Efficient Mapping of Irregular C++ Applications to Integrated GPUs

Rajkishore Barik (Presenter)
  Rashid Kaleem, UT Austin
  Deepak Majeti, Rice University
  Brian T. Lewis, Intel Labs
  Tatiana Shpeisman, Intel Labs
  Chunling Hu, Intel Labs
  Yang Ni, Google
  Ali-Reza Adl-Tabatabai, Google
Heterogeneous Platforms

- **Heterogeneity is ubiquitous**: mobile devices, laptops, servers, & supercomputers
- **Emerging hardware trend**: CPU & GPU cores integrated on same die, share physical memory & even last-level cache

Intel 4th generation core processors

AMD Trinity

How do we program these integrated GPU systems?
Motivation: GPU Programming

- Existing work: regular data-parallel applications using array-based data structures map well to the GPUs
  - OpenCL 1.x, CUDA, OpenACC, C++ AMP, ...

- Enable other existing multi-core applications to quickly take advantage of the integrated GPUs
  - Often use object-oriented design, pointers
    - Enable pointer-based data structures on the GPU
  - Irregular applications on GPU: benefits are not well-understood
    - Data-dependent control flow
      - Graph-based algorithms such as BFS, SSSP, etc.

Widen the set of applications that target GPUs
Contributions

• **Concord**: a seamless C++ heterogeneous programming framework for integrated CPU and GPU processors
  - **Shared Virtual Memory (SVM) in software**
    - share pointer-containing data structures like trees
  - Adapts existing data-parallel C/C++ constructs to heterogeneous computing: TBB, OpenMP
  - Supports most C++ features including **virtual functions**
  - Demonstrates programmability, performance, and energy benefits of SVM

• Available open source at [https://github.com/IntelLabs/iHRC/](https://github.com/IntelLabs/iHRC/)
Concord Framework

- Concord C++
  - CLANG
  - LLVM
  - CLANG code gen
  - OpenCL code gen

- Object: IA binary + OpenCL
  - Linker

- Execute: IA binary + OpenCL
  - Compute runtime
  - OpenCL JIT Compiler
    - OpenCL to GPU ISA
  - GPU binary

- CPU
- GPU
Concord C++ programming constructs

Concord extends TBB APIs:

```cpp
template <class Body>
parallel_for_hetero (int numiters, const Body &B, bool device);
```

```cpp
template <class Body>
parallel_reduce_hetero (int numiters, const Body &B, bool device);
```

Supported C++ features:
- Classes
- Namespaces
- Multiple inheritance
- Templates
- Operator and function overloading
- Virtual functions

Existing TBB APIs:

```cpp
template <typename Index, typename Body>
parallel_for (Index first, Index last, const Body & B)
```

```cpp
template <typename Index, typename Body>
parallel_reduce (Index first, Index last, const Body & B)
```

Currently not supported on GPU
- Recursion
- Exceptions
- Memory allocation
Concord C++ Example: Parallel LinkedList Search

```cpp
class ListSearch {
    ...
    void operator()(int tid) const{
        ... list->key...
    }};
    ...
    ListSearch *list_object = new ListSearch(...);

    parallel_for(0, num_keys, *list_object);
}

TBB Version

```cpp
class ListSearch {
    ...
    void operator()(int tid) const{
        ... list->key...
    }};
    ...
    ListSearch *list_object = new ListSearch(...);

    parallel_for_hetero (num_keys, *list_object, GPU);
}

Concord Version

```

Minimal differences between two versions

Concord Version Run on CPU or GPU

2/27/14 Programming Systems Lab, Intel Labs
Key Implementation Challenges

- Shared Virtual Memory (SVM) support to enable pointer-sharing between CPU and GPU
  - Compiler optimization to reduce SVM translation overheads

- Virtual functions on GPU

- Parallel reduction on GPU [paper]

- Compiler optimizations to reduce cache line contention [paper]
SVM Implementation on IA

![Diagram showing SVM implementation]

\[ \text{GPU}_\text{ptr} = \text{GPU}_\text{Base} + \text{CPU}_\text{ptr} - \text{CPU}_\text{Base} \]
SVM Translation in OpenCL code

```cpp
class ListSearch {
    ...
    void operator()(int tid) const{
        ...
        list->key...
    }
};
...
ListSearch *list_object = new ListSearch(...);
parallel_for_hetero (num_keys, *list_object, GPU);
```

```opencl
//__global char * svm_const = (GPU_Base - CPU_Base);
#define AS_GPU_PTR(T,p) ((__global T *) (svm_const + p))
__kernel void opencl_operator ( __global char *svm_const, unsigned long B_ptr ) {
    AS_GPU_PTR(LinkedList, list)->key...
}
```

- `svm_const` is a runtime constant and is computed once
- Every CPU pointer before dereference on the GPU is converted into GPU addressspace using `AS_GPU_PTR`
Compiler Optimization of SVM Translations

Best strategy:
- Eagerly convert to GPU addressspace & keep both CPU & GPU representations
- If a store is encountered, use CPU representation
- Additional optimizations
  - Dead-code elimination
  - Optimal code motion to perform redundancy elimination and place the translations

Eager

int **a = AS_GPU_PTR(int *, data->a);
for ( int i=0; i<N; i++)
    ... = a[i];
// a[i] is not used after this

Lazy

Overhead: 2N + 1

Best

Overhead: 1

Overhead: N

int **a = AS_GPU_PTR(int *, data->a);
for ( int i=0; i<N; i++)
    ... = AS_GPU_PTR(int *, a[i]);
Virtual Functions on GPU

Original hierarchy:
```cpp
class Shape {
    virtual void intersect() {...}
    virtual void compute() {...}
};
class Triangle : Shape {
    virtual void intersect() {...}
};
```

Virtual Function call:
```cpp
void foo(Shape *s) {
    s->compute();
}
```

Object layout with vtable:

![Diagram of object layout with vtable](image)

CPU Virtual Function call:
```cpp
void foo(Shape *s) {
    (s->vtableptr[1])();
}
```

GPU Virtual Function call:
```cpp
void foo(Shape *s, void *gCtx) {
    if (s->vtableptr[1] == gCtx->Shape::compute)
        Shape::compute();
}
```

Generated code:

- Copy necessary metadata into shared memory for GPU access
- Translate virtual function calls into if-then-else statements

- Copy to shared memory

• Copy necessary metadata into shared memory for GPU access
• Translate virtual function calls into if-then-else statements
Experimental setup

- **Experimental Platform:**
  - Intel Core 4th Generation Ultrabook
    - CPU: 2 cores, hyper-threaded, 1.7GHz
    - GPU: Intel HD Graphics 5000 with 40 cores, 200MHz-1.1GHz
    - Power envelope 15W
  - Intel Core 4th Generation Desktop
    - CPU: 4 cores, hyper-threaded, 3.4GHz
    - GPU: Intel HD Graphics 4600 with 20 cores, 350MHz-1.25GHz
    - Power envelope 84W

- **Energy measurements:** MSR_PKG_ENERGY_STATUS

- **Comparison with multi-core CPU:**
  1. GPU-SPEEDUP: speedup using GPU execution
  2. GPU-ENERGY-SAVINGS: energy savings using GPU execution
### Workloads

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Origin</th>
<th>Input</th>
<th>LoC</th>
<th>Device LoC</th>
<th>Data Structure</th>
<th>Parallel Construct</th>
</tr>
</thead>
<tbody>
<tr>
<td>BarnesHut</td>
<td>In-house</td>
<td>1M bodies</td>
<td>828</td>
<td>105</td>
<td>Tree</td>
<td>Parallel_for</td>
</tr>
<tr>
<td>BFS</td>
<td>Galois</td>
<td></td>
<td>V</td>
<td>=6.2M</td>
<td>E</td>
<td>=15M</td>
</tr>
<tr>
<td>Btree</td>
<td>Rodinia</td>
<td>Command.txt</td>
<td>3111</td>
<td>84</td>
<td>Tree</td>
<td>Parallel_for</td>
</tr>
<tr>
<td>ClothPhysics</td>
<td>Intel</td>
<td></td>
<td>V</td>
<td>=50K</td>
<td>E</td>
<td>=200K</td>
</tr>
<tr>
<td>ConnComp</td>
<td>Galois</td>
<td></td>
<td>V</td>
<td>=6.2M</td>
<td>E</td>
<td>=15M</td>
</tr>
<tr>
<td>FaceDetect</td>
<td>OpenCV</td>
<td>3000x2171</td>
<td>3691</td>
<td>378</td>
<td>Cascade</td>
<td>Parallel_for</td>
</tr>
<tr>
<td>Raytracerm*</td>
<td>In-house</td>
<td>sphere=256, material=3, light=5</td>
<td>843</td>
<td>134</td>
<td>Graph</td>
<td>Parallel_for</td>
</tr>
<tr>
<td>Skip_List</td>
<td>In-house</td>
<td>50M keys</td>
<td>467</td>
<td>21</td>
<td>Linked-list</td>
<td>Parallel_for</td>
</tr>
<tr>
<td>SSSP</td>
<td>Galois</td>
<td></td>
<td>V</td>
<td>=6.2M</td>
<td>E</td>
<td>=15M</td>
</tr>
</tbody>
</table>
Dynamic estimates of irregularity

- BFS, Btree, ConnComp, FaceDetect, SkipList & SSSP exhibit a lot of irregularities (>50%)
- FaceDetect exhibits maximum percentage of memory irregularities
Ultrabook: Speedup & Energy savings compared to multicore CPU

Average speedup of 2.5x and energy savings of 2x vs. multicore CPU
Desktop: Speedup & Energy savings compared to multicore CPU

Average speedup of 1.01x and energy savings of 1.7x vs. multicore CPU
Overhead of SW-based SVM implementation

Sw-based SVM overhead is negligible for smaller images and is ~6% for the largest image.
Conclusions & Future work

• Runs out-of-the-box C++ applications on GPU
• Demonstrates that SVM is a key enabler in programmer productivity of heterogeneous systems
• Implements SVM in software with low-overhead
• Implements virtual functions and parallel reductions on GPU
• Saves energy of 2.04x on Ultrabook and 1.7x on Desktop compared to multi-core CPU for irregular applications

• Future work:
  - Support advanced features on GPU: exceptions, memory allocation, locks, etc.
  - Support combined CPU+GPU heterogeneous execution
Cloth Physics demo using Concord:

Questions?

Please try it out: [https://github.com/IntellLabs/iHRC/](https://github.com/IntellLabs/iHRC/)