OpenCL Code from Parallel Loops

Alexey Kravets, **Alexander Monakov**, Andrey Belevantsev

Institute for System Programming

GCC Developers' Summit, October 27, 2010

What is OpenCL

OpenCL: a programming model for massively parallel processors

GPU offloading issues:

- Hundreds of ALUs per chip, thousands of in-flight threads
- Computations are launched as parallel blocks of threads
- Separate memory space
- Separate compilation of GPU code

AMD has an OpenCL implementation targeting CPUs as well

Life of a Simple OpenCL Program

1. Initialize library context and command queue
2. Prepare accelerator-side code
   - Load and compile OpenCL source code: from a separate file or from a string constant
   - Or load a previously compiled blob
3. Allocate memory buffers, copy-in data
4. Set args and enqueue kernel launches
5. Copy-out, release buffers

Life of a Simple OpenCL Program

1. Initialize library context and command queue
2. Prepare accelerator-side code
   - Load and compile OpenCL source code: from a separate file or from a string constant
   - Or load a previously compiled blob
3. Allocate memory buffers, copy-in data
4. Set args and enqueue kernel launches
5. Copy-out, release buffers

Life of a Simple OpenCL Program

1. Initialize library context and command queue
2. Prepare accelerator-side code
   - Load and compile OpenCL source code: from a separate file or from a string constant
   - Or load a previously compiled blob
3. Allocate memory buffers, copy-in data
4. Set args and enqueue kernel launches
5. Copy-out, release buffers

Our Work

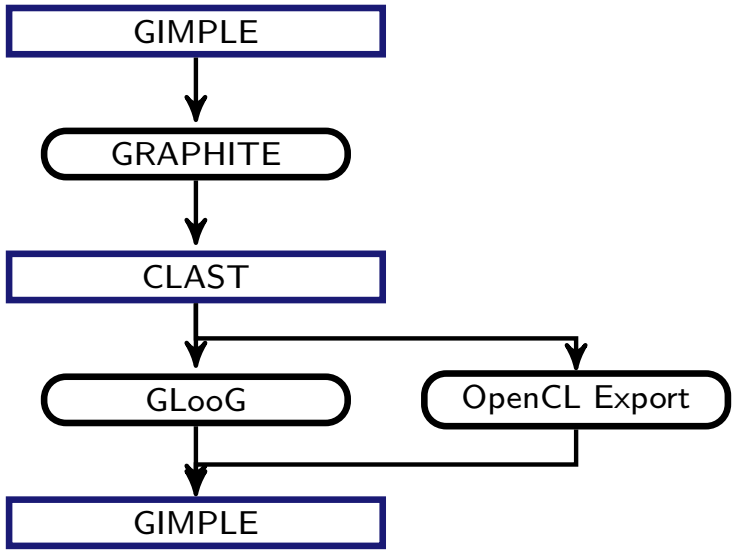
Generating OpenCL code during compilation

- Not making an OpenCL compiler!
- Rely on external OpenCL support library

Proof-of-concept implementation

- Works for simple kernels
- Does not speed up real code
- Highlights spots in GRAPHITE that could be improved

What we need to emit OpenCL code



1. Detect code that can be offloaded
   Close to autopar: a counted loop without cross-iteration dependencies
   Use GRAPHITE's dependence testing code
2. Generate equivalent OpenCL code
   Use CLAST high-level loop representation
   Replace GLooG with a custom CLAST→GIMPLE generator

Code generation example

```
for (scat_1 = 0; scat_1 < N; scat_1++)
  for (scat_3 = 0; scat_3 < M; scat_3++)
    S1(scat_1, scat_3);
```

may become

```
__kernel void
opencl_auto_function(int ofs_0, int div_0, int mod_0)
{
  size_t tid = get_global_id(0);
  int scat_1 = tid / div_0 % mod_0 + ofs_0;
  int scat_3;
  for (scat_3 = 0; scat_3 < M; scat3++)
    S1(scat_1, scat_3);
}
```

Code generation example

```
for (scat_1 = 0; scat_1 < N; scat_1++)
  for (scat_3 = 0; scat_3 < M; scat_3++)
    S1(scat_1, scat_3);
```

or, it may become

```
__kernel void
opencl_auto_function(int ofs_0, int div_0, int mod_0,
                     int ofs_1, int div_1, int mod_1)
{
  size_t tid = get_global_id(0);
  int scat_1 = tid / div_0 % mod_0 + ofs_0;
  int scat_3 = tid / div_1 % mod_1 + ofs_1;
  S1(scat_1, scat_3);
}
```

Code Generation

OpenCL code is emitted into a string constant and compiled by libOpenCL.so at runtime.

Generator: a pretty printer from CLAST + subset of Gimple to C

- No PHI nodes: GRAPHITE rewrites regs live across BBs to 1-element arrays
- No calls, no branches
- Need to rewrite array/pointer references (separate memory space)
  Rely on info from data-dep analysis (base object) to link references to unique memory regions

Host-side Code Surgery

1. Library context, command queue
   - Need to initialize once per executable
   - Create common variables
   - Insert code in the beginning of each function
2. OpenCL code compilation
   - Need to compile once per SCoP
   - Create static variables
   - Insert code in the beginning of the SCoP

Host-side Code Surgery

1. Library context, command queue
   - Need to initialize once per executable
   - Create common variables
   - Insert code in the beginning of each function
2. OpenCL code compilation
   - Need to compile once per SCoP
   - Create static variables
   - Insert code in the beginning of the SCoP

Host-side Code Surgery

1. Kernel launches
   - Need to replace loop nests with linear code
   - Walk the CLAST loop tree
   - For parallel loops, emit a BB with OpenCL calls
   - Otherwise, emit GIMPLE code like GLooG
2. Memory copying
   - Need to know exactly which regions to copy
   - Need to create memory buffers (per SCoP)
   - Need to minimize copying

Host-side Code Surgery

1. Kernel launches
   - Need to replace loop nests with linear code
   - Walk the CLAST loop tree
   - For parallel loops, emit a BB with OpenCL calls
   - Otherwise, emit GIMPLE code like GLooG
2. Memory copying
   - Need to know exactly which regions to copy
   - Need to create memory buffers (per SCoP)
   - Need to minimize copying

Minimizing memory copying

```
for ()
{
  host_write(B[]);

  for ()
  {
    gpu_read(A[], B[]);
    gpu_write(A[]);
  }

  host_read(A[]);
}
```

Minimizing memory copying

```
copyin(A);
for ()
{
  host_write(B[]);
  copyin(B);
  for ()
  {
    gpu_read(A[], B[]);
    gpu_write(A[]);
  }
  copyout(A);
  host_read(A[]);
}
```

Cost Model

- Copy-in/copy-out cost: require that memory copying is at least one nesting level above the parallelized loop (i.e. there are "asymptotically less" copies than kernel launches)
- Data reuse in the kernel: for the innermost dataref, require that nesting depth is greater than the data dimension (number of indices)
- Avoid generating kernels for deeply nested loops in a SCoP

Results

Almost no change on SPEC2K, Polyhedron

- Most transformations are prohibited by the cost model
- With the cost model disabled, wupwise and lucas improve by 7%, parser regresses by 53%

Improvements on small kernels (PolyKernels)

- Matmul: 3x on a quad-core CPU, 39x on a GPU
- Some other kernels also improve (up to 3x)
- No large regressions on a CPU, one on a GPU (2x slowdown)

Why matmul actually works

```
for (i)
  for (j)
  {
    t[0] = 0;
    for (k)
      t[0] += B[i][k] * C[k][j];
    A[i][j] = t[0];
  }

__kernel void opencl_auto_func(...)
{
  size_t = get_global_id(0);
  i = ...
  j = ...
  t = 0;
  for (k)
    t += B[i][k] * C[k][j];
  A[i][j] = t;
}
```

Wishlist

1. Larger SCoPs

```
for (...)
{
  for (...)
    A[i] += B[j];
  std::cerr << A[k];
}
```

2. Multi-dimensional variable-size arrays in SCoPs
   SCEV for `&A[i][j]` is not linear

3. Need privatization/expansion
4. Need good cost model

Thanks!

Questions?