Automatic OpenCL Optimization for Locality and Parallelism Management

Xing Zhou, Swapnil Ghike
In collaboration with:
Jean-Pierre Giacalone, Bob Kuhn and Yang Ni *(Intel)*
Maria Garzaran and David Padua *(UIUC)*
Motivation

- **Tiling & fusion loops to improve locality**
  - For sequential loops:
    - If $N >$ cache size, cache misses $N$ times in the second loop;
    - If $T <$ cache size, no cache miss in the second loop.

```c
for(int i = 0; i < N; i++)
    A[i] = ...;
for(int j = 0; j < N; j++)
    ... = A[j];
```

- For OpenCL:

```c
__kernel void f(__global char *A) {
    int i = get_global_id(0);
    A[i] = ...;
}
__kernel void g(__global char *A) {
    int j = get_global_id(0);
    ... = A[j];
}
```
Motivation

- When inter-tile dependence exists:
  - The transformation becomes illegal, unless introducing additional synchronization:

```c
parallel for(int i = 0 : N-1)
    A[i] = ...;
parallel for(int j = 0 : N-1)
```

- If we allow some redundant computation:

```c
parallel for(int t = 0 : N/T) {
    private a[t*T-1 : min(N, (t+1)*T)];
    for(int i=t*T-1; i<min(N, (t+1)*T)+1; i++)
        a[i] = ...;
    for(int j = t*T; j<min(N, (t+1)*T); j++)
        ... = a[j-1] + a[j] + a[j+1];
}
```
Motivation

• Tile with redundant computation:
Across Kernel Boundaries

- We need to optimize kernel code across kernel boundaries

What the host code compiler (GCC) see:

```c
clCreateProgramWithSource()
clBuildProgram()

clSetKernelArg(kernel_f, 0, sizeof(cl_mem), &mem_A);
clSetKernelArg(kernel_g, 0, sizeof(cl_mem), &mem_A);

size_t global_work_size[] = {N};
clEnqueueNDRangeKernel(queue, kernel_f, 1, NULL,
                        global_work_size, NULL, 0, NULL, &event);
clEnqueueNDRangeKernel(queue, kernel_g, 1, NULL,
                        global_work_size, NULL, 1, &event, NULL);
```

What the kernel code compiler see:

```c
__kernel void f(__global char *A) {
    int i = get_global_id(0);
    A[i] = ...;
}

__kernel void g(__global char *A) {
    int j = get_global_id(0);
    ... = A[j] + A[j+1];
}
```

```
parallel for (int i = 0 : N-1)
    A[i] = ...;
parallel for (int j = 0 : N-1)
    ... = A[j] + A[j+1];
```
Across Kernel Boundaries

• Lazy compilation framework:

What the host code compiler (GCC) see:

```c
clCreateProgramWithSource()  
clBuildProgram()  
clSetKernelArg(kernel_f, 0, sizeof(cl_mem), &mem_A);  
clSetKernelArg(kernel_g, 0, sizeof(cl_mem), &mem_A);  
size_t global_work_size[] = {N};  
clEnqueueNDRangeKernel(queue, kernel_f, 1, NULL,  
    global_work_size, NULL, 0, NULL, &event);  
clEnqueueNDRangeKernel(queue, kernel_g, 1, NULL,  
    global_work_size, NULL, 1, &event, NULL);  

parallel for(int i = 0 : N-1)  
    A[i] = ...;  
parallel for(int j = 0 : N-1)  
    ... = A[j] + A[j+1];
```

- Save kernel source without compilation
- Save kernel arguments
- Save work size
- Save the event waiting relation between f and g
- Finally, compiler the kernel code
Our Approach

- **A OpenCL source to source compiler:**
  - Kernel code compiler only
  - Accept naive OpenCL kernel code containing fine-grain kernel functions as input
    - Typically each task (work item) contains the computation of a single data element in the output domain.
    - Use global buffers to pass data between kernel functions
  - Output transformed kernel source and supportive host code
  - Implemented as a pass in Cetus source to source compiler
Integer Tuple Space

- Unified representation for iteration space and index space
  - Omega Library provides a serial of manipulation routines

- Integer tuple:
  - a vector of integers; a point in space \( \mathbb{Z}^d \)
  - \( x = [x_0, x_1, \ldots, x_{d-1}] \)

- Integer tuple set:
  - A set of integer tuples; described with constraints
  - \( S = \{[0], [1], \ldots, [N - 1]\} = \{[i] : 0 \leq i < N\} \)

- Integer tuple relation:
  - A set-to-set mapping
  - \( R = \{[k] \rightarrow [x] : k = x + 1\} \)
Integer Tuple Space

- $\epsilon, \cup, \cap$, operators are defined for integer tuples and sets
- *minimum covering rectangle (MCR)*, *bottom-left point (BLP)* and *top-right point (TRP)* operators for integer tuple sets:
Kernel code analysis

• What information do we need?
  – Given a set of output elements, which kernel instances (tasks) can produce them?
    • index space (buffer array) → iteration space (implicit parallel loop)
  – Given a set of kernel instances, which input elements they need to consume?
    • iteration space → index space

• Use two relations to represent the information above:
  – Producing relations:
    • For each array base with any write set associated, the relation from the write index set to the work item id set
  – Consuming relations:
    • For each array base with any read set associated, the relation from the work item set to the read index set
Kernel code analysis

Example: the producing and consuming *relations*

```c
__kernel void g(__global char *A) {
    int j = get_global_id(0);
}
```

- Consuming *relation* of array A:
  
  `\{[j] \rightarrow [t]: j-1 \leq t \leq j+1\}`

- Producing *relation* of array B:
  
  `\{[t] \rightarrow [j]: t=j\}`
Kernel code analysis

- Example of the algorithm

```c
__kernel void g(__global char *A) {
    int j = get_global_id(0);
}
```

- The result consuming relation for array A is $R$:

$$R1 = \{[id_0] \rightarrow [x_0] : x_0 = j - 1 \land j = id_0\}$$
$$R2 = \{[id_0] \rightarrow [x_0] : x_0 = j \land j = id_0\}$$
$$R3 = \{[id_0] \rightarrow [x_0] : x_0 = j + 1 \land j = id_0\}$$

$$R = R1 \cup R2 \cup R2 = \{[id_0] \rightarrow [x_0] : id_0 - 1 \leq x_0 \leq id_0 + 1\}$$
Tiling & fusion transformation

• How to determine tile size?
  – Goal: the data touched within a tile can fit local storage
  – Problem: the size of accessed data of different kernels are related to each other because of producing-consuming relation, and also related to the tile size

• Solution: Symbolic tiling
  – Assign symbolic boundaries for the tile
  – Build a data size function of the boundary symbols.
  – Search for the boundary symbol values with which the total data size can fit local storage
Tiling & fusion transformation

- **Algorithm of symbolic tiling:**

  ```
  Input: ConsumingRel[K] and ProducingRel[K]
  for each kernel K
  Output: Tile[K] for each K
  Initialization: Tile[K] = ∅, for each K

  K_{start} = the last kernel in a topological order;
  K_{end} = the last kernel in the same topological order;
  Tile[K_{start}] = \{[id_0, ..., id_{d-1}] : s_0 \leq id_0 < s_0 + len_0 \wedge \ldots \wedge s_{d-1} \leq id_{d-1} < s_{d-1} + len_{d-1}\};

  for(each K in K_{start}, ..., K_{end} in reversed topological order) {
    T = Tile[K];
    for(each array \ A that is an argument passed to K) {
      C = ConsumingRel[K][A];
      S = C(T);
      for(each kernel K_1 which is an ancestor of K in the DAG) {
        if(ProducingRel[K_1][\ A] \neq null) {
          P = ProducingRel[K_1][\ A];
          T_1 = P(S);
          Tile[K_1] = Tile[K_1] \cup T_1;
        }
      }
    }
  }
  ```
Tiling & fusion transformation

• Algorithm of symbolic tiling:

Topological order: k1, k2, k3, k4
Processing order: k4, k3, k2, k1

Fused kernel:
Tiling & fusion transformation

- **Algorithm example:**
  - Kernel \( g \)'s consuming function:
    \[ R = \{ [id_0] \rightarrow [x_0] : id_0 - 1 \leq x_0 \leq id_0 + 1 \} \]
  - Symbolic tile:
    \[ Tile[g] = \{ [id_0] : start_0 \leq id_0 < start_0 + len_0 \} \]
  - Kernel \( f \)'s producing function:
    \[ P = \{ [x_0] \rightarrow [id_0] : x_0 = id_0 \} \]
  - Calculate kernel \( f \)'s tile:
    \[ S = R(Tile[g]) = \{ [x_0] : start_0 - 1 \leq x_0 < start_0 + len_0 + 1 \} \]
    \[ Tile[f] = P(S) = \{ [id_0] : start_0 - 1 \leq id_0 < start_0 + len_0 + 1 \} \]

```c
__kernel void f(__global char *A) {
    int i = get_global_id(0);
    A[i] = ...;
}
__kernel void g(__global char *A) {
    int j = get_global_id(0);
}
```
Tiling & fusion transformation

• **Search for the best tile size** $len$ :
  – **Bound constraint**: tile size must be smaller than the global work size.
    - This constraint provides the lower and upper bound for the tile size search process.
  – **Parallelism constraint**: There must be enough tiles to keep the target device busy.
    - The constraint provides an upper bound for the value of $len$ .
  – **Locality constraint**: The total size of all local buffers must fit in the local memory or cache
    - This constraint is another upper bound.
Code Generation

- Generate the fused kernel:

```c
__kernel void fused(__global char *A) {
    int gid_0 = get_global_id(0);
    int start_0 = gid_0 * len;
    __local a[len + 2];
    for (int i_0 = start_0 - 1; i_0 <
        start_0 + len + 1; i_0++)
        f(a, i_0);
    for (int i_0 = start_0; i_0 < start_0 +
        len; i_0++)
        g(a, i_0);
}

size_t global_work_size[] = {
    [N_0/len], [N_1/len], ...
    [N_{d-1}/len]};
clEnqueueNDRangeKernel(queue, kernel_fused, ...,
    global_work_size, NULL, ...);
```
Code Generation

• Generate the fused kernel with parallelism recovery:

```c
__kernel void fused(__global char *A) {
    int lid_0 = get_local_id(0);
    int gid_0 = get_group_id(0);
    int start_0 = gid_0 * len;
    __local a[len + 2];
    int i_0 = lid_0 + start_0 - 1;
    if (i_0 >= start_0 - 1 && i_0 < start_0 + len + 1)
        f(a, i_0);
    barrier(LOCAL_MEM_FENCE);
    i_0 = lid_0 + start_0;
    if (i_0 >= start_0 && i_0 < start_0 + len)
        g(a, i_0);
}
```

```c
size_t global_work_size[] = {[N_0/len] * l_0,
                           [N_1/len] * l_1, ..., [N_{d-1}/len] * l_{d-1}};
size_t local_work_size[] = {l_0, l_1, ..., l_{d-1}};
cEnqueueNDRangeKernel(queue, kernel_fused, ..., 
global_work_size, local_work_size, ...);
```
Experiments

- Platform:

<table>
<thead>
<tr>
<th>CPU</th>
<th>ATOM D525</th>
</tr>
</thead>
<tbody>
<tr>
<td>Freq.:</td>
<td>1.8GHz;</td>
</tr>
<tr>
<td></td>
<td>Number of cores: 2;</td>
</tr>
<tr>
<td></td>
<td>Number of hw threads: 4;</td>
</tr>
<tr>
<td></td>
<td>L1 data cache (per-core): 24KB;</td>
</tr>
<tr>
<td></td>
<td>L2 cache (per-core): 512KB.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GPU</th>
<th>NVIDIA ION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of</td>
<td>multiprocessors: 2;</td>
</tr>
<tr>
<td></td>
<td>Number of SPs: 16;</td>
</tr>
<tr>
<td></td>
<td>Local memory size: 16KB;</td>
</tr>
<tr>
<td></td>
<td>Global memory size: 511MB.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CPU</th>
<th>Xeon L7555</th>
</tr>
</thead>
<tbody>
<tr>
<td>Freq.:</td>
<td>1.87GHz;</td>
</tr>
<tr>
<td></td>
<td>Number of processors: 4;</td>
</tr>
<tr>
<td></td>
<td>Number of cores: 32;</td>
</tr>
<tr>
<td></td>
<td>Number of hw threads: 32 (SMT</td>
</tr>
<tr>
<td></td>
<td>disabled)</td>
</tr>
<tr>
<td></td>
<td>L1 data cache (per-core): 32KB;</td>
</tr>
<tr>
<td></td>
<td>L2 cache (per-core): 256KB;</td>
</tr>
<tr>
<td></td>
<td>L3 cache (per-processor): 24MB.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GPU</th>
<th>GeForce GTX 480</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of</td>
<td>multiprocessors: 15;</td>
</tr>
<tr>
<td></td>
<td>Number of SPs: 480;</td>
</tr>
<tr>
<td></td>
<td>Local memory size: 48KB;</td>
</tr>
<tr>
<td></td>
<td>Global memory size: 1535MB.</td>
</tr>
</tbody>
</table>
Experiments

- **Mobile platform**
  - Video post-processing application (VPP) with 3 video post processing filters: *StrongPostFilter, DenoiseDegrain* and *IppSharp*.
  
<table>
<thead>
<tr>
<th>QVGA</th>
<th>VGA</th>
<th>HD 720p</th>
<th>HD 1080p</th>
<th>GEO MEAN</th>
<th>QVGA</th>
<th>VGA</th>
<th>HD 720p</th>
<th>HD 1080p</th>
<th>GEO MEAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>GPU</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **S**: Simple tiling and fusion with global barriers
- **OB**: Tiling & fusion with redundant computation, and with global barriers
- **O**: The optimized code (tiling & fusion with redundant computation)
- **NR**: Tiling & fusion WITHOUT redundant computation (cannot produce correct result)
Experiments

• **Workstation platform:**
  – 4 iterative stencil loop applications

<table>
<thead>
<tr>
<th>Dimension of data</th>
<th>Jacobi</th>
<th>PathFinder</th>
<th>HotSpot</th>
<th>Cell</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input size</td>
<td>64K</td>
<td>100K</td>
<td>512x512</td>
<td>60x60x60</td>
</tr>
</tbody>
</table>

```c
while (some_condition) {
    clSetKernelArg(kernel, 0, input);
    clSetKernelArg(kernel, 1, output);
    clEnqueueNDRangeKernel(kernel, …);
    tmp = input;
    input = output;
    output = tmp;
}
```
Experiments

- **Workstation platform:**
  - RC: the percentage of redundant computation introduced
Conclusion

• Machine-independent memory hierarchy optimization and work item organization for OpenCL programs.
• A lazy compilation framework which makes global optimizations across kernel boundaries possible.
• Tiling & fusion transformation with redundant computation to eliminate synchronizations.
The End

• Questions?