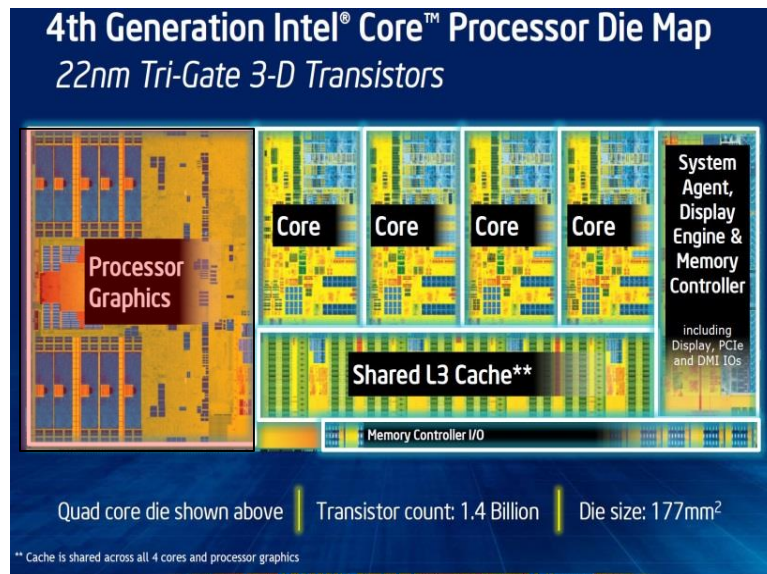

Concord: Homogeneous Programming for Heterogeneous Architectures

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

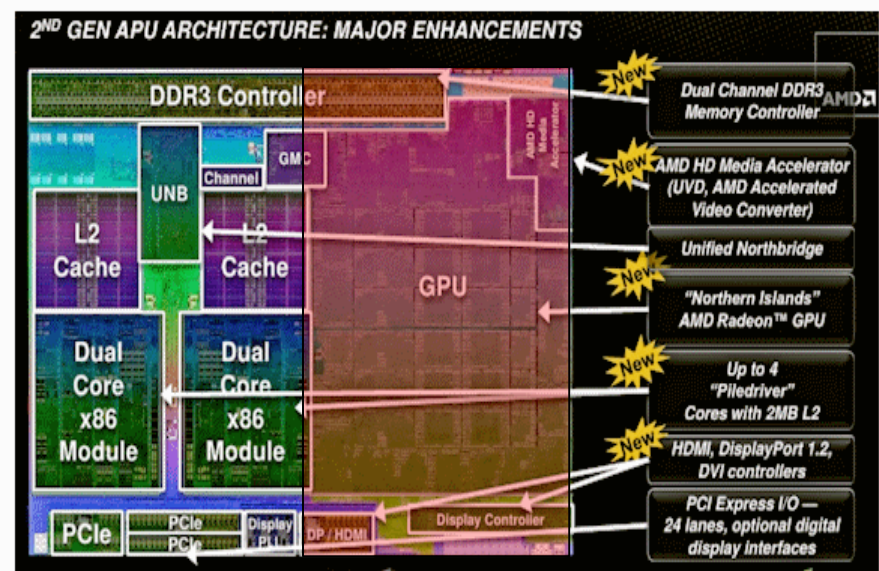
Heterogeneous Platforms

- **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.

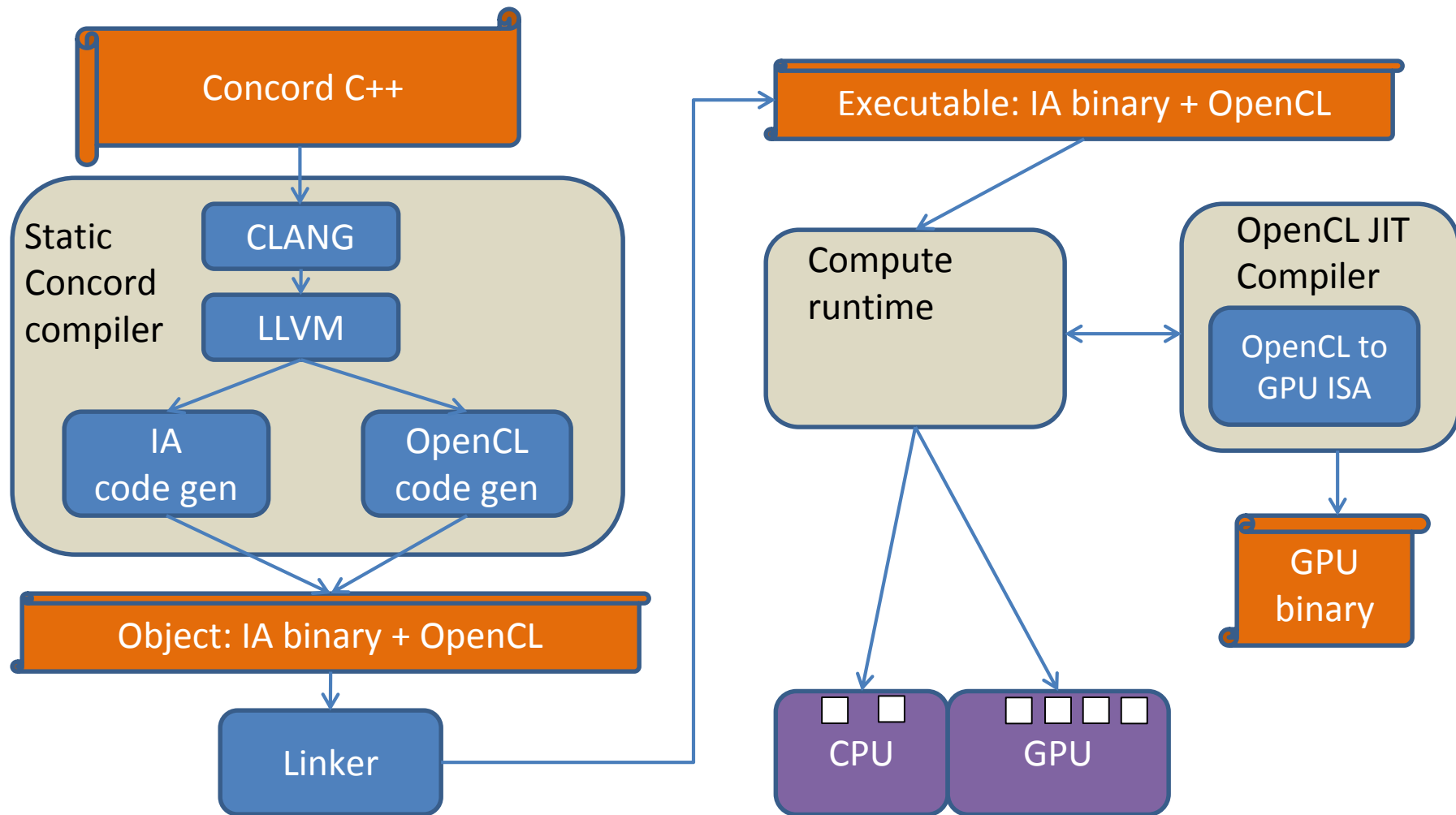
Widen the set of applications that target GPUs

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);
```

Concord Version

Run on CPU
or GPU

Minimal differences between two versions

Example: parallel_for_hetero

```
class Foo {  
    float *A, *B, *C;  
    public:  
        Foo(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];  
        }  
};  
  
.....  
Foo *f = new Foo(A,B,C);  
parallel_for_hetero (1024, *f, GPU); // Data parallel operation for GPU
```


Example: parallel_reduce_hetero

```
class Bar {  
    float *A, sum;  
    public:  
        Bar(float *a_): A(a_), sum(0.0f) { }  
        void operator()(int i) { // execute in parallel  
            sum = f(A[i]); // compute local sum  
        }  
        void join(Bar &rhs) {  
            sum += rhs.sum; // perform reduction  
        }  
};  
  
.....  
Bar *b = new Bar(A);  
parallel_reduce_hetero (1024, *b, GPU); // Data parallel reduction on GPU
```

Restrictions

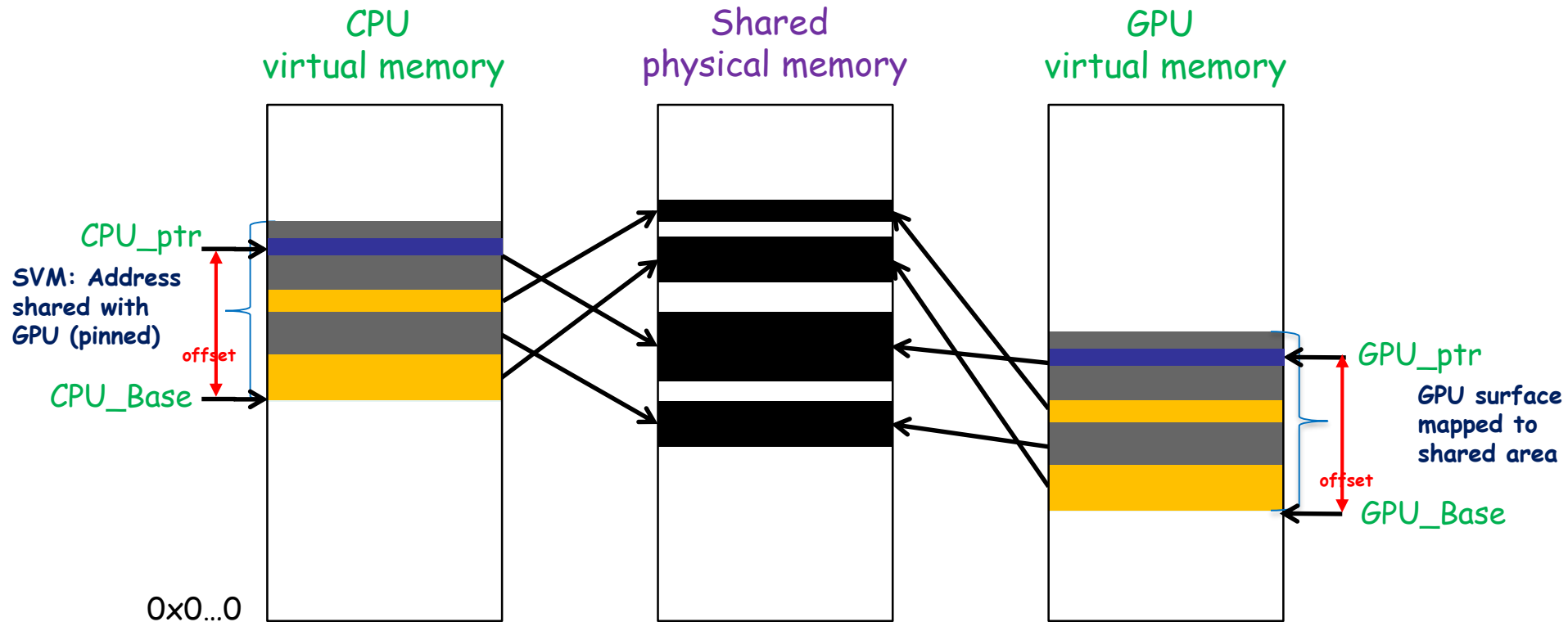
- No guarantee that the parallel loop iterations will be executed in parallel
- No ordering among different parallel iterations
 - Floating-point determinism is not guaranteed
- Features not yet supported on the GPU
 - Recursion (except tail recursion which can be converted to loop)
 - Exception
 - Taking address of local variable
 - Memory allocation and de-allocation
 - Function calls via function pointers (virtual functions are handled)

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



$$\text{GPU_ptr} = \text{GPU_Base} + \text{CPU_ptr} - \text{CPU_Base}$$

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++

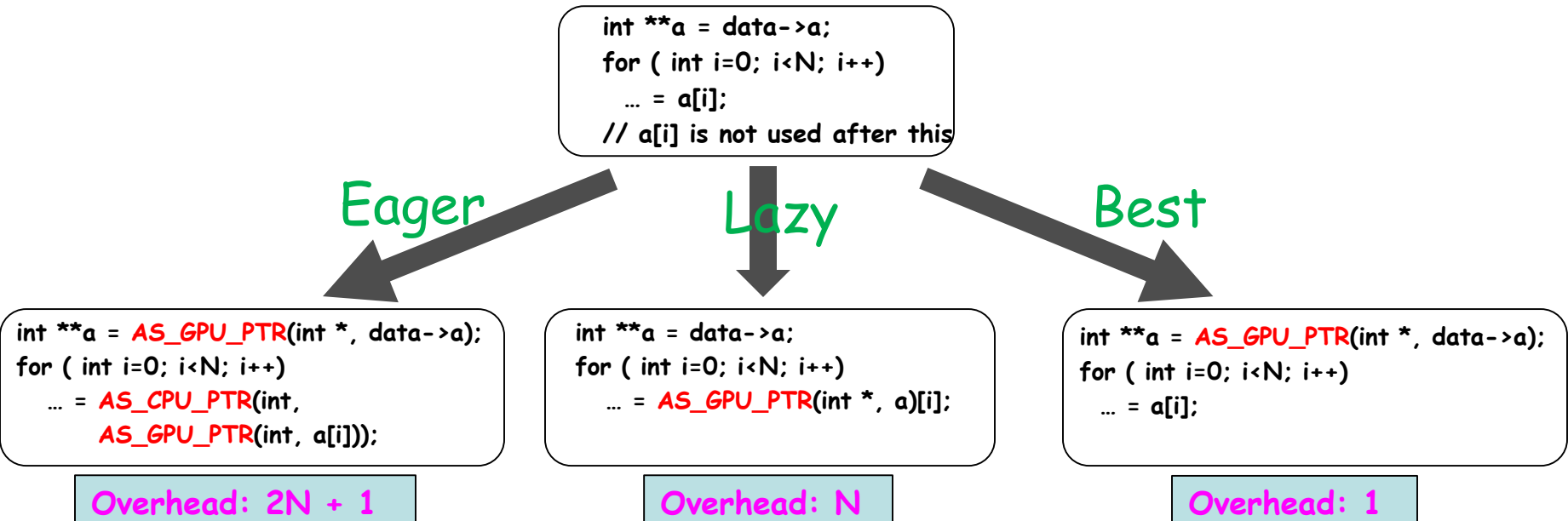


```
//__global char * svm_const = (GPU_Base - CPU_Base);  
  
#define AS_GPU_PTR(T,p) (__global T *) (svm_const + p)  
  
__kernel void openc1_operator (  
    __global char *svm_const,  
    unsigned long B_ptr) {  
  
    AS_GPU_PTR(LinkedList, list)->key...  
}
```

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



- **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:

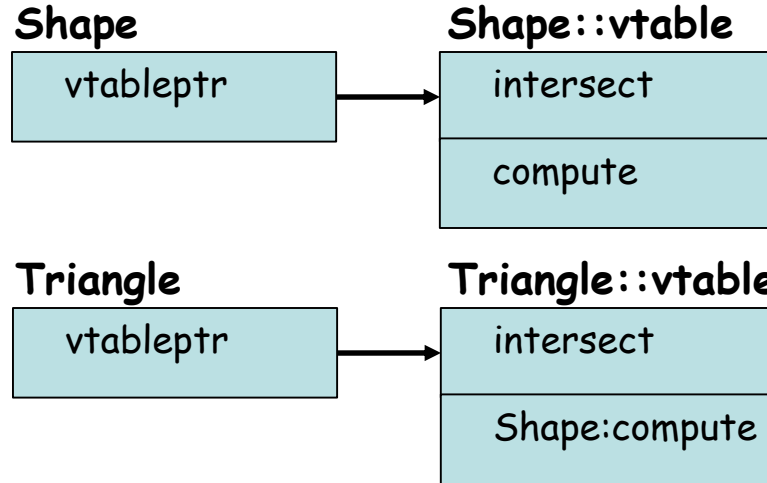
```
class Shape {  
    virtual void intersect() {...}  
    virtual void compute() {...}  
};  
class Triangle : Shape {  
    virtual void intersect() {...}  
};
```

Virtual Function call:

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

Original code

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();  
}
```

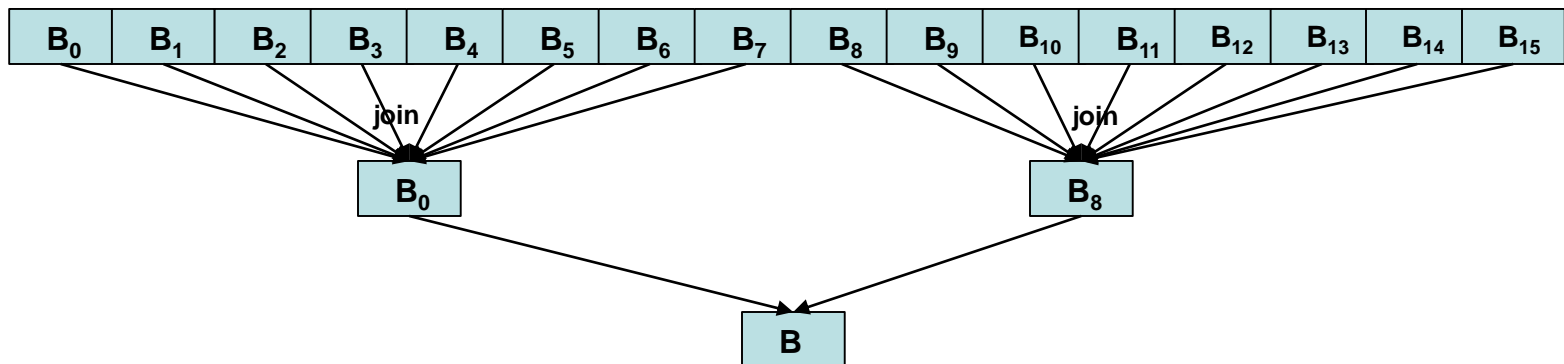
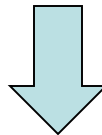
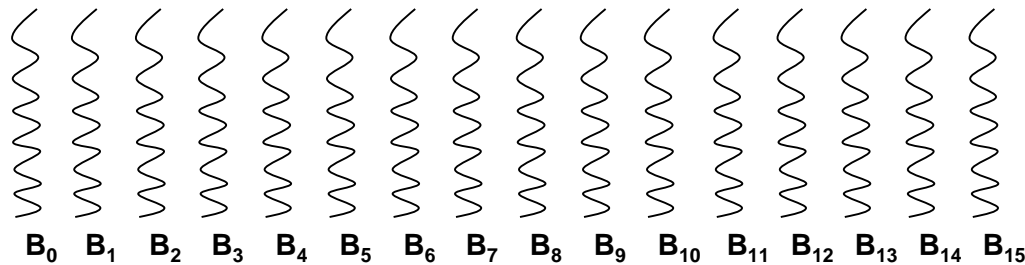
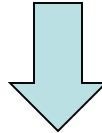
Generated code

- 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, GPU)`

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



Compiler Optimization for Cache Contention

- Integrated GPUs often 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];  
    }  
}
```

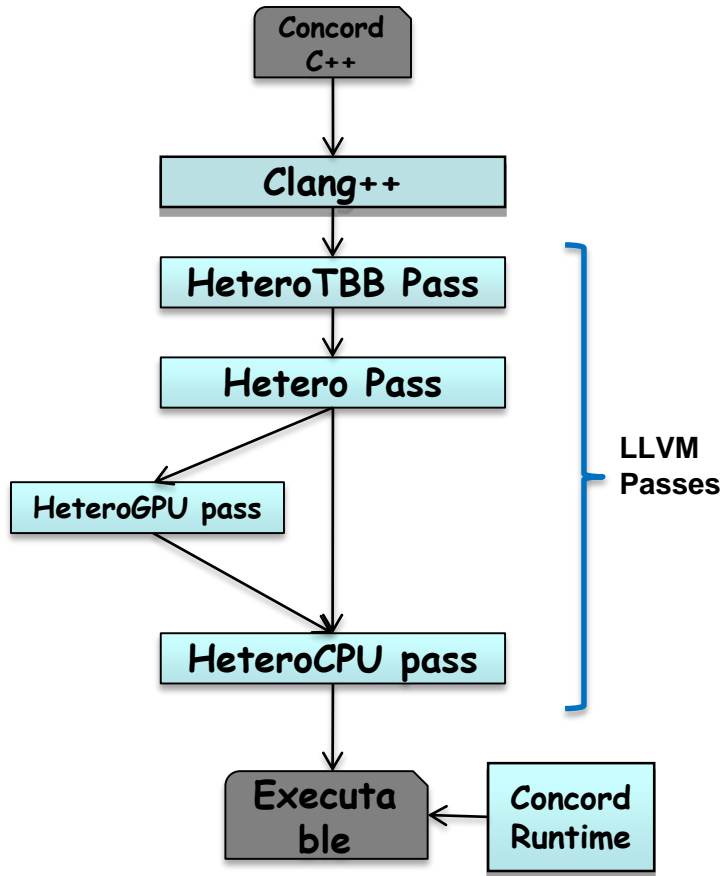
Each GPU core accesses different data

- Key idea: Ensure that the j loop is accessed in a different order for each GPU core

Using GPU Memory hierarchy

- Stack allocated objects in C++ are promoted to OpenCL private memory
- Reductions are performed in OpenCL local shared memory
- Automatic generation of local memory code for regular applications (work-in-progress)

Compiler Details



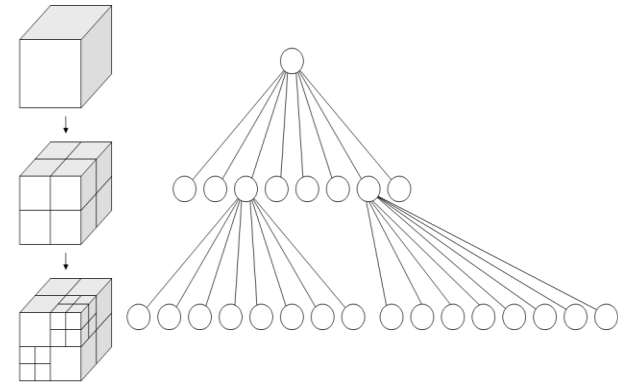
- **HeteroTBB pass:**
 - identify and lower Concord constructs
 - Handles virtual functions
- **Hetero pass:**
 - Check restrictions
 - Generates a list of kernels
- **HeteroGPU pass:**
 - Perform compiler optimizations
 - Generate OpenCL code
- **HeteroCPU pass:**
 - Generates x86 executable with embedded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

Case Study: Barnes-Hut

- An efficient algorithm for the N-body problem
 - Approximates far away bodies
- Algorithm:
 - Build an oct-tree representing positions of bodies
 - Update the centers of masses for all subtrees
 - Sort the bodies based on relative positions
 - Calculate gravitational forces between bodies (offload to GPU)
 - Update positions and velocities
- Takes advantage of (shared) pointers



Barnes-Hut CUDA Kernel

```
void ForceCalculationKernel()
{
    if (0 == threadIdx.x) {
        tmp = radiusd;
        dq[0] = tmp * tmp * itolsqd;
        for (i = 1; i < maxdepthd; i++) {
            dq[i] = dq[i - 1] * 0.25f;
            dq[i - 1] += epssqd;
        }
        dq[i - 1] += epssqd;

        if (maxdepthd > MAXDEPTH) {
            *errd = maxdepthd;
        }
    }
    __syncthreads();

    if (maxdepthd <= MAXDEPTH) {
        base = threadIdx.x / WARPSIZE;
        sbase = base * WARPSIZE;
        j = base * MAXDEPTH;
        diff = threadIdx.x - sbase;
        if (diff < MAXDEPTH) {
            dq[diff+j] = dq[diff];
        }
        __syncthreads();

        // iterate over all bodies assigned to thread
        for (k = threadIdx.x + blockDim.x * blockDim.x;
             k < nbodiesd; k += blockDim.x * gridDim.x) {
            i = sortd[k]; // get permuted/sorted index
            // cache position info
            px = posxd[i];
            py = posyd[i];
            pz = poszd[i];

            ax = 0.0f;
            ay = 0.0f;
            az = 0.0f;

            // initialize iteration stack, i.e., push root
            node onto stack
            depth = j;
            if (sbase == threadIdx.x) {
                node[j] = nnodesd;
                pos[j] = 0;
            }

            while (depth >= j) {
                // stack is not empty
                while ((t = pos[depth]) < 8) {
                    // node on top of stack has more children
                    to process
                    n = childd[node[depth]*8+t]; // load child
                    pointer
                    if (sbase == threadIdx.x) {
                        // I'm the first thread in the warp
                        pos[depth] = t + 1;
                    }
                }

                if (n >= 0) {
                    dx = posxd[n] - px;
                    dy = posyd[n] - py;
                    dz = poszd[n] - pz;
                    tmp = dx*dx + (dy*dy + (dz*dz +
                        epssqd)); // compute distance squared
                                // plus softening)
                    if ((n < nbodiesd) || __all(tmp >=
                        dq[depth])) { // check if all threads
                        agree that cell is far enough away (or is
                        a body)
                            tmp = rsqrtf(tmp); // compute
                                distance
                            tmp = massd[n] * tmp * tmp *
                                tmp;

                            ax += dx * tmp;
                            ay += dy * tmp;
                            az += dz * tmp;
                        } else {
                            // push cell onto stack
                            depth++;
                            if (sbase == threadIdx.x) {
                                node[depth] = n;
                                pos[depth] = 0;
                            }
                        }
                    } else {
                        depth = max(j, depth - 1); //
                        early out because all remaining children
                        are also zero
                    }
                }
                depth--; // done with this level
            }

            if (stepd > 0) {
                // update
                velocity
                velxd[i] += (ax -
                    accxd[i]) * dthfd;
                velyd[i] += (ay -
                    accyd[i]) * dthfd;
                velzd[i] += (az -
                    acczd[i]) * dthfd;
            }

            // save computed
            acceleration
            accxd[i] = ax;
            accyd[i] = ay;
            acczd[i] = az;
        }
    }
}
```

Source: <http://www.gpucomputing.net/?q=node/1314>

~100 Lines of CUDA Code with optimization, hard to read and maintain

Barnes-Hut Concord C++ Kernel

```
1. void update (BH_Tree **stack, Body *body) {
2.     while(!stack.empty()) {
3.         Octree *tree = stack.top();
4.         stack.pop();
5.         Octree **children = ((OctreeInternal*)tree)->child;
6.         for(int i=0;i<8;i++) {
7.             Octree *child = children[i];
8.             if (!child) continue;
9.             if (child->nodeType == LEAF || body->pos.distance(child->pos)
10.                * THETA > child->box.size()) {
11.                 computeForce(body, child);
12.             } else {
13.                 stack.push(child);
14.             }
15.         }
16. }
```

- *distance* is 5 lines. *computeForce* is 9 lines. *push* is 2 lines and *pop* is 1 line
- Total 33 lines of code
- No extra host code for device malloc and data copy

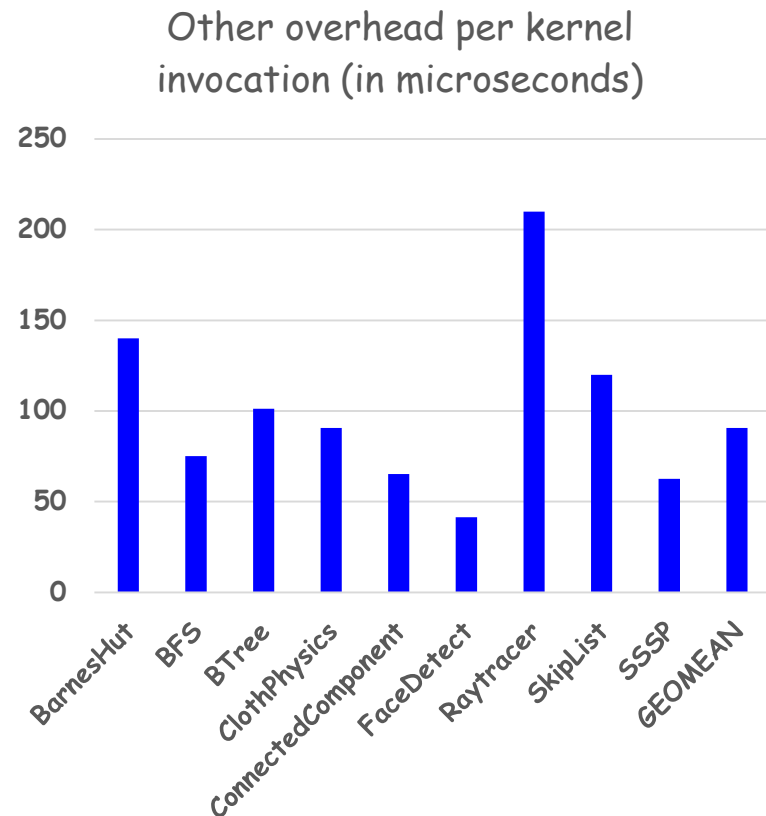
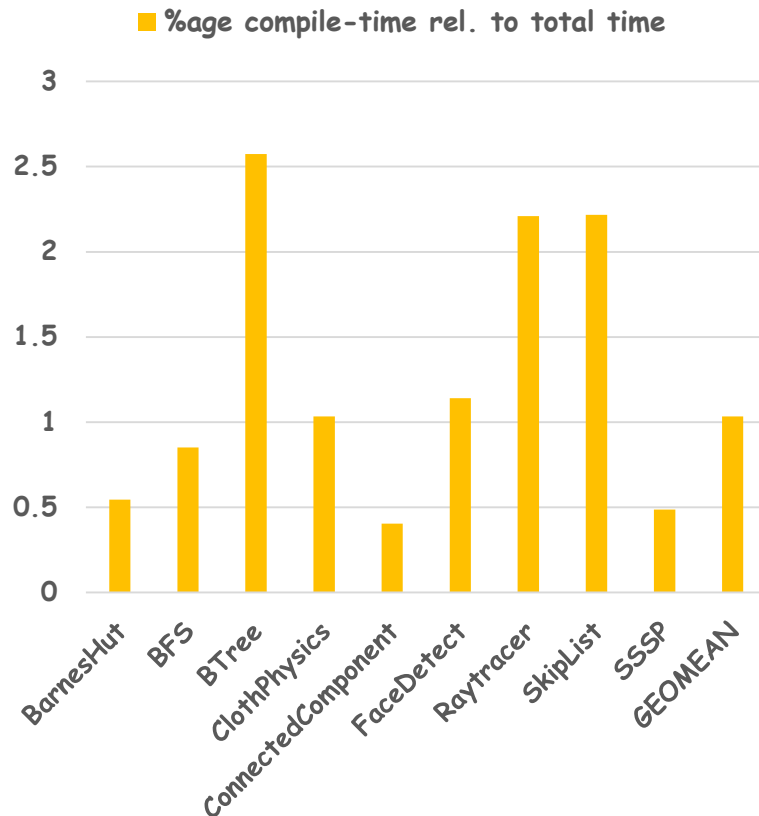
Experimental setup

- 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

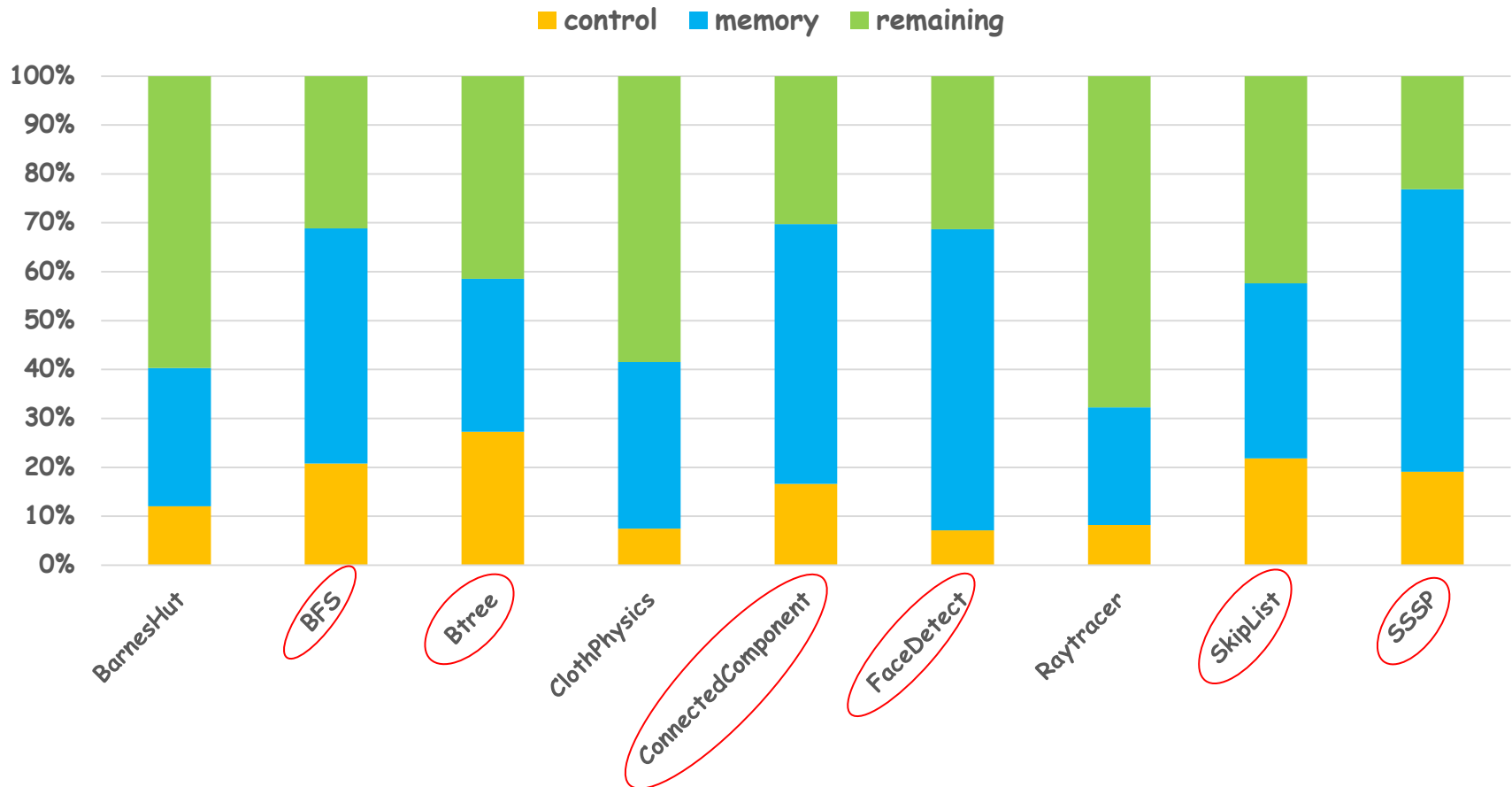
Benchmarks	Origin	Input	LoC	Device LoC	Data Structure	Parallel Construct
BarnesHut	In-house	1M bodies	828	105	Tree	Parallel_for
BFS	Galois	V =6.2M E =15M	866	19	Graph	Parallel_for
Btree	Rodinia	Command.txt	3111	84	Tree	Parallel_for
ClothPhysics	Intel	V =50K E =200K	9234	411	Graph	Parallel_reduce
ConnComp	Galois	V =6.2M E =15M	473	36	Graph	Parallel_for
FaceDetect	OpenCV	3000x2171	3691	378	Cascade	Parallel_for
Raytracer* *uses virtual function	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

Overheads (on desktop system)



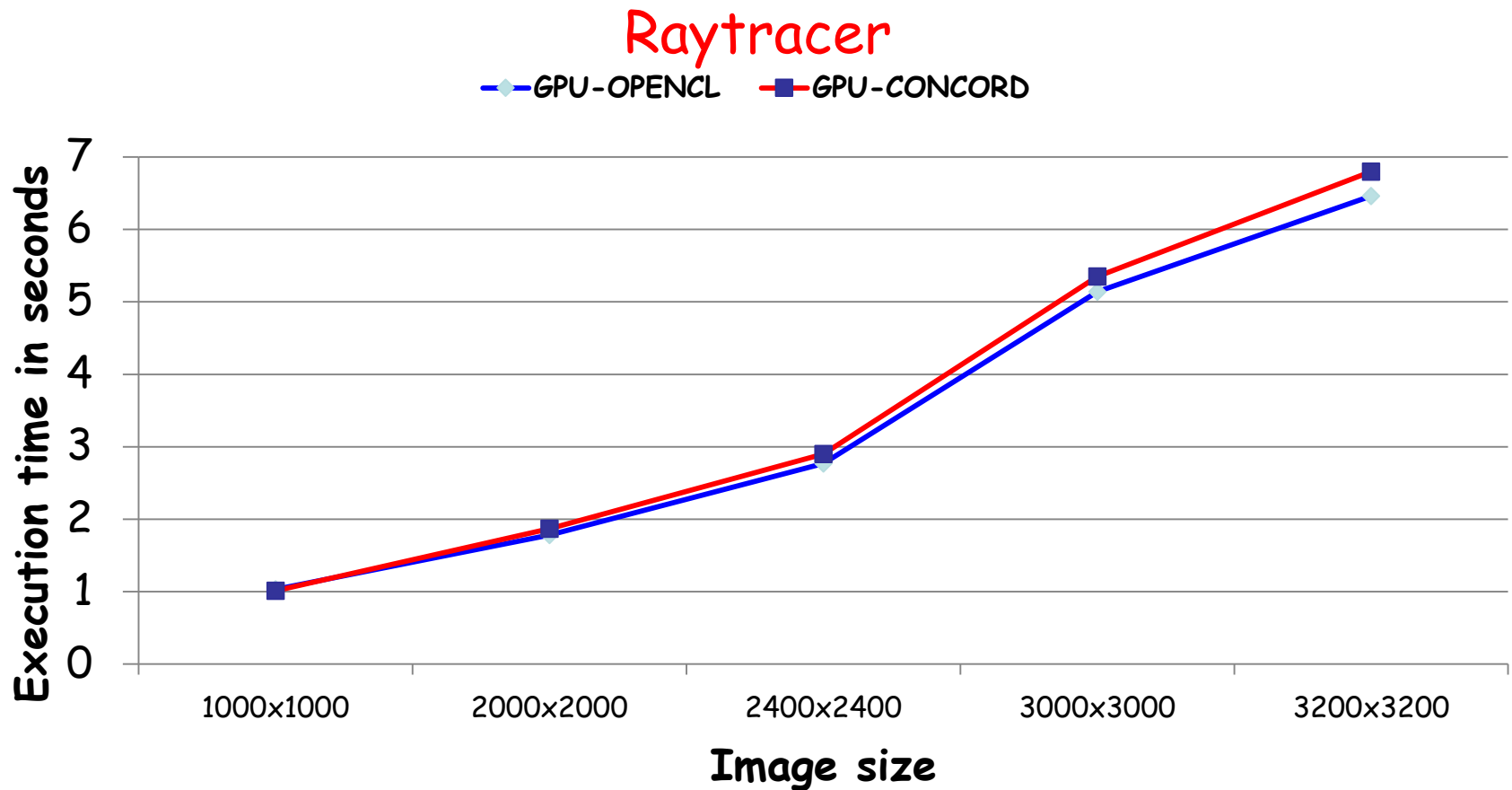
- Compile-time is 1.03% of total execution time
- Other overheads (excluding compile-time) is ~90 microseconds

Dynamic estimates of irregularity



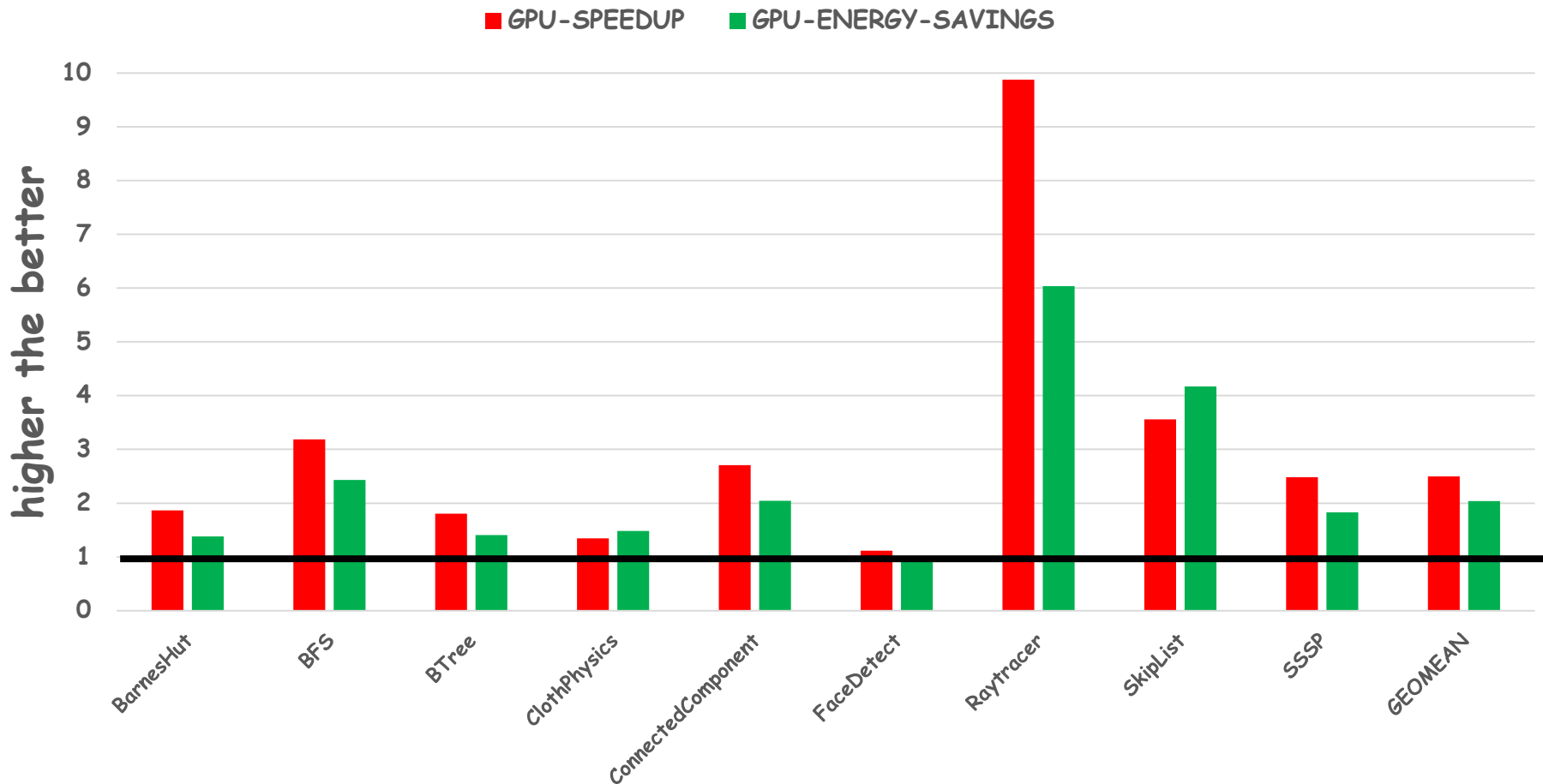
- BFS, Btree, ConnComp, FaceDetect, SkipList & SSSP exhibit a lot of irregularities (>50%)
- FaceDetect exhibits maximum percentage of memory irregularities

Overhead of SW-based SVM implementation



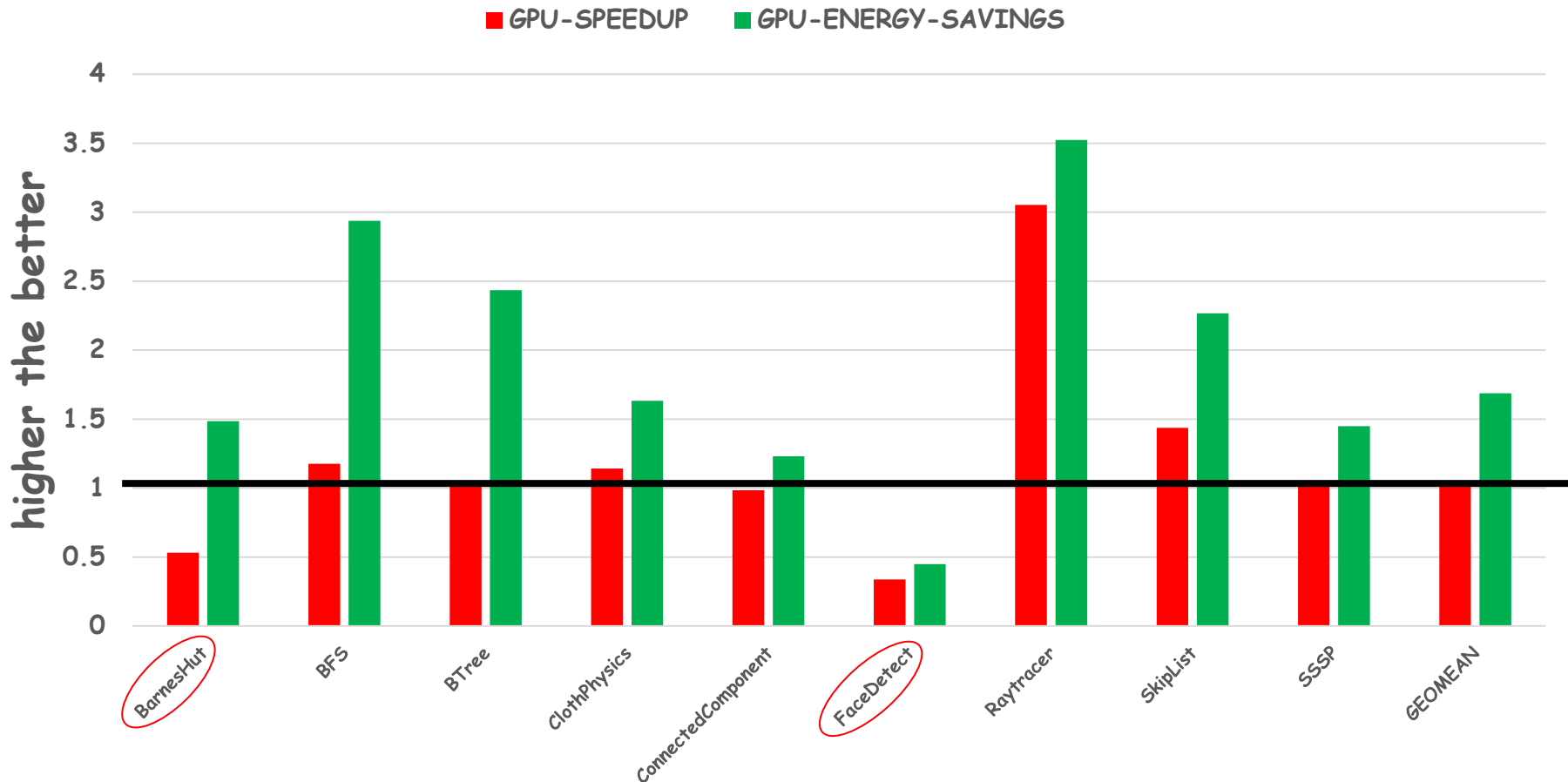
SW-based SVM overhead is negligible for smaller images and is ~6% for the largest image

Ultrabook: Speedup & Energy savings compared to multicore CPU



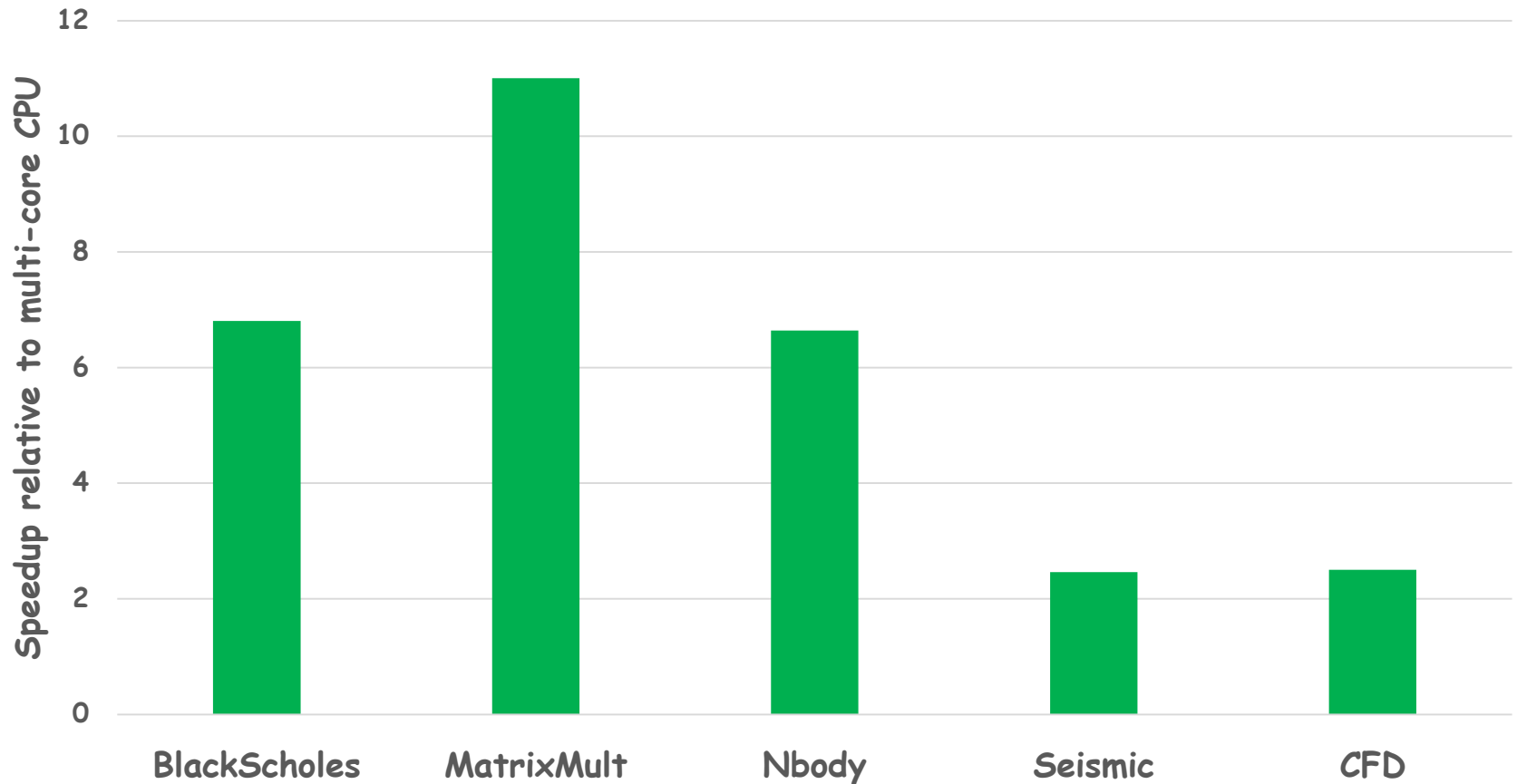
Average speedup of 2.5x and energy savings of 2x vs. multicore CPU

Desktop: Speedup & Energy savings compared to multicore CPU



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

Regular Workloads on Quad-core desktop: Speedup compared to multi-core CPU



Automatic local memory code generation can further boost performance

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

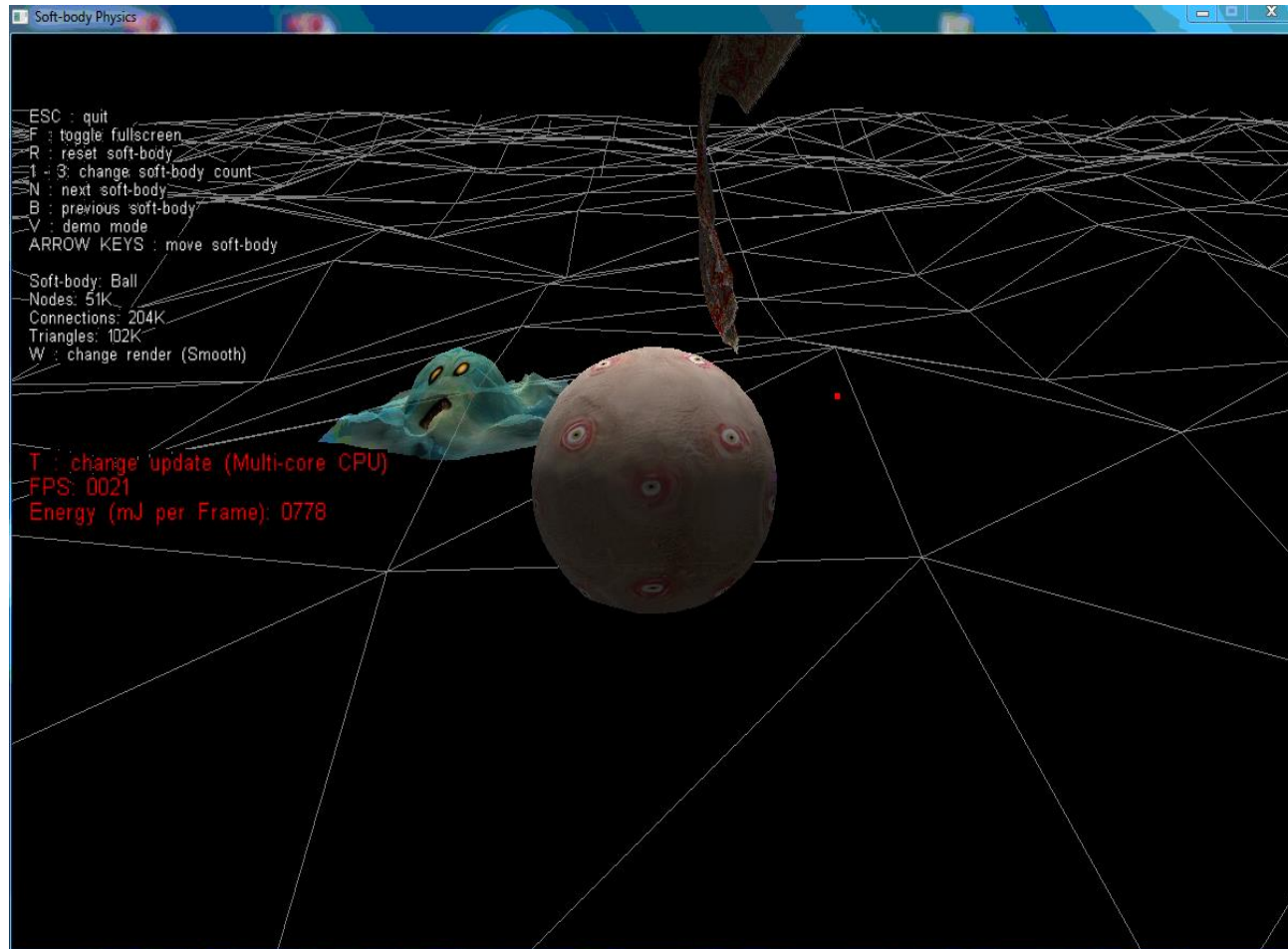
- Runs out-of-the-box C++ applications on GPU
 - No new language invention
- Demonstrates that SVM is a key enabler in programmer productivity of heterogeneous systems
- Implements SVM in software with low-overhead
- Implements virtual functions and parallel reductions on GPU
- Saves energy of 2.04x 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?

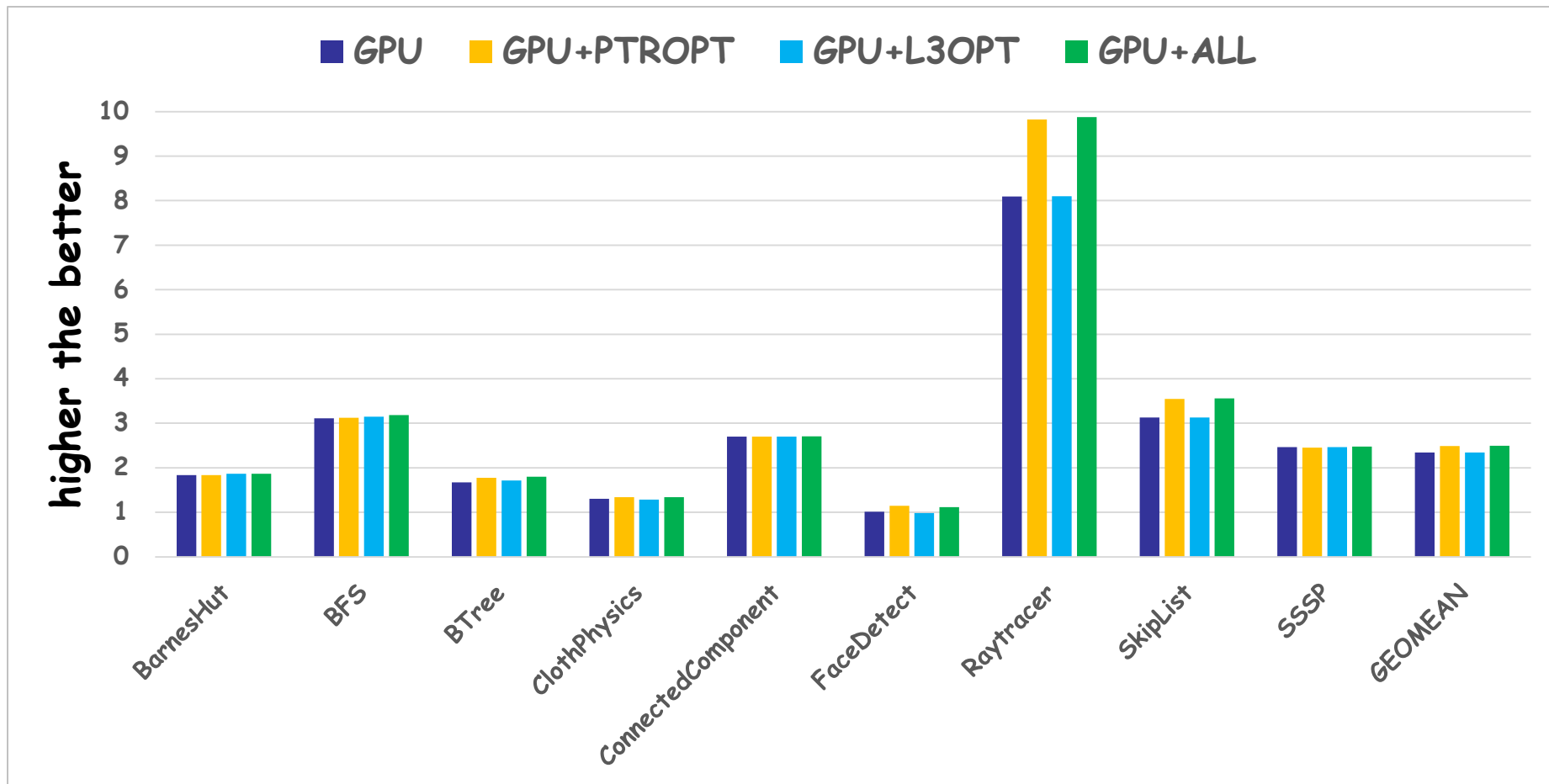
Please try it out:

<https://github.com/IntelLabs/iHRC/>



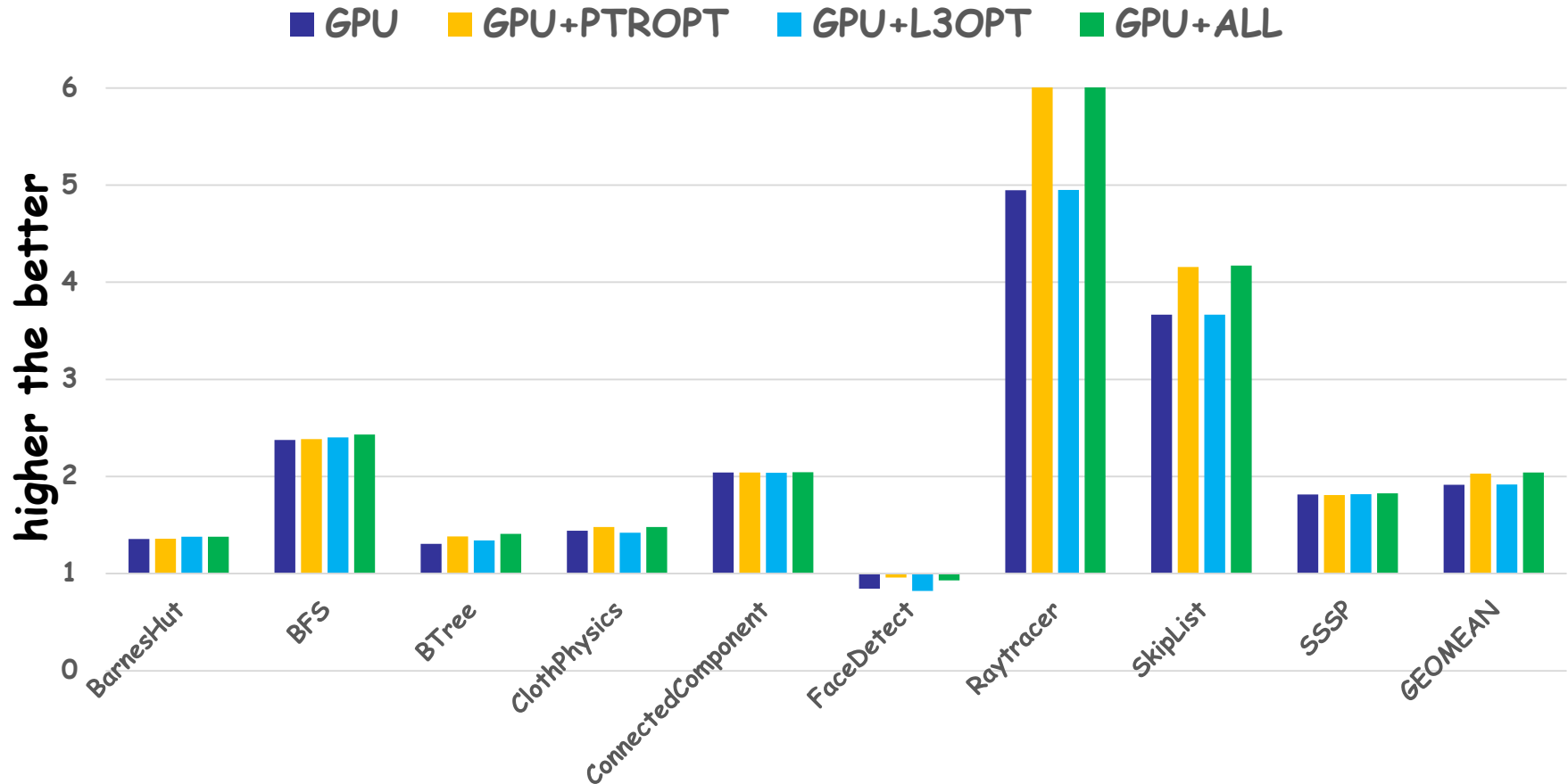
Backup

Ultrabook: Speedup compared to multicore CPU



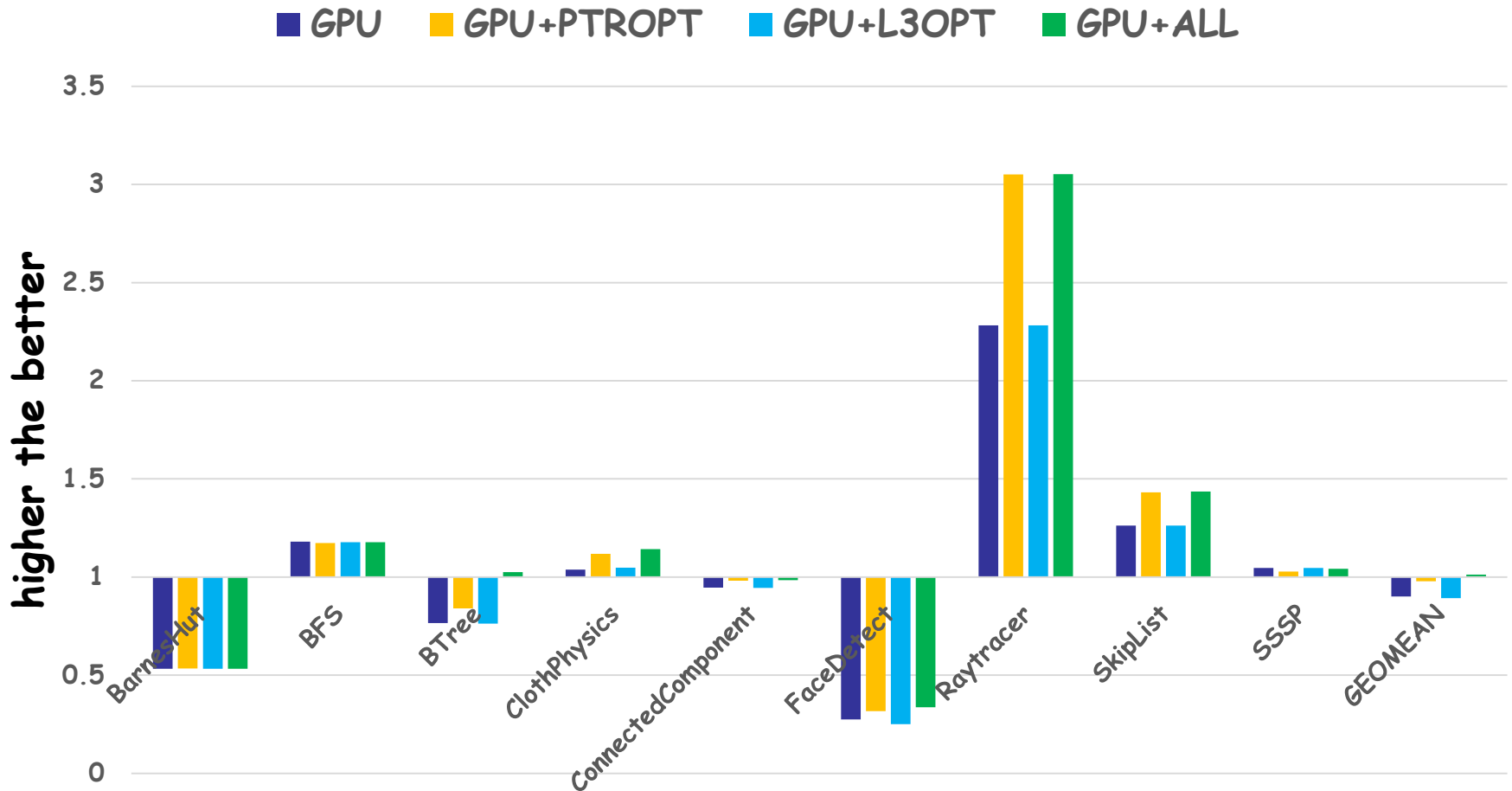
Average speedup of 2.5x vs. multicore CPU

Ultrabook: Energy savings compared to multi-core CPU

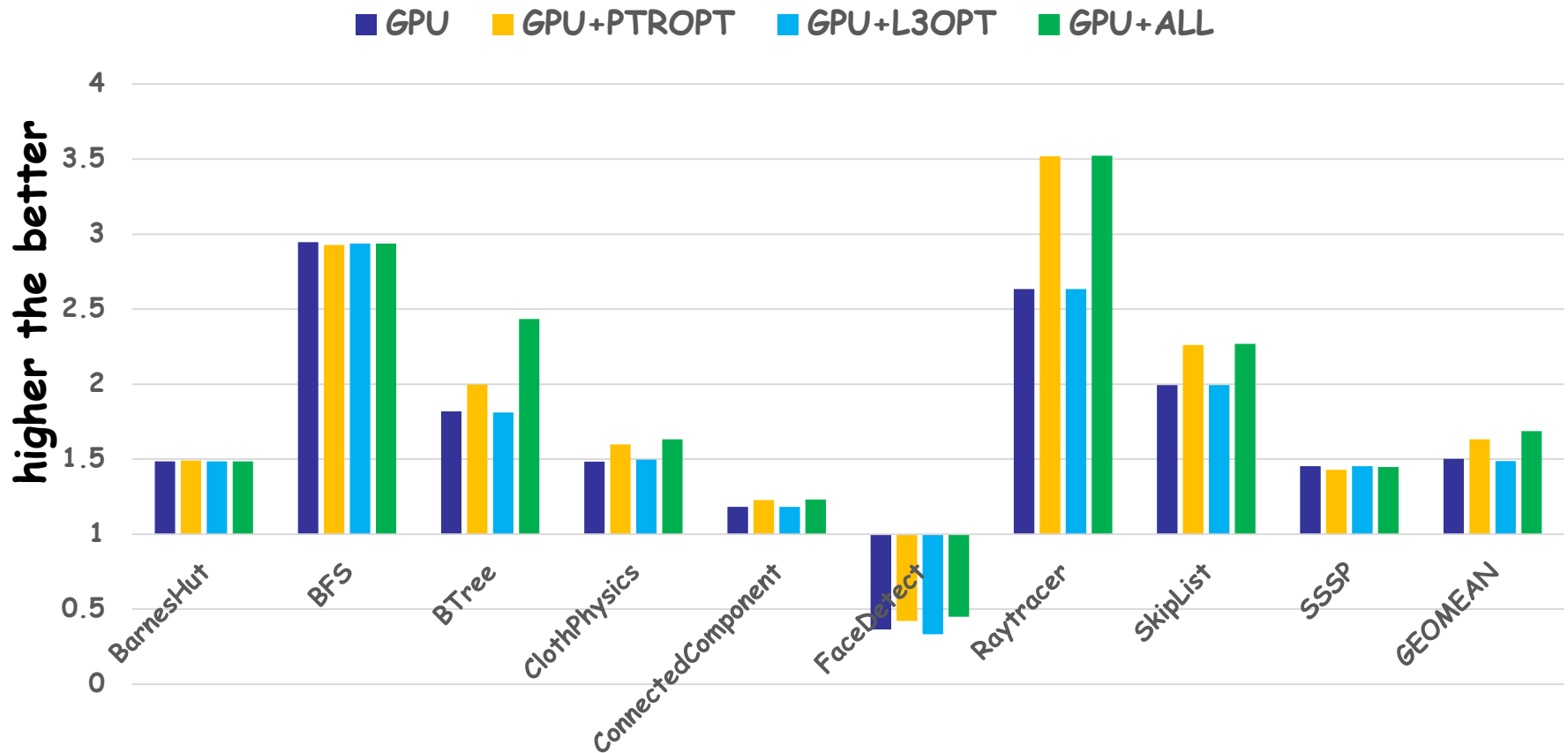


Average energy savings of 2.04x vs. multicore CPU

Desktop: Speedup compared to multi-core CPU



Desktop: Energy savings compared to multi-core CPU



Average energy savings of 1.7x vs. multicore CPU

GPU Programming is hard

