Concord: Homogeneous Programming for Heterogeneous Architectures

Rajkishore Barik, Intel Labs
Brian T. Lewis, Intel Labs

Presenters

- **Rajkishore Barik, Intel Labs**
  - Research Scientist, Programming Systems Lab
  - Interests: Compilers and runtimes for heterogeneous architectures
  - Ph.D: Rice University
- **Brian T. Lewis, Intel Labs**
  - Senior Research Scientist, Programming Systems Lab
  - Interests: Programming languages, runtimes, VM for modern heterogeneous architectures
  - Ph.D: University of Washington

Tutorial Motivation & Goals

- **Motivation**
  - GPUs should be programmed the same way as multi-core CPUs
    - No new language invention
  - How can we enable existing multi-core software stack to take advantage of GPU with minimal effort?

- **Goals**
  - Introduction to GPGPU programming model
  - Concord C++ programming model
    - Compiler and runtime framework
    - Implementation of software based SVM
    - Implementation of virtual functions and reductions on GPU
    - Compiler optimizations

What this tutorial will cover

- GPGPU architectures
  - Discrete devices
  - Integrated devices
- Existing GPGPU programming models
  - CUDA, OpenCL, C++ AMP, RenderScript
- Concord C++ programming model
  - Language constructs and restrictions
  - Compiler and runtime framework
    - Virtual Functions and Reductions on GPUs
    - Compiler optimizations
  - Experimental evaluation
- Demonstration
- Q&A

Tutorial Slides

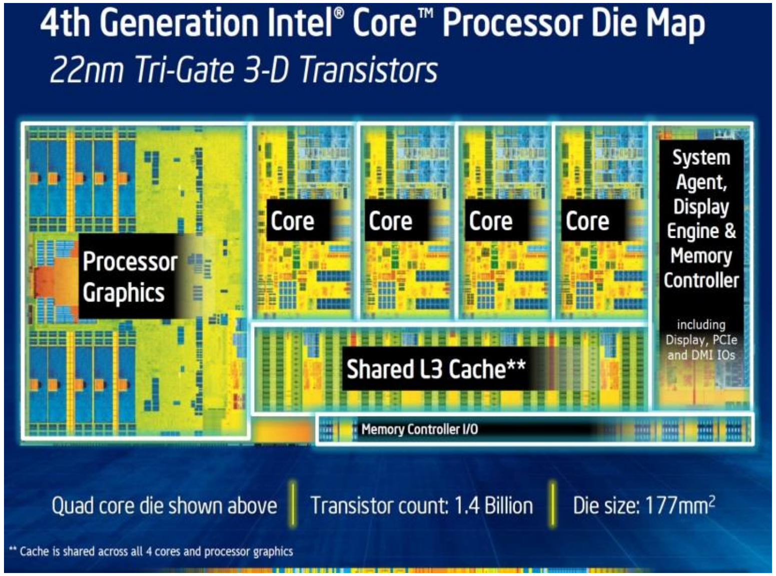
- **Available at …**

- **Research compiler is available at https://github.com/IntelLabs/iHRC/**

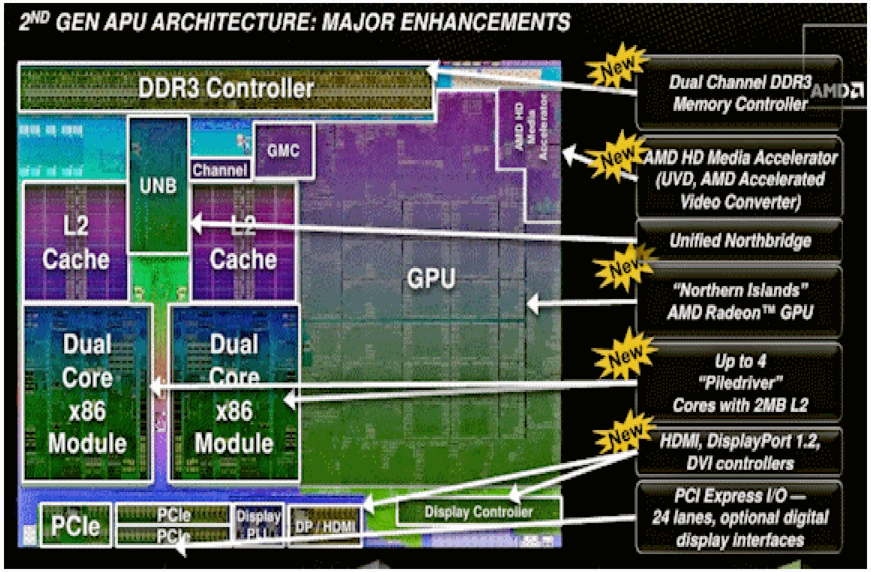
Recap: Heterogeneous Platforms

- **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



Source: http://www.hardwarezone.com.my/feature-amd-trinity-apu-look-inside-2nd-generation-apu/conclusion-118

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.

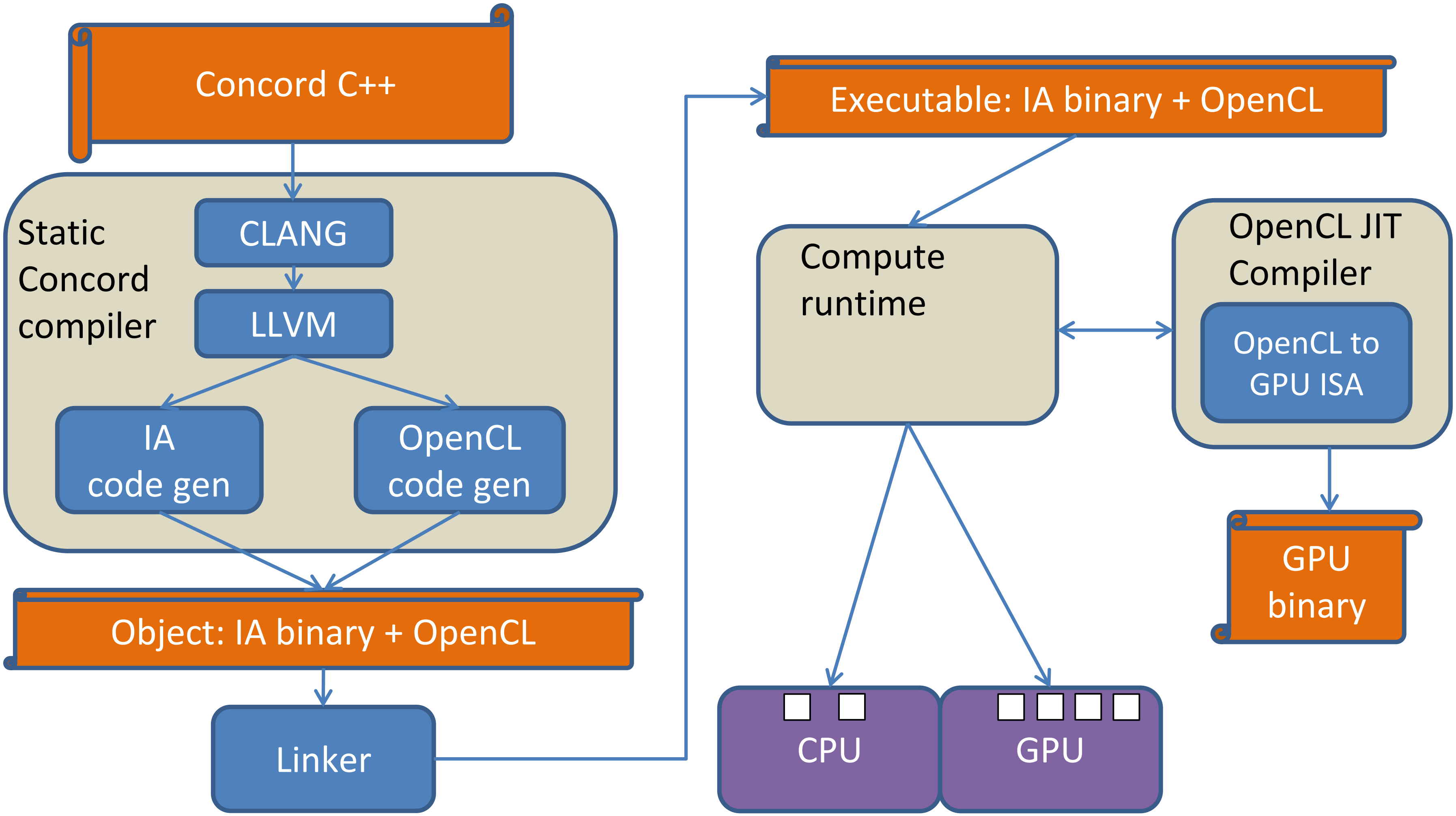
Widen the set of applications that target GPUs

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++ 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/

Concord Framework



Concord C++ programming constructs

Concord extends TBB APIs:

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

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

Existing TBB APIs:

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

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

Supported C++ features:

- Classes
- Namespaces
- Multiple inheritance
- Templates
- Operator and function overloading
- Virtual functions

Concord C++ Example: Parallel LinkedList Search

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


parallel_for(0, num_keys, *list_object);
```

TBB Version

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


parallel_for_hetero (num_keys, *list_object, GPU);
```

Run on CPU
or GPU

Concord Version

Minimal differences between two versions

Example: parallel_for_hetero

```
class Body {
   float *A, *B, *C;
   public:
      Body(float *a_, float *b_, float *c_):A(a_),B(b_),C(c_)  { }
      void operator()(int i) const { // execute in parallel
            A[i] = B[i] + C[i];
      }
};
……
Body *body = new Body(A,B,C);
parallel_for_hetero (1024, *body);
```

Example: parallel_reduce_hetero

```
class Body {
    float *A, sum;
    public:
      Body(float *aa): A(aa), sum(0.0f) {  }
      void operator()(int i) { // execute in parallel
            sum = f(A[i]);  // compute local sum
      }
      void join(Body &rhs) {
            sum += rhs.sum;  // perform reduction
       }
};
……
Body *body = new Body(A);
parallel_reduce_hetero (1024, *body);
```

Restrictions

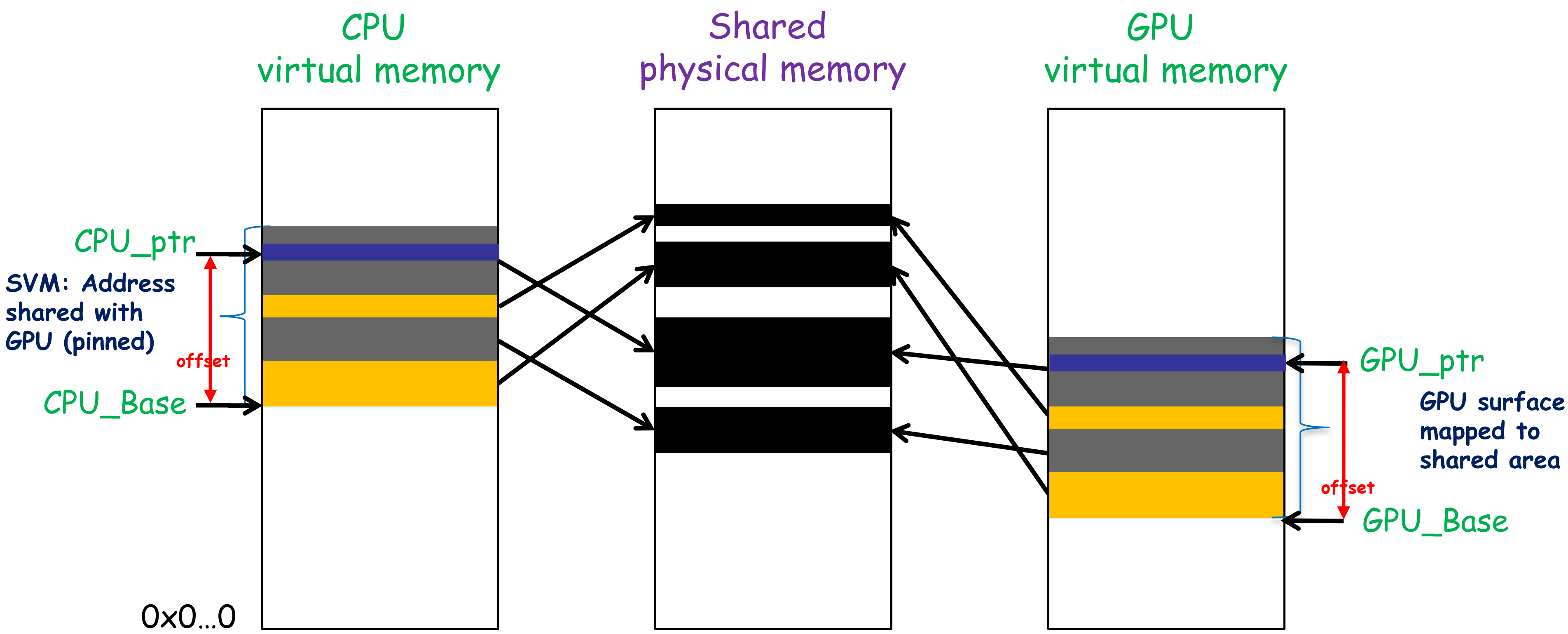
- No guarantee that the parallel loop iterations will be executed in parallel
- No ordering among different paralleliterations
  - Floatingpoint determinismis not guaranteed
- Features not yet supported on the GPU
  - Recursion (except tail recursion which can be converted to loop)
  - Exception
  - Address of local variable
  - Memory allocation and deallocation
  - Function calls via function pointers

Silently execute on CPU if these features are present in GPU code

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**
- **Compiler optimizations to reduce cache line contention**

SVM Implementation on x86



GPU_ptr = GPU_Base + CPU_ptr – CPU_Base

SVM Translation in OpenCL code

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

parallel_for_hetero (num_keys, *list_object, GPU);
```

Concord C++

```
//__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…
}
```

Generated OpenCL

- 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

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

Eager

```
int **a = AS_GPU_PTR(int *, data->a);
for ( int i=0; i<N; i++)
  … = AS_CPU_PTR(int,
        AS_GPU_PTR(int, a[i]));
```

Overhead: 2N + 1

Lazy

```
int **a = data->a;
for ( int i=0; i<N; i++)
  … = AS_GPU_PTR(int *, a)[i];
```

Overhead: N

Best

```
int **a = AS_GPU_PTR(int *, data->a);
for ( int i=0; i<N; i++)
  … = a[i];
```

Overhead: 1

- **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

Virtual Functions on GPU

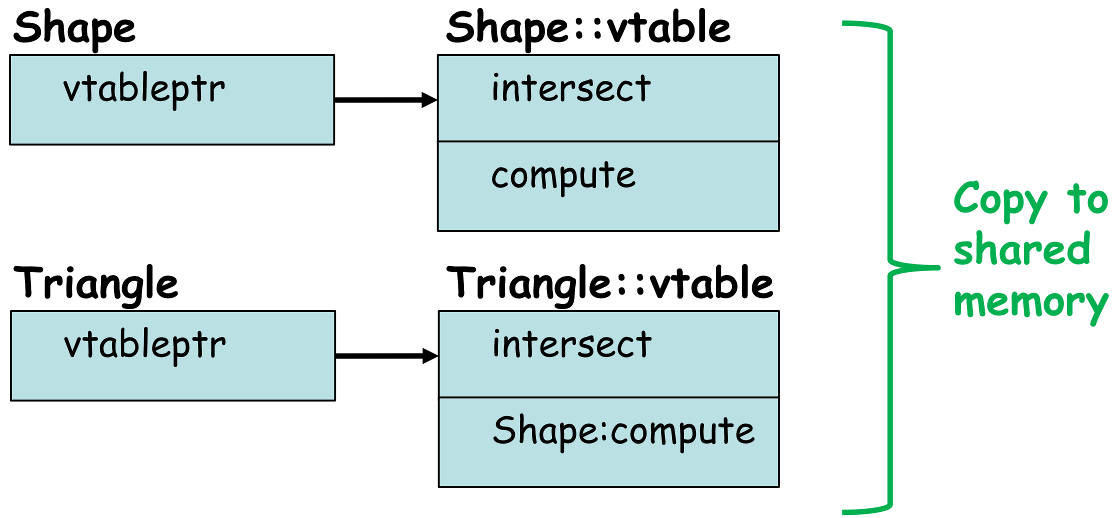
Original hierarchy:

```
class Shape {
  virtual void intersect() {…}
  virtual void compute() {…}
};
class Triangle : Shape {
  virtual void intersect() {…}
};
```

Virtual Function call:

```
void foo(Shape *s) {
  s->compute();
}
```

Object layout with vtable:



CPU Virtual Function call:

```
void foo(Shape *s) {
  (s->vtableptr[1])();
}
```

GPU Virtual Function call:

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

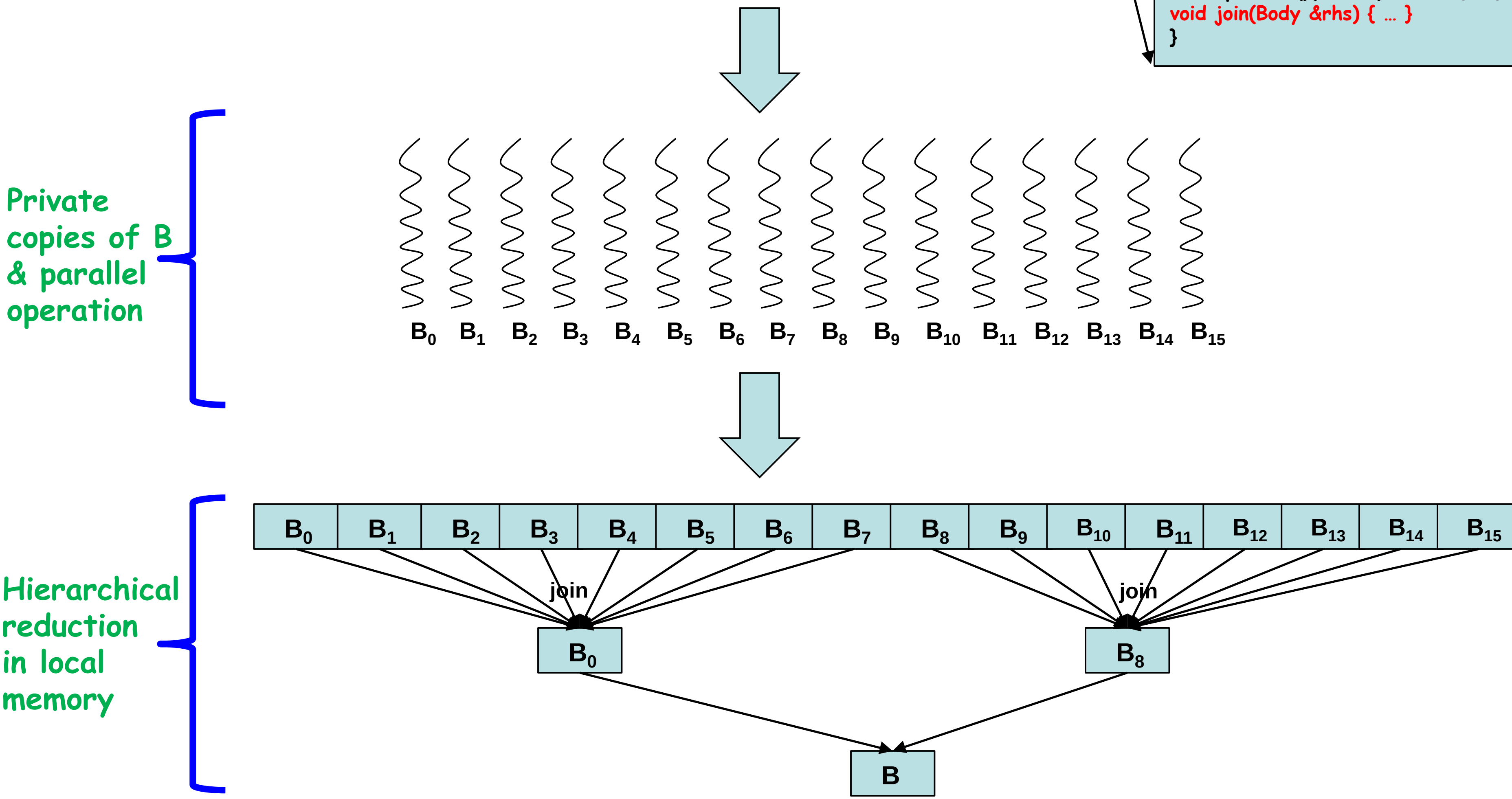
Original code | **Generated code**

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

Parallel Reduction on GPU

parallel_reduce_hetero(16, B, false)

```
class Body {
…
void operator()(int tid)  const { … }
void join(Body &rhs) { … }
}
```



Compiler Optimization for Cache Contention

- **Integrated GPUs use a unified cache among all GPU cores**
  - **Contention among GPU cores to access same cache line**
    - **number of simultaneous read and write ports to a cache line may not be same as the number of GPU cores**

```
void operator ()(int i) {
  for (j=0; j<N; j++)
      ... = a[j];
}
```

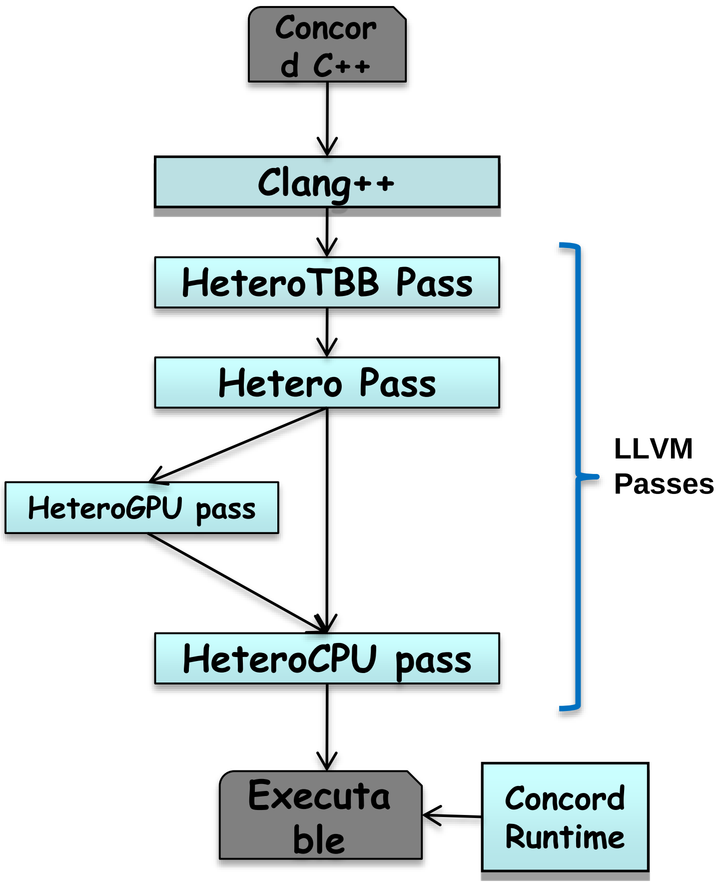
All GPU cores access same data

```
void operator ()(int i) {
  int start = i / W; /* W: no. of GPU cores */
  for (j=0; j<N; j++) {
     j_tmp = (j + start ) % N;
     ... = a[j_tmp];
}}
```

Each GPU core accesses different data

- **<u>Key idea</u>: Ensure that the j loop is accessed in a different order for each GPU core**

Compiler Details



- **HeteroTBB pass:**
  - identify and lower Concord constructs
  - Handles virtual functions
- **Hetero pass:**
  - Check restrictions
  - Generates a list of kernels
- **HeteroGPU pass:**
  - Perform compiler optimizations
  - Generate OpenCL code
- **HeteroCPU pass:**
  - Generates x86 executable with embedded OpenCL code

Runtime Details

- OpenCL host program
  - Setup shared region and map to an OpenCL buffer
- Extract OpenCL code and JIT to GPU binary
  - Vendor OpenCL compiler
- Compile all the kernels at once
  - Cache the binary per function for future invocations
  - Amortizes the cost
- Allows heterogeneous CPU+GPU execution

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**

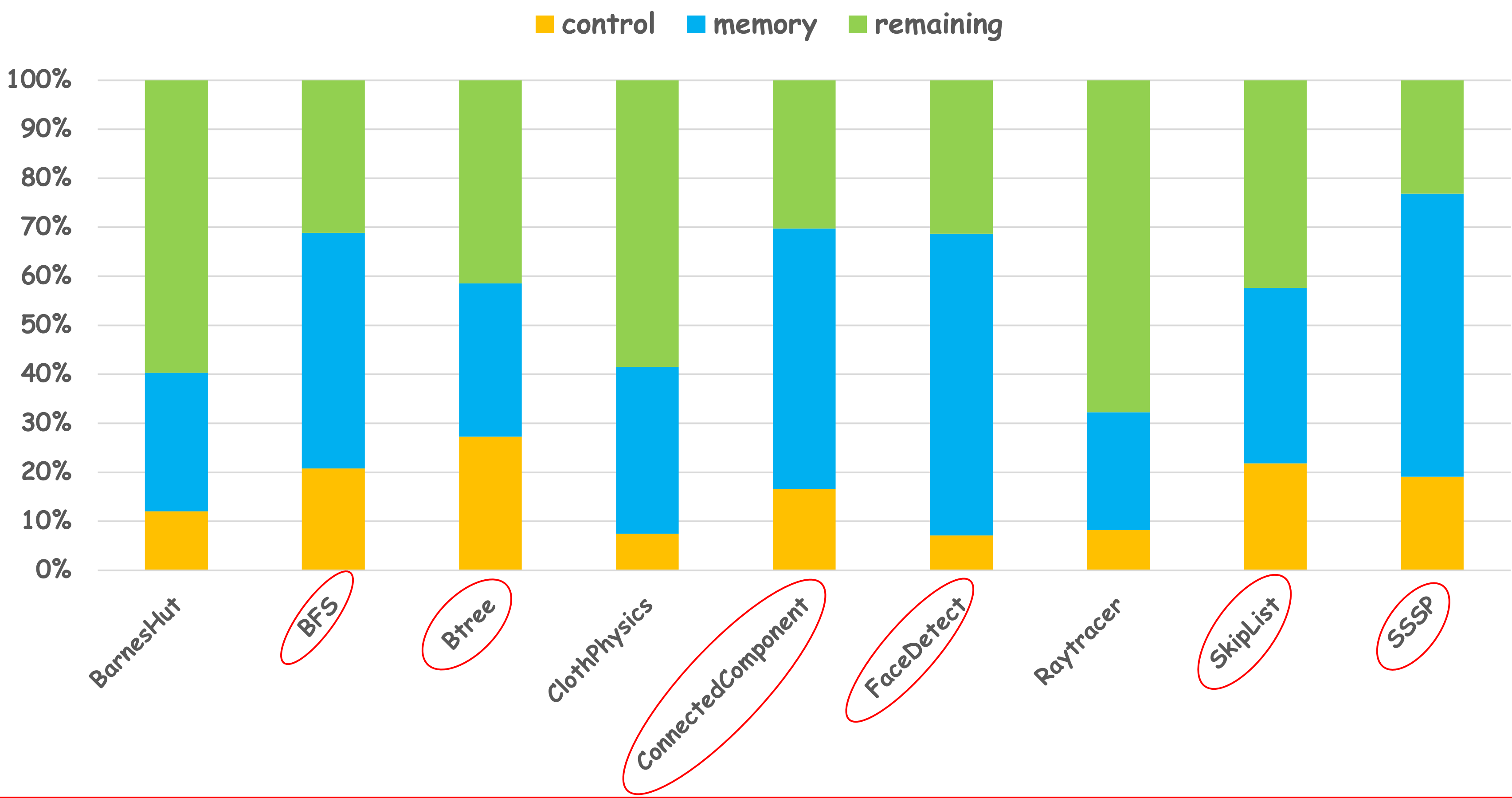
- **Comparison with multi-core CPU:**
  1. GPU-SPEEDUP: speedup using GPU execution
  2. GPU-ENERGY-SAVINGS: energy savings using GPU execution

Workloads

Benchmarks	Origin	Input	LoC	Device LoC	Data Structure	Parallel Construct
BarnesHut	In-house	1M bodies	828	105	Tree	Parallel_for
BFS	Galois	|V|=6.2M |E|=15M	866	19	Graph	Parallel_for
Btree	Rodinia	Command.txt	3111	84	Tree	Parallel_for
ClothPhysics	Intel	|V|=50K |E|=200K	9234	411	Graph	Parallel_reduce
ConnComp	Galois	|V|=6.2M |E|=15M	473	36	Graph	Parallel_for
FaceDetect	OpenCV	3000x2171	3691	378	Cascade	Parallel_for
Raytracer*	In-house	sphere=256, material=3, light=5	843	134	Graph	Parallel_for
Skip_List	In-house	50M keys	467	21	Linked-list	Parallel_for
SSSP	Galois	|V|=6.2M |E|=15M	1196	19	graph	Parallel_for

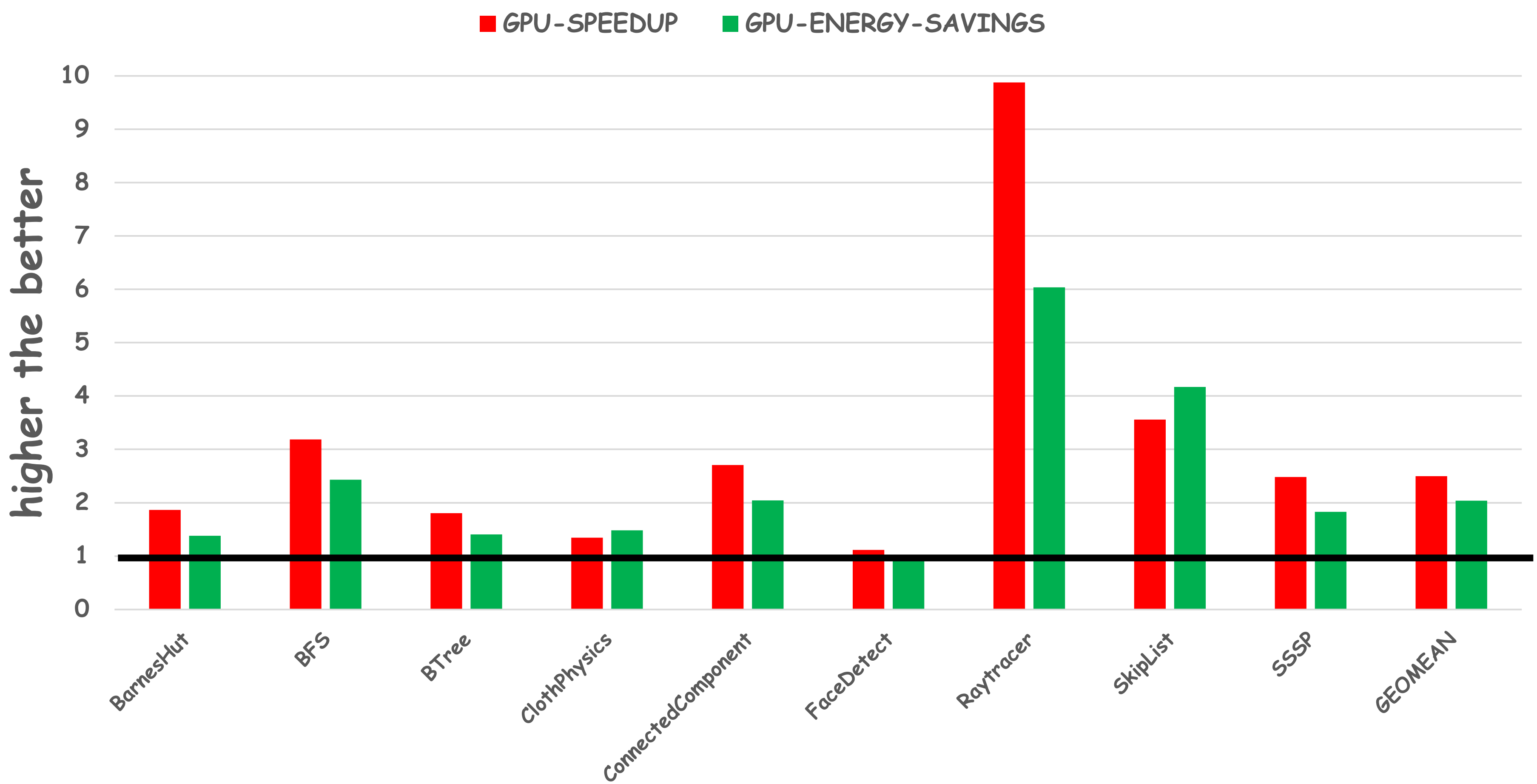
*uses virtual function

Dynamic estimates of irregularity



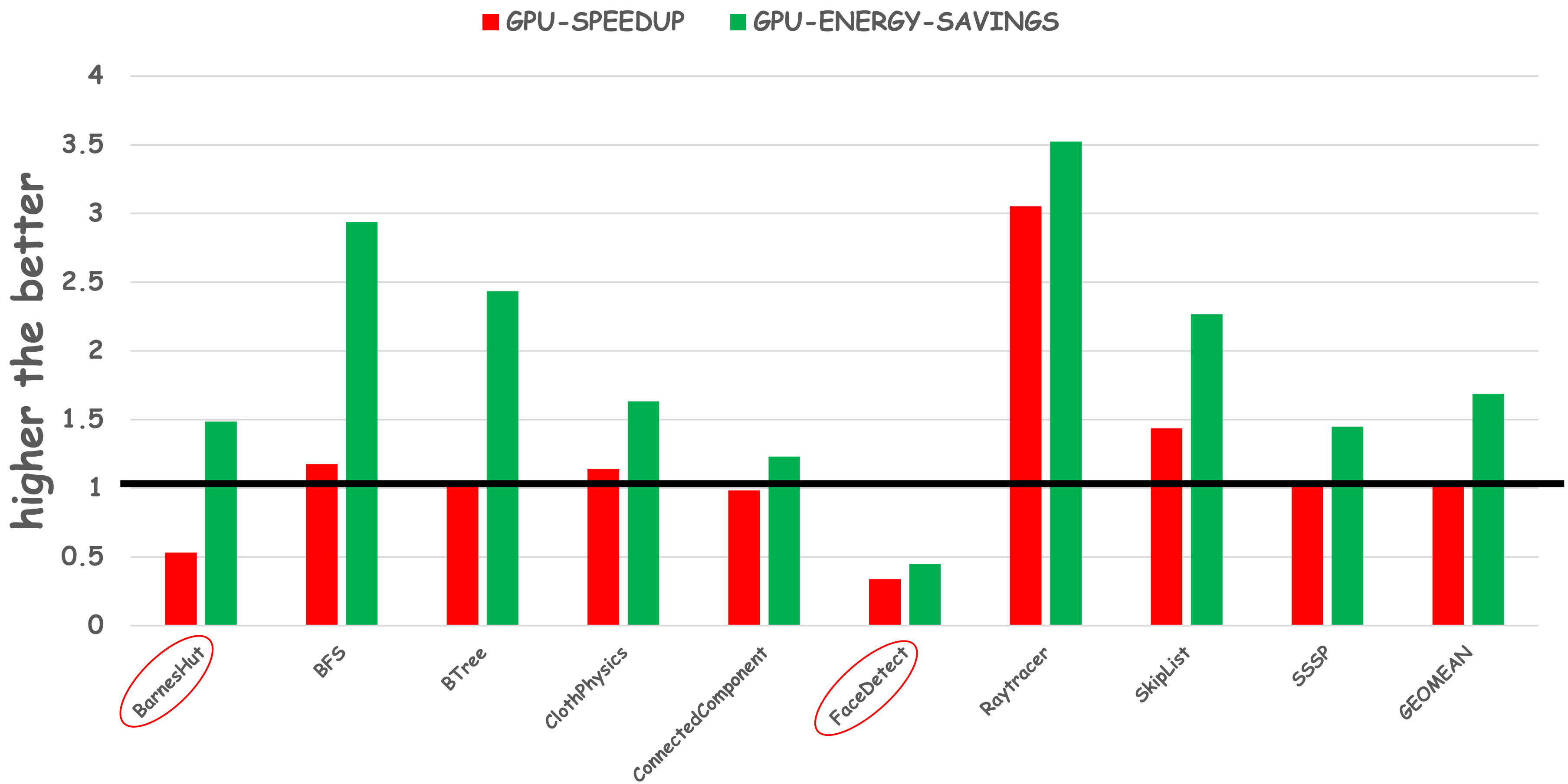
- BFS, Btree, ConnComp, FaceDetect, SkipList & SSSP exhibit a lot of irregularities (>50%)
- FaceDetect exhibits maximum percentage of memory irregularities

Ultrabook: Speedup & Energy savings compared to multicore CPU



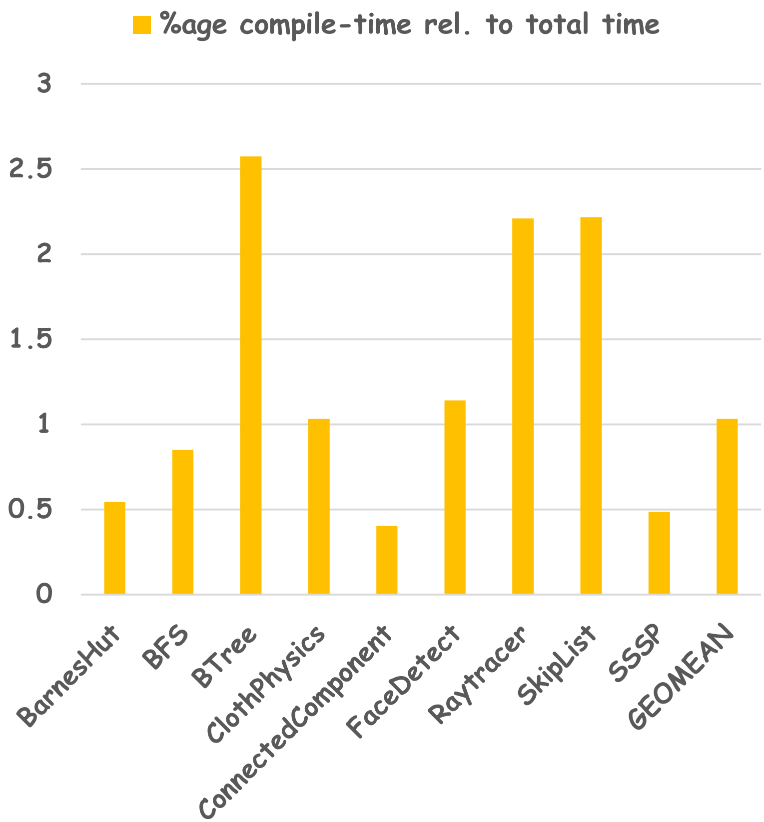
Average speedup of 2.5x and energy savings of 2x vs. multicore CPU

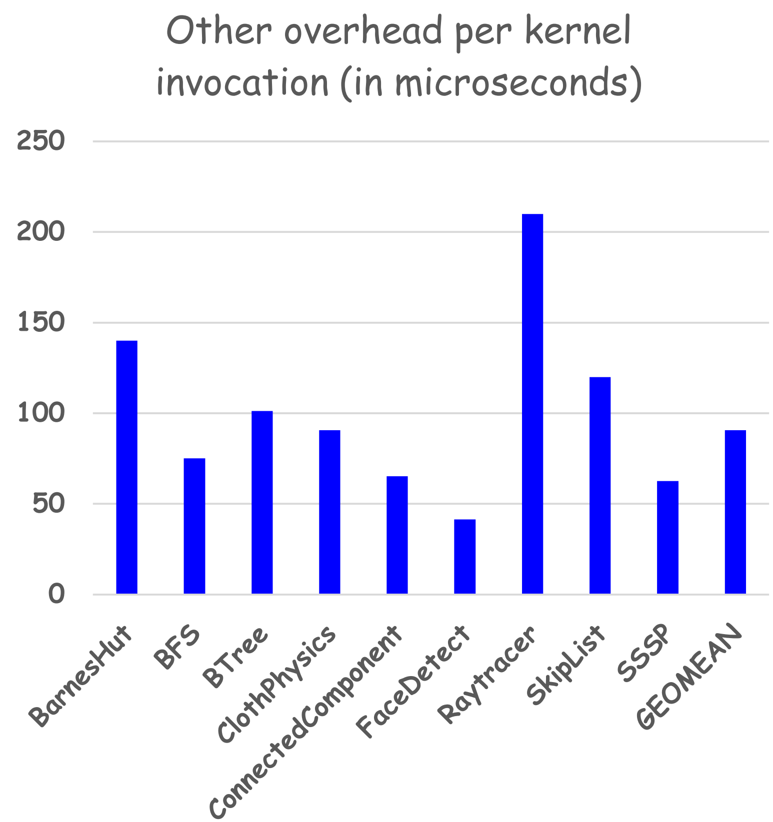
Desktop: Speedup & Energy savings compared to multicore CPU



Average speedup of 1.01x and energy savings of 1.7x vs. multicore CPU

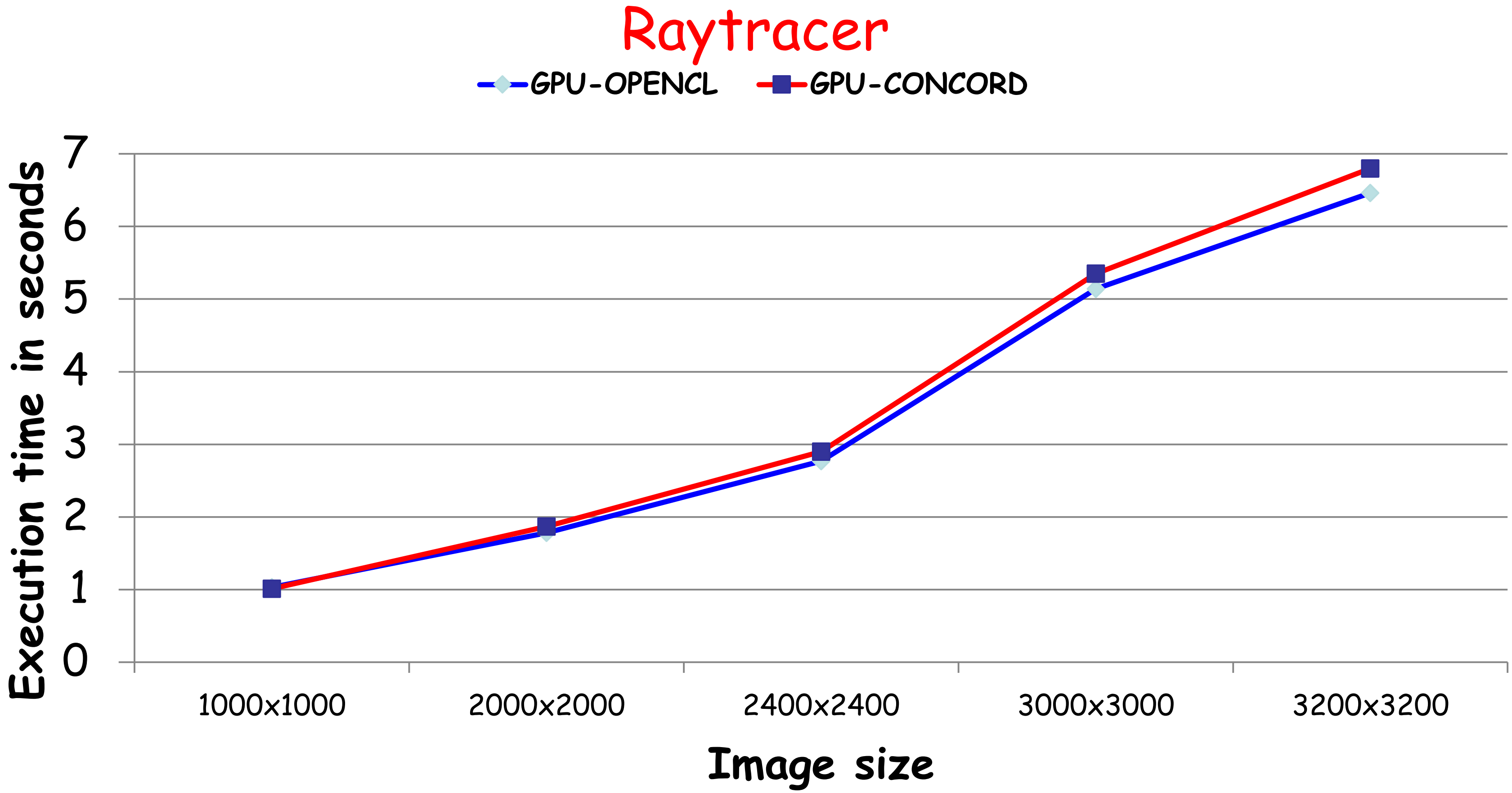
Overheads





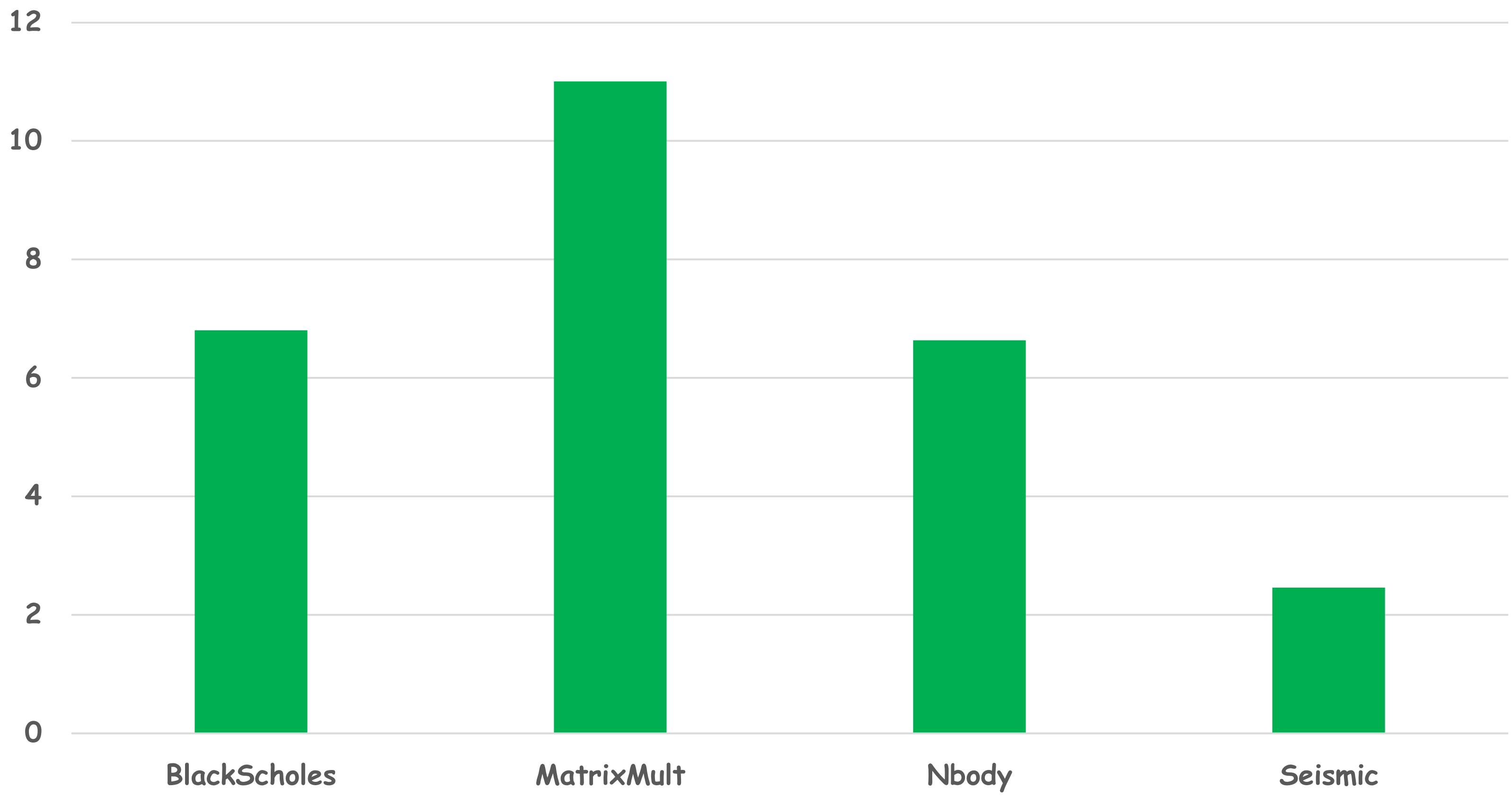
- **Compile-time is 1.03% of total execution time**
- **Other overheads (excluding compile-time) is ~90 microseconds**

Overhead of SW-based SVM implementation



SW-based SVM overhead is negligible for smaller images and is ~6% for the largest image

Regular Workloads on Desktop: Speedup compared to multi-core CPU



Comparison with Manual code

- **BTree from Rodinia: Concord takes 2.68s vs. 3.26s for hand-coded OpenCL on the Desktop Haswell system**

Conclusions & Future work

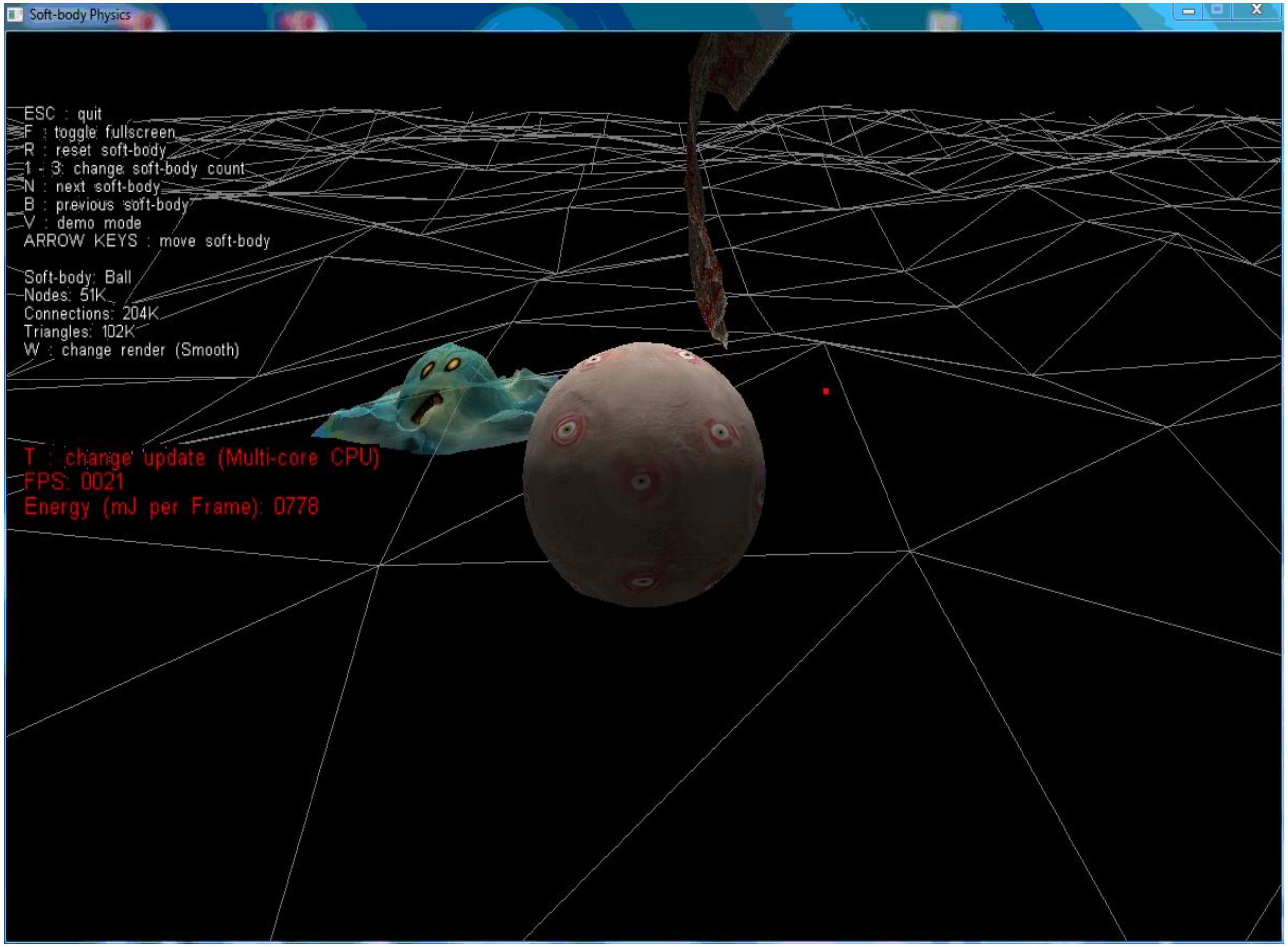
- **Runs out-of-the-box C++ applications on GPU**
- **Demonstrates that SVM is a key enabler in programmer productivity of heterogeneous systems**
- **Implements SVM in software with low-overhead**
- **Implements virtual functions and parallel reductions on GPU**
- **Saves energy of 2.04x on Ultrabook 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:

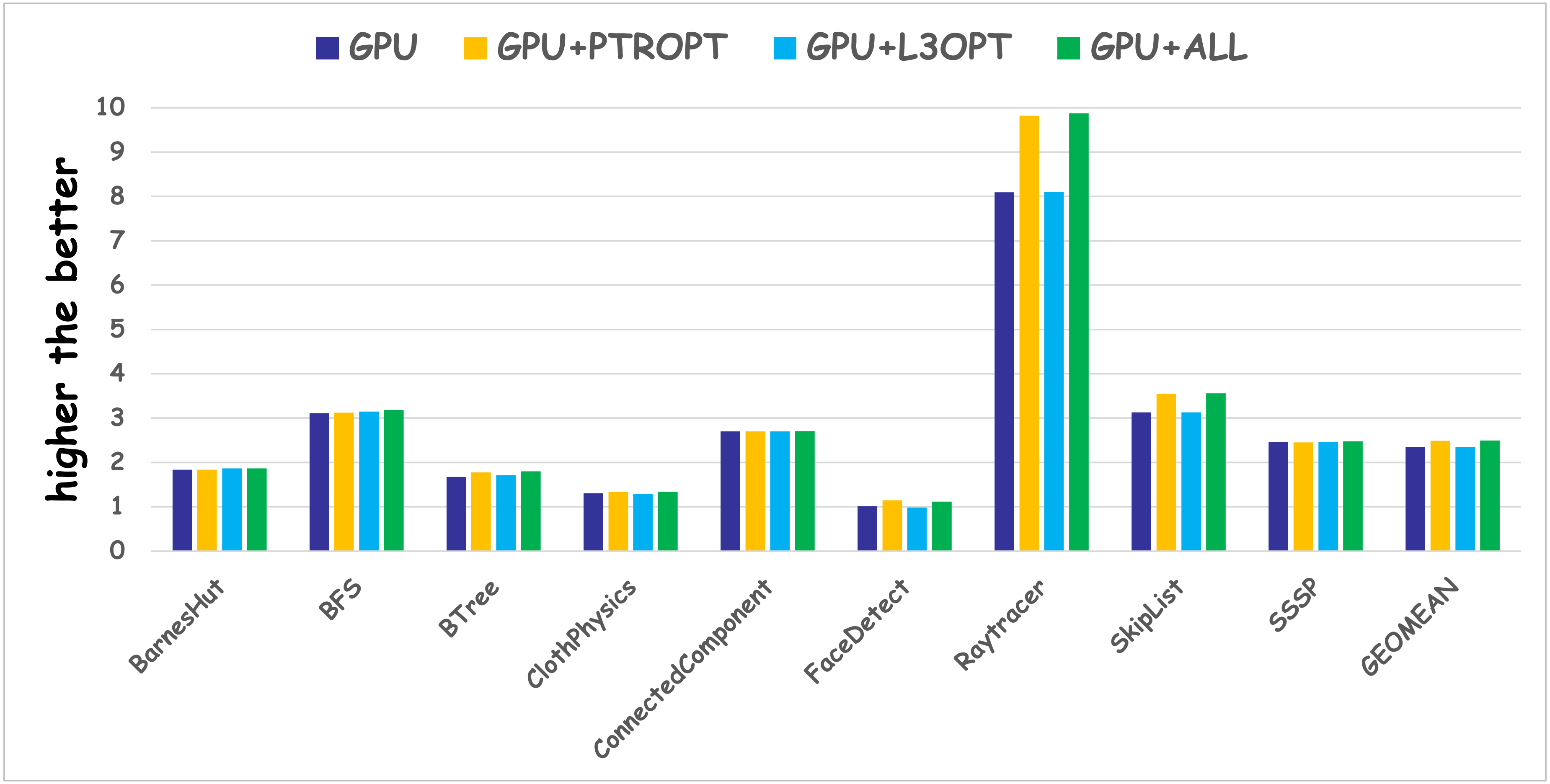
Questions?

Please try it out:
https://github.com/IntelLabs/iHRC/



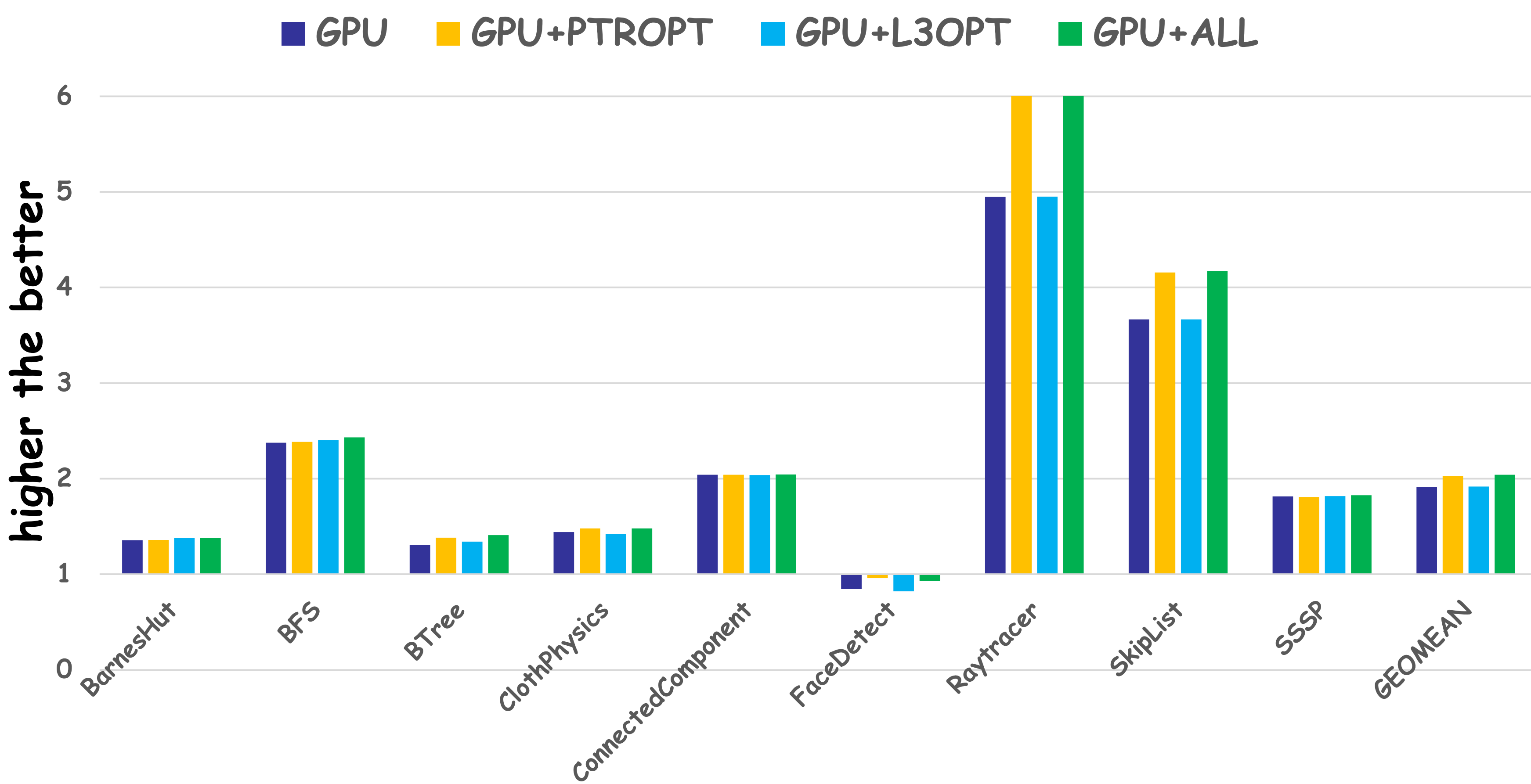
Backup

Ultrabook: Speedup compared to multicore CPU



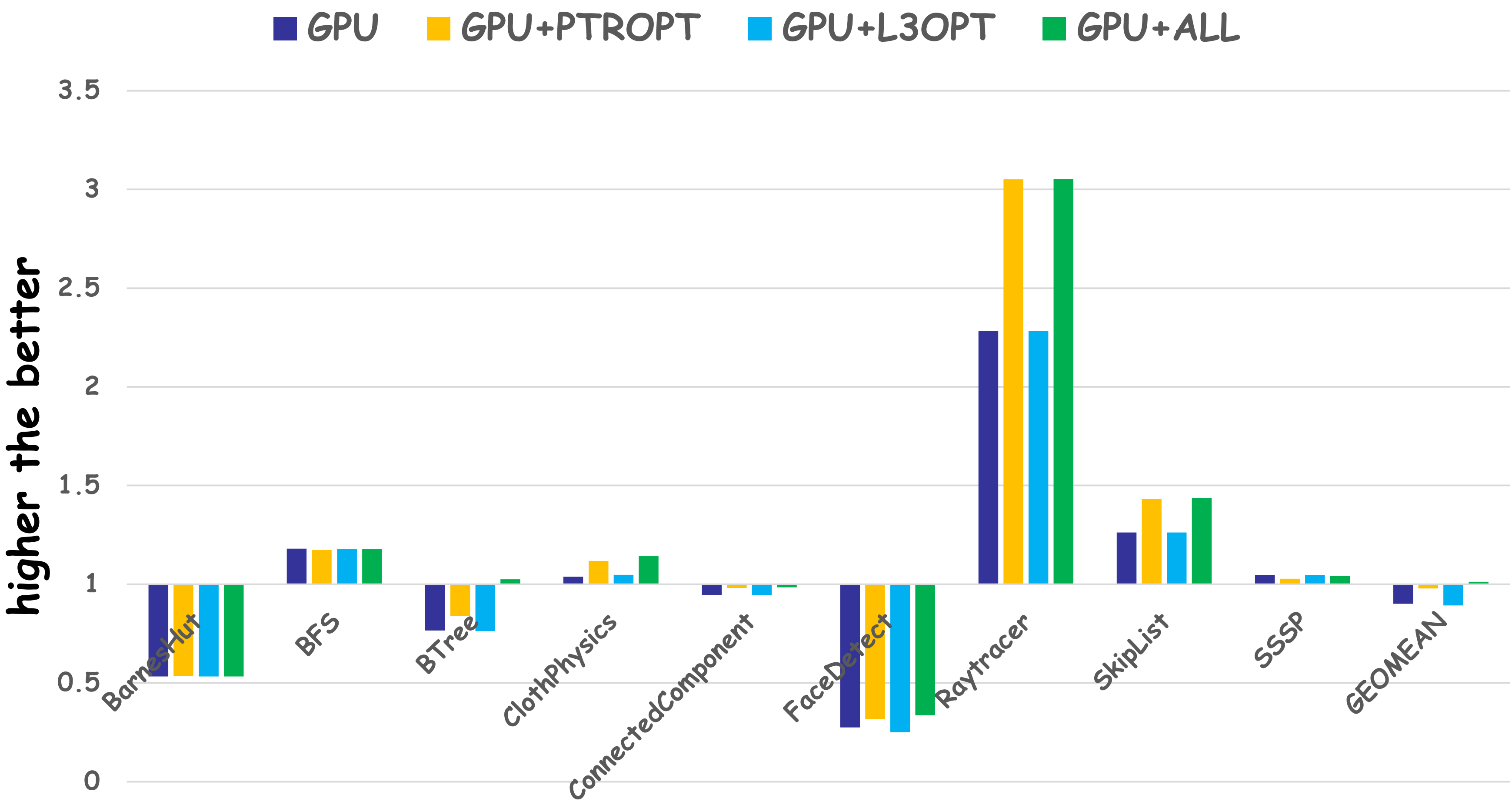
Average speedup of 2.5x vs. multicore CPU

Ultrabook: Energy savings compared to multi-core CPU

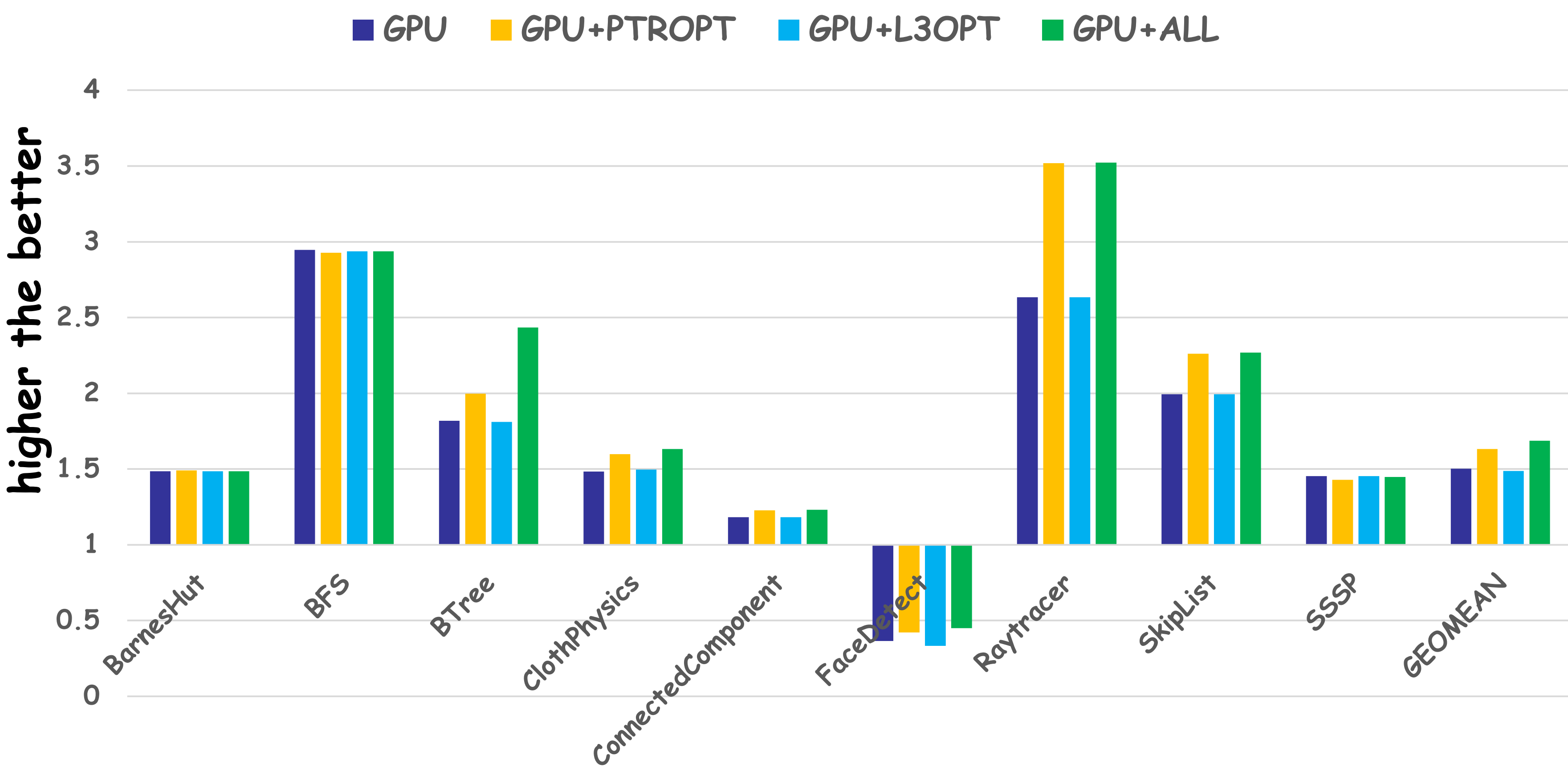


Average energy savings of 2.04x vs. multicore CPU

Desktop: Speedup compared to multi-core CPU



Desktop: Energy savings compared to multi-core CPU



Average energy savings of 1.7x vs. multicore CPU