations still applied to IR
- X86 and PTX backends currently supported
- Backends used to produce machine code for CPU / GPU
GPU CG with LLVM

- NVidia’s LLVM PTX backend (Grover)
- Open-source LLVM PTX Backend (Chiou, Holewinski)
- Experimental PTX Backend for AnySL (Rhodin)
- LLVM to AMD IL (Villmow)
NVidia PTX

- NVidia GPU Compute ISA
- JIT Compiled to Native GPU ISA
- Supported by CUDA and OpenCL

<table>
<thead>
<tr>
<th></th>
<th>LLVM IR</th>
<th>PTX</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSA</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Integer Types Signed</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Register Set</td>
<td>Infinite</td>
<td>Finite</td>
</tr>
</tbody>
</table>
Open-source LLVM PTX Backend

- Target-independent codegen approach
- Instruction-selection in Tablegen
- Supports PTX 2.0+ and SM 1.0+
- Supports 32-bit and 64-bit targets
PTX Backend Usage

• Generating PTX code with llc:

```
llc -march=ptx32 < source.ll
llc -march=ptx64 < source.ll
```

• Generating PTX code with clang:

```
clang -ccc-host-triple ptx32 source.c -O1 -S
clang -ccc-host-triple ptx64 source.c -O1 -S
```

github.com/jholewinski/llvm-px-samples
Current State

- Instruction selection:
  - arithmetic, bitwise, control-flow
- Multiple address spaces
- Special register intrinsics
- Preliminary function call support
- Thread synchronisation
Register Usage

- Now use untyped registers
- Register overflow handled by spilling
- Register allocation done by ptxas
Performance Considerations

- Divergent branching
- Warp occupancy
- Computation vs register re-use
In Development

- Better function call support
- Stack frame allocation
- PTX-specific optimisations
- Predicated instructions
- Improved debugging
Jet Language and Compiler

- Separates logic from implementation
- LLVM X86 and PTX targets work well
- Future-proof methodology

Next Steps

- More Primitives
- More Targets
- More Optimisations
**Squirt**

- Grid Solver (80 kernels)
  - Multigrid Poisson Solver
- RK2 Cubic Advection Scheme

<table>
<thead>
<tr>
<th></th>
<th>MGPCG</th>
<th>Jet CPU</th>
<th>Jet Q4000</th>
<th>Jet C2050</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>64^3</td>
<td>1.76s</td>
<td>1.02s</td>
<td>0.25s</td>
<td>0.14s</td>
<td>7.29x</td>
</tr>
<tr>
<td>128^3</td>
<td>11.5s</td>
<td>9.10s</td>
<td>1.58s</td>
<td>0.75s</td>
<td>12.2x</td>
</tr>
<tr>
<td>256^3</td>
<td>93.1s</td>
<td>93.5s</td>
<td>13.2s</td>
<td>6.07s</td>
<td>15.4x</td>
</tr>
</tbody>
</table>
Squirt

- Grid Solver (80 kernels)
- Multigrid Poisson Solver
- RK2 Cubic Advection Scheme

<table>
<thead>
<tr>
<th></th>
<th>Squirt</th>
<th>Jet CPU</th>
<th>Jet GPU</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>50^3</td>
<td>1m58s</td>
<td>1m40s</td>
<td>12.5s</td>
<td>6.70x</td>
</tr>
<tr>
<td>100^3</td>
<td>17m40s</td>
<td>13m24s</td>
<td>1m19s</td>
<td>10.4x</td>
</tr>
<tr>
<td>200^3</td>
<td>3h06m</td>
<td>2h41m</td>
<td>11m45s</td>
<td>13.7x</td>
</tr>
</tbody>
</table>
Jet

- Flexible, Expressive Language
- Fast, Heterogeneous Compiler
- Productivity, Performance, Portability

Double Negative are recruiting compiler engineers!

Email: dan@dneg.com