A New Compilation Path: From Python/NumPy to OpenCL

Xunhao Li (IBM Software Lab, Canada)
Rahul Garg (McGill University)
J. Nelson Amaral (University of Alberta)
Special Purpose Computation on Graphics Processing Units
GPGPU from high-level languages?

Python

MATLAB

Missing programming models and compilers

???

GPGPU APIs Such as AMD CAL, nVidia CUDA and OpenCL
Workflow for a Python programmer

- Python/NumPy Code
- Profile prototype
- OpenMP sections
- OpenCL sections
- Glue code
  - OpenMP Compiler
  - Native OpenCL Compiler
  - Python Interpreter
- CPU/GPU Execution
unPython:

```
for \( i_1 \) in prange(\( u_1 \)):
    for \( i_2 \) in prange(\( u_2 \)):
        \ldots
        for \( i_m \) in prange(\( u_m \)):
            for \( i_{m+1} \) in range(\( u_{m+1} \)):
                \ldots
                \ldots
                for \( i_d \) in range(\( u_d \))
                #loop body
```

Figure 5.1: Loop nests considered for array access analysis
Python/NumPy to CPU/GPU

- A special parallel loop annotation to accelerate a particular loop nest using the GPU
- Programming model based on shared memory model similar to OpenMP
- Compiler attempts to identify data accessed inside the loop
- Copies all relevant data to GPU
- Generates GPU code (using CAL API)
- Executes in GPU and copies data back
- If analysis fails, fall back to generated C/C++ code
Workflow using unPython

1. Python/NumPy Code
2. Profile prototype
3. Add Annotation to existing code
4. New Compilation Framework
5. CPU/GPU Execution
Data Transfer to GPU

To transfer data to GPU execution:
  Identify set of memory locations accessed
  Find the amount of GPU memory required
  Translate CPU memory addresses to GPU addresses

Hard to do ahead-of-time, not enough information

We introduce a new just-in-time compiler named "jit4OpenCL"
jit4OpenCL

- jit4OpenCL is a specialized compiler:
  - Only to handle parallel loop-nests
  - Generated OpenCL code targets GPU execution
- For loops marked parallel, unPython inserts a call to jit4OpenCL to compile the loop
- jit4OpenCL performs data access analysis and attempts to generate OpenCL code
- If jit4OpenCL fails, then execution falls back to OpenMP code generated by unPython
- If jit4OpenCL succeeds, it generates OpenCL code and code for all relevant data transfers
Problem:

Given a loop nest:

Which memory locations are accessed?

Do these locations fit in the GPU memory?

If not, how to make them fit?
Key Insights:

Use Linear Memory Access Descriptors to represent referenced locations

Restrict to loops with constant-stride accesses

Further restrict to loops in which each reference is to a unique memory location.

Tile loop nest to make it fit in the GPU memory.
Array access analysis in jit4OpenCL

• identifies the set of memory locations accessed by array references in a loop nest

• uses a new array access analysis algorithm based on Linear Memory Access Descriptors (LMADs) [PaekHoeflingPaduaTOPLAS02]

• LMADs:
  • represent memory-address expressions directly
  • require subscripts to be affine expressions of loop counters
Constant-Stride LMADs (CSDLMADs)

- Given a loop nest with $d$ nested loops:
  - The lower bound of all loops is 0
  - Upper bounds are affine functions of the loop indices of the outer loops
  - The indices and the strides of the loops are:

\[
\vec{i} = (i_1, i_2, \cdots, i_d)
\]
\[
\vec{s} = (s_1, s_2, \cdots, s_d)
\]
CSLMADs (cont.)

- The set of memory locations accessed by a CSLMAD are given by:

\[ M = f(\vec{i}) = b + \sum_{k=1}^{d} s_k \times i_k \]

- Base
- Strides
- Indices
char A[P][80]

1: for(u=0; u<100; u++)
2: for(v=0; v< v+5; v++)
3: ... = A[u+v+1][2*u+2];

\[
f(i,j) = \&A[0][0] + 80 \times (u+v+1)+(2 \times u +2)
\]
\[
= \&A[0][0] + 82 \times u + 80 \times v + 82
\]
\[
= (\&A[0][0] + 82) + 82 \times u + 80 \times v
\]

\[
M = f(\vec{i}) = b + \sum_{k=1}^{d} s_k \times i_k
\]
Restricted CSLMADs

CSLMAD:

\[ M = f(\tilde{i}) = b + \sum_{k=1}^{d} s_k \times i_k \]

Additional constraint for RCSLMAD:

\[ s_{r_k} \geq s_{r_d} - 1 + \sum_{j=k+1}^{d} u_{r_j} \times s_{r_j} \]

- \( r_k \): position of stride \( s_{r_k} \) in a stride list sorted in decreasing order.
- \( s_{r_d} \): smallest stride
- \( s_{r_1} \): largest stride

In an RSCLMAD each reference is to a unique memory location.
CSLMAD $\times$ RCSLMAD (examples)

CSLMAD:

$M = 3u+v,\ u<10,\ v<4$

$(u, v) = (0,3) \rightarrow M = 3$
$(u, v) = (1,0) \rightarrow M = 3$

RCSLMAD:

$M = 4u+v,\ u<10,\ v<4$

$(u, v) = (0,3) \rightarrow M = 3$
$(u, v) = (1,0) \rightarrow M = 4$

There are no two pairs $(u,v)$ that map to the same location.
jit4OpenCL Code Generation

• For OpenCL compiler has to perform on-chip memory allocations.

• jit4OpenCL introduces new memory compactation techniques.

• Transform reference addresses to new space

• Match # of accesses with # of thread blocks (currently uses 16×16 blocks)
Data Transfers

• OpenCL API requires initialization of device memory objects with specific sizes

• Jit4OpenCL:
  • Configures the OpenCL grid
  • Sorts RCSLMAD elements
    • Use memory-address mapping for memory transfers
  • Matches RCSLMAD elements to threads
    • Decides which elements will be in local memory for each thread.
  • Redirects Array Access References
Evaluation - Benchmarks

CP: Columbic Potential (planar grid)
NB: N-Body simulation (a kernel only)
BS: Black-Sholes formulas
MB: approximation of Mandelbrot set
MM: Matrix multiplication
Comparison with jit4GPU and Handwritten OpenCL code
(on AMD Radeon 5850)
Where is time spent? (jit4OpenCL on NVidia GTX260)
Take-Away Points

• OpenCL portability comes with a performance cost
• Data transfers in OpenCL are more constrained and might be limiting speed.
• Jit4OpenCL is a quick way to test a Python/NumPy application in many platforms.
• Increased memory bandwidth in GPUs and improvement of native OpenCL compilers is likely to help with performance.