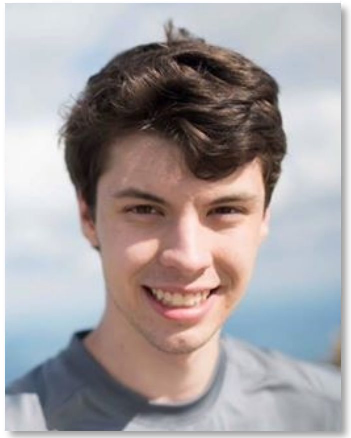
High-Performance GPU-to-CPU Transpilation and Optimization via High-Level Parallel Constructs


William S. Moses

MIT


Ivan R. Ivanov

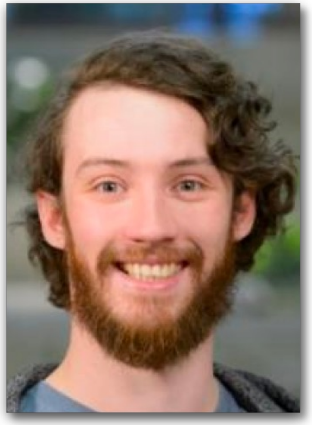
Tokyo Tech


Jens Domke

Riken


Toshio Endo

Tokyo Tech


Johannes Doerfert

LLNL


Alex Zinenko

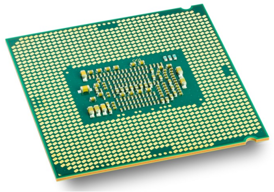
Google

PPoPP

Feb 27, 2023

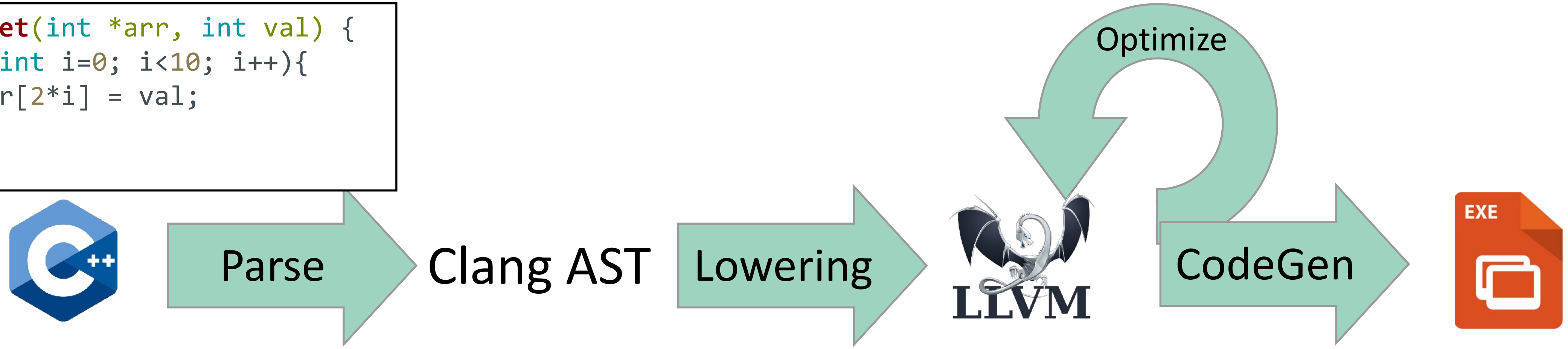
A Diverse Parallel Ecosystem

- Recent explosion of parallel software packages and hardware architectures.
- Changing landscape frequently requires re-engineering application to run at all, let alone fast.
- Existing approaches require some rewriting in either a performance portability library or DSL – infeasible for large / complex applications.
- Can fast and automated parallel performance portability be achieved through the compiler?



The Current Compilation Pipeline

```
void set(int *arr, int val) {
  for(int i=0; i<10; i++){
    arr[2*i] = val;
  }
}
```




```
FunctionDecl set 'void (int *, int)'
ForStmt
DeclStmt
`-VarDecl used i 'int' cinit
`-IntegerLiteral 'int' 0
BinaryOperator 'bool' '<'
|-ImplicitCastExpr 'int' <LValueToRValue>
| `-DeclRefExpr 'int' lvalue Var 0x563e22a396b8 'i'
'int'
`-IntegerLiteral 'int' 10
UnaryOperator 'int' postfix '++'
`-DeclRefExpr 'int' lvalue Var 0x563e22a396b8 'i' 'int'
…
```

```
define void @_Z3setPii(i32* %0, i32 %1) {
  br label %4

3: ; preds = %4
  ret void

4: ; preds = %2, %4
  %5 = phi i64 [ 0, %2 ], [ %8, %4 ]
  %6 = shl i64 %5, 1
  %7 = getelementptr inbounds i32, i32* %0, i64 %6
  store i32 %1, i32* %7
  %8 = add i64 %5, 1
  %9 = icmp eq i64 %8, 10
  br i1 %9, label %3, label %4
}
```

Losing High Level Structure

- LLVM, while general enough to represent any program, must represent all parts of a program in a single, low-level IR
  - Loses control flow constructs (if, for, etc)
  - Hides parallelism behind runtime
  - High-level semantics & properties cannot be represented and are lost.

```
void foo(DataStructure& x) {
  print(size(x));
  insert(x);
  print(size(x));
}
```

```
define void @foo(ptr %x) {
  %2 = call @size(ptr %x)
  call @print(i32 %2)
  call @insert(ptr %x)
  ; %3 = add i32 %2, 1
  %3 = call @size(ptr %x)
  call @print(i32 %3)
  ret void
}
```

GPU Programming

- Mainstream compilers do not have a high-level representation of parallelism, making optimization difficult or impossible
- This is accentuated for GPU programs where the kernel is kept in a separate module to allow emission of different assembly and synchronization is treated as a complete optimization barrier.

```
__global__ void normalize(int *out, int* in, int n) {
  int tid = blockIdx.x;
  if (tid < n)
    out[tid] = in[tid] / sum(in, n);
}

void launch(int *out, int* in, int n) {
  normalize<<<n>>>(d_out, d_in, n);
}
```

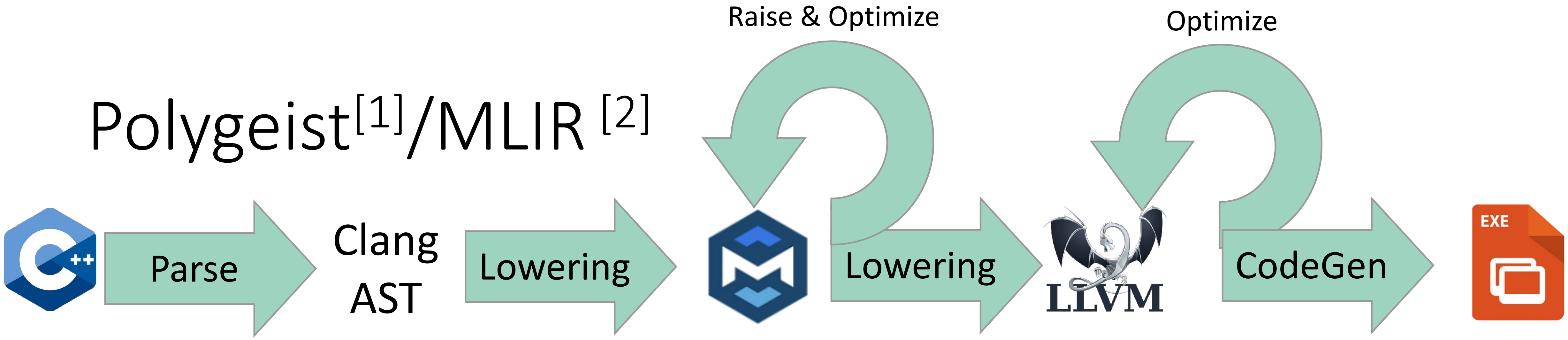
Host Code

```
target triple = "x86_64-unknown-linux-gnu"

define void @_Z6launchPiS_i(i32* %out,
                            i32* %in,
                            i32 %n) {
  call i32 @pushCallConfiguration(…)
  call i32 @cudaLaunch(@_device_stub, …)
  ret void
}
```

Device Code

```
target triple = ”nvptx”

define void @_Z9normalize(i32* %out,
                          i32* %in, i32 %n) {
  %4 = call i32 @llvm.tid.x()
  %5 = icmp slt i32 %4, %n
  br i1 %5, label %6, label %13

6:
  %8 = getelementptr i32, i32* %in, i32 %4
  %9 = load i32, i32* %8, align 4
  %10 = call i32 @_Z3sumPii(i32* %in, i32 %n)
  %11 = sdiv i32 %9, %10
  %12 = getelementptr i32, i32* %out, i32 %4
  store i32 %11, i32* %12, align 4
  br label %13

13:
  ret void
}
```

Polygeist[1]/MLIR [2]



- Generic C and C++ frontend that generates "standard" and user-defined MLIR (templates, classes, unions, etc. all supported)
- Preserves the structure of programs (parallelism, control flow, etc)
- Raising transformations for raising "standard" MLIR to high-level
- Collection of high-level optimization and analysis passes (general mem2reg, parallel optimizations)

[1] Polygeist: Raising C to Polyhedral MLIR; Moses, Chelini, Zhao, and Zinenko. PACT '21.
[2] MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. CGO'21.

Polygeist Frontend Example

```
void set(int *arr, int val) {
  for(int i=0; i<10; i++){
    arr[2*i] = val;
  }
}
```

```
func @set(%arg0: memref<?xi32>, %arg1: i32) {
   %c0 = constant 0 : index
   %0 = alloca() : memref<1xmemref<?xi32>>
   store %arg0, %0[%c0] : memref<1xmemref<?xi32>>
   %1 = alloca() : memref<1xi32>
   store %arg1, %1[%c0] : memref<1xi32>
   %c0_i32 = constant 0 : i32
   %c2_i32 = constant 2 : i32
   %c10_i32 = constant 10 : i32
   %2 = index_cast %c10_i32 : i32 to index
   scf.for %arg2 = %c0_i32 to %2 {
     %3 = index_cast %arg2 : index to i32
     %4 = alloca() : memref<1xi32>
     store %3, %4[%c0] : memref<1xi32>
     %5 = load %0[%c0] : memref<1xmemref<?xi32>>
     %6 = load %4[%c0] : memref<1xi32>
     %7 = muli %c2_i32, %6 : i32
     %8 = index_cast %7 : i32 to index
     %9 = load %1[%c0] : memref<1xi32>
     store %9, %5[%8] : memref<?xi32>
   }
   return
 }
```

Polygeist Raising

```
func @set(%arg0: memref<?xi32>, %arg1: i32) {
    %c0 = constant 0 : index
    %0 = alloca() : memref<1xmemref<?xi32>>
    store %arg0, %0[%c0] : memref<1xmemref<?xi32>>
    %1 = alloca() : memref<1xi32>
    store %arg1, %1[%c0] : memref<1xi32>
    %c0_i32 = constant 0 : i32
    %c10_i32 = constant 10 : i32
    %2 = index_cast %c10_i32 : i32 to index
    scf.for %arg2 = %c0_i32 to %2 {
      %3 = index_cast %arg2 : index to i32
      %4 = alloca() : memref<1xi32>
      store %3, %4[%c0] : memref<1xi32>
      %5 = load %0[%c0] : memref<1xmemref<?xi32>>
      %c2_i32 = constant 2 : i32
      %6 = load %4[%c0] : memref<1xi32>
      %7 = muli %c2_i32, %6 : i32
      %8 = index_cast %7 : i32 to index
      %9 = load %1[%c0] : memref<1xi32>
      store %9, %5[%8] : memref<?xi32>
    }
    return
  }
```

Polygeist Raising

```
func @set(%arg0: memref<?xi32>, %arg1: i32) {
    %c0 = constant 0 : index




    %c0_i32 = constant 0 : i32
    %c10_i32 = constant 10 : i32
    %2 = index_cast %c10_i32 : i32 to index
    scf.for %arg2 = %c0_i32 to %2 {
      %3 = index_cast %arg2 : index to i32



      %c2_i32 = constant 2 : i32

      %7 = muli %c2_i32, %3 : i32
      %8 = index_cast %7 : i32 to index

      store %arg1, %arg0[%8] : memref<?xi32>
    }
    return
  }
```

1. Mem2Reg

Polygeist Raising

```
func @set(%arg0: memref<?xi32>, %arg1: i32) {
    %c0 = constant 0 : index
    %c2 = constant 2 : i32
    %c10 = constant 10 : i32





    scf.for %arg2 = %c0 to %c10 {






      %7 = muli %c2_i32, %arg2 : index


      store %arg1, %arg0[%7] : memref<?xi32>
    }
    return
  }
```

1. Mem2Reg
2. Canonicalize

Polygeist Raising

```
func @set(%arg0: memref<?xi32>, %arg1: i32) {







    affine.for %arg2 = 0 to 10 {







      affine.store %arg1, %arg0 [2 * %arg2] :
                                  memref<?xi32>
    }
    return
  }
```

1. Mem2Reg
2. Canonicalize
3. If legal, raise while to for, for to affine, etc

Polygeist Raising

```
func @set(%arg0: memref<?xi32>, %arg1: i32) {
    affine.for %arg2 = 0 to 10 {
      affine.store %arg1, %arg0 [2 * %arg2] :
                                      memref<?xi32>
    }
    return
  }
```

```
void set(int *arr, int val) {
  for(int i=0; i<10; i++){
    arr[2*i] = val;
  }
}
```

Preserving the GPU parallel structure

- Maintain GPU parallelism in a form understandable to the compiler
- Enables optimization between caller and kernel
- Enable parallelism-specific optimization

```
__global__ void normalize(int *out, int *in, int n) {
  int tid = blockIdx.x;
  if (tid < n)
    out[tid] = in[tid] / sum(in, n);
}

void launch(int *out, int* in, int n) {
  normalize<<<n>>>(d_out, d_in, n);
}
```

→

```
func @_Z6launch(%out: memref<?xi32>,
                %in: memref<?xi32>, %n: i32) {
  %c1 = constant 1 : index
  %c0 = constant 0 : index

  parallel (%tid) = (%c0) to (%n) step (%c1) {
    %2 = load %in[%tid]
    %sum = call @_Z3sumPii(%in, %n)
    %4 = divsi %2, %sum : i32
    store %4, %out[%tid]
    yield
  }
  return
}
```

Preserving the GPU parallel structure

- Maintain GPU parallelism in a form understandable to the compiler
- Enables optimization between caller and kernel
- Enable parallelism-specific optimization

```
__global__ void normalize(int *out, int *in, int n) {
  int tid = blockIdx.x;
  if (tid < n)
    out[tid] = in[tid] / sum(in, n);
}

void launch(int *out, int* in, int n) {
  normalize<<<n>>>(d_out, d_in, n);
}
```

```
func @_Z6launch(%out: memref<?xi32>,
                %in: memref<?xi32>, %n: i32) {
  %c1 = constant 1 : index
  %c0 = constant 0 : index
  %sum = call @_Z3sumPii(%in, %n)
  parallel (%tid) = (%c0) to (%n) step (%c1) {
    %2 = load %in[%tid]

    %4 = divsi %2, %sum : i32
    store %4, %out[%tid]
    yield
  }
  return
}
```

Preserve the parallel structure

```
func @launch(%h_out : memref<?xf32>, %h_in : memref<?xf32>, %n : i64) {

  parallel.for (%gx, %gy, %gz) = (0, 0, 0) to (grid.x, grid.y, grid.z) {

    %shared_val = memref.alloca : memref<f32>

    parallel.for (%tx, %ty, %tz) = (0, 0, 0) to (blk.x, blk.y, blk.z) {

      if %tx == 0 {
         store …, %shared_val[] : memref<f32>
      }

      polygeist.barrier(%tx, %ty, %tz)

      …
    }
  }
}
```

Synchronization via Memory

- Synchronization (sync_threads) ensures all threads within a block finish executing codeA before executing codeB
- The desired synchronization behavior can be reproduced by defining sync_threads to have the union of the memory semantics of the code before and after the sync.
- This prevents code motion of instructions which require the synchronization for correctness, but permits other code motion (e.g. index computation).

```
codeA(fib(idx));

sync_threads;

codeB(fib(idx));
```

↕

```
off = fib(idx);
codeA(off);

sync_threads;

codeB(off);
```

Synchronization via Memory

- High-level synchronization representation enables new optimizations, like sync elimination.
- A synchronize instruction is not needed if the set of read/writes before the sync don't conflict with the read/writes after the sync.

```
__global__ void bpnn_layerforward(...) {
  __shared__ float node[HEIGHT];
  __shared__ float weights[HEIGHT][WIDTH];

  if ( tx == 0 )
    node[ty] = input[index_in] ;

  // Unnecessary Barrier #1
  // None of the read/writes below the sync
  //  (weights, hidden)
  // intersect with the read/writes above the sync
  //  (node, input)
  __syncthreads();


  // Unnecessary Store #1
  weights[ty][tx] = hidden[index];

  __syncthreads();

  // Unnecessary Load #1
  weights[ty][tx] = weights[ty][tx] * node[ty];
  …
}
```

GPU Transpilation

- A unified representation of parallelism enables programs in one parallel architecture (e.g. CUDA) to be compiled to another (e.g. OpenMP)
- Most CPU backends do not have an equivalent block synchronization
  - Many existing approaches create a heavy-weight state machine of all synchronizations that stores all values [1,2]
  - Efficiently lower a top-level synchronization by distributing the parallel for loop around the sync, and interchanging control flow, pioneered by MCUDA for source code [3] and used in POCL, Ocelot, and COX

[1] Efficient Compilation of Fine-Grained SPMD-Threaded Programs for Multicore CPUs. CGO (2010)
[2] Improving performance of OpenCL on CPUs, CC (2012)
[3] MCUDA: An Efficient Implementation of CUDA Kernels for Multi-core CPUs. In Languages and Compilers for Parallel Computing (2008)

GPU Synchronization Lowering: Fission

- Outermost synchronization can be handled by performing fission on the surrounding parallel for loop.

```
parallel_for %i = 0 to N {
  codeA(%i);
  sync_threads;
  codeB(%i);
}
```

```
parallel_for %i = 0 to N {
  codeA(%i);
}
parallel_for %i = 0 to N {
  codeB(%i);
}
```

GPU Synchronization Lowering: Registers

- Registers defined before the synchronization and used after the synchronization must be preserved through an allocation.
- If the memory semantics allow us to more efficiently recompute the value, it doesn't need to be stored.

```
parallel_for %i = 0 to N {
  %off = %i + 1
  codeA(%off);
  sync_threads;
  codeB(%off);
}
```

```
%offm = alloca N
parallel_for %i = 0 to N {
  %off = %i + 1
  %offm[%i] = %off
  codeA(%off);
}
parallel_for %i = 0 to N {
  codeB(%off_m%[%i]);
}
```

```
parallel_for %i = 0 to N {
  %off = %i + 1
  codeA(%off);
}
parallel_for %i = 0 to N {
  %off = %i + 1
  codeB(%off);
}
```

GPU Synchronization Lowering: Control Flow

- Synchronization within control can be lowered by splitting around the control flop and interchanging the parallelism.

```
parallel_for %i = 0 to N {
  for %j = … {
    codeB1(%i, %j);
    sync_threads;
    codeB2(%i, %j);
  }
}
```

```
for %j = … {
  parallel_for %i = 0 to N {
    codeB1(%i, %j);
    sync_threads;
    codeB2(%i, %j);
  }
}
```

```
for %j = … {
  parallel_for %i = 0 to N {
    codeB1(%i, %j);
  }
  parallel_for %i = 0 to N {
    codeB2(%i, %j);
  }
}
```

GPU Synchronization Lowering: Control Flow

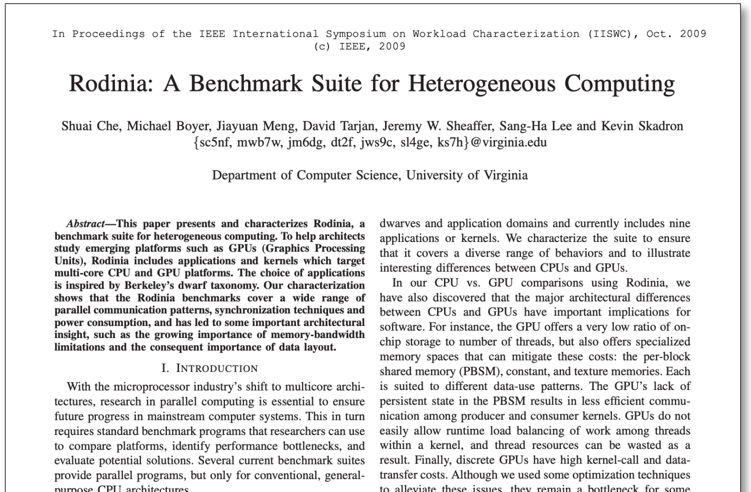
- Less structured control flow can still be lowered, but requires more infrastructure.

```
parallel_for %i = 0 to N {
  do {
    run(%i);
    sync_threads;
    run2(%i);
  } while (condition());
}
```

```
%helper = alloca i1
do {
  parallel_for %i = 0 to N {
    run(%i);
    sync_threads;
    run2(%i);
    %c = condition();
    if %i == 0 {
      store %helper[] = %c;
    }
  }
  %c2 = load helper[]
} while (%c2);
```

Evaluating Performance Portability

- Motivation of this work was to enable the often GPU-only versions of programs to run on the CPU-only systems, like the Fugaku supercomputer.
- Having demonstrated the ability to convert GPU code to CPU, how close do these transpiled versions get to hand written CPU performance?



In Proceedings of the IEEE International Symposium on Workload Characterization (IISWC), Oct. 2009

Rodinia: A Benchmark Suite for Heterogeneous Computing

Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee and Kevin Skadron
{sc5nf, mwb7w, jm6dg, dt2f, jws9c, sl4ge, ks7h}@virginia.edu

Department of Computer Science, University of Virginia

Abstract—**This paper presents and characterizes Rodinia, a benchmark suite for heterogeneous computing. To help architects study emerging platforms such as GPUs (Graphics Processing Units), Rodinia includes applications and kernels which target multi-core CPU and GPU platforms. The choice of applications is inspired by Berkeley's dwarf taxonomy. Our characterization shows that the Rodinia benchmarks cover a wide range of parallel communication patterns, synchronization techniques and power consumption, and has led to some important architectural insight, such as the growing importance of memory-bandwidth limitations and the consequent importance of data layout.**

I. Introduction

With the microprocessor industry's shift to multicore architectures, research in parallel computing is essential to ensure future progress in mainstream computer systems. This in turn requires standard benchmark programs that researchers can use to compare platforms, identify performance bottlenecks, and evaluate potential solutions. Several current benchmark suites provide parallel programs, but only for conventional, general-purpose CPU architectures.

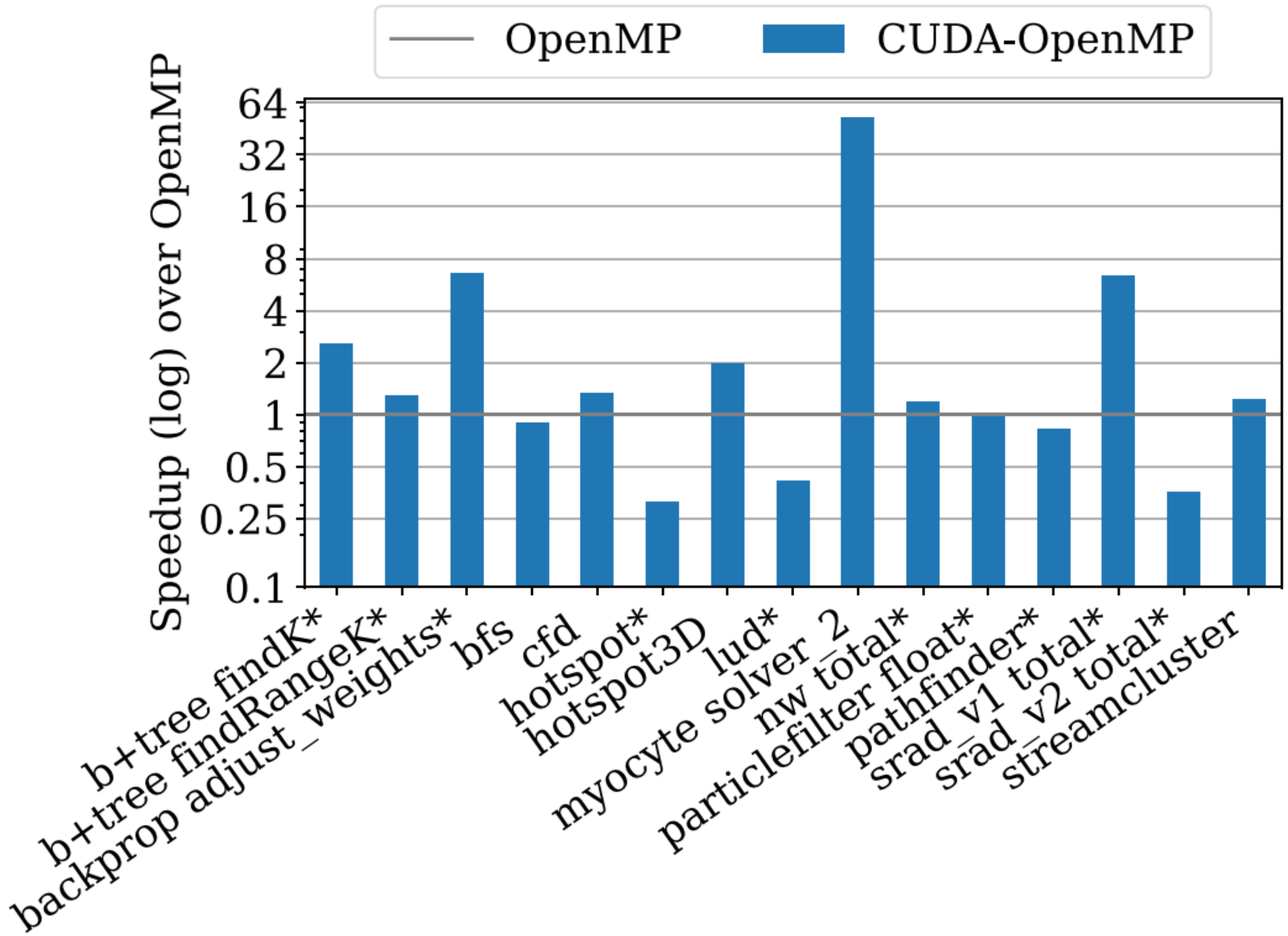
dwarves and application domains and currently includes nine applications or kernels. We characterize the suite to ensure that it covers a diverse range of behaviors and to illustrate interesting differences between CPUs and GPUs.

In our CPU vs. GPU comparisons using Rodinia, we have also discovered that the major architectural differences between CPUs and GPUs have important implications for software. For instance, the GPU offers a very low ratio of on-chip storage to number of threads, but also offers specialized memory spaces that can mitigate these costs: the per-block shared memory (PBSM), constant, and texture memories. Each is suited to different data-use patterns. The GPU's lack of persistent state in the PBSM results in less efficient communication among producer and consumer kernels. GPUs do not easily allow runtime load balancing of work among threads within a kernel, and thread resources can be wasted as a result. Finally, discrete GPUs have high kernel-call and data-transfer costs. Although we used some optimization techniques to alleviate these issues, they remain a bottleneck for some

PyTorch

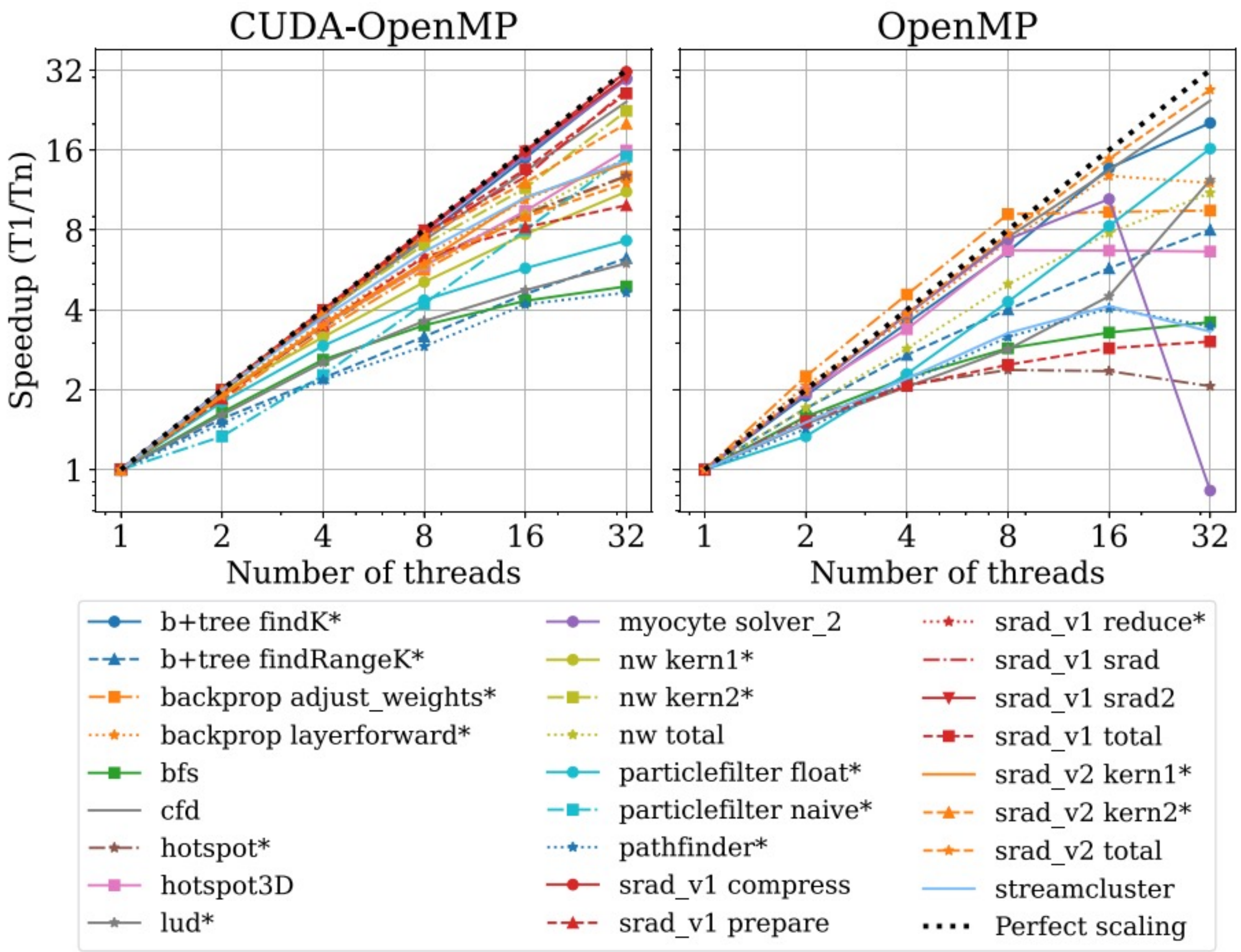
Rodinia Benchmarks

- Geomean 54% ***improvement*** over hand-written OpenMP code.



Rodinia Scalability

- CUDA-OpenMP has 14x speedup over single code program on 32 cores
- OpenMP has 7x speedup on 32 cores

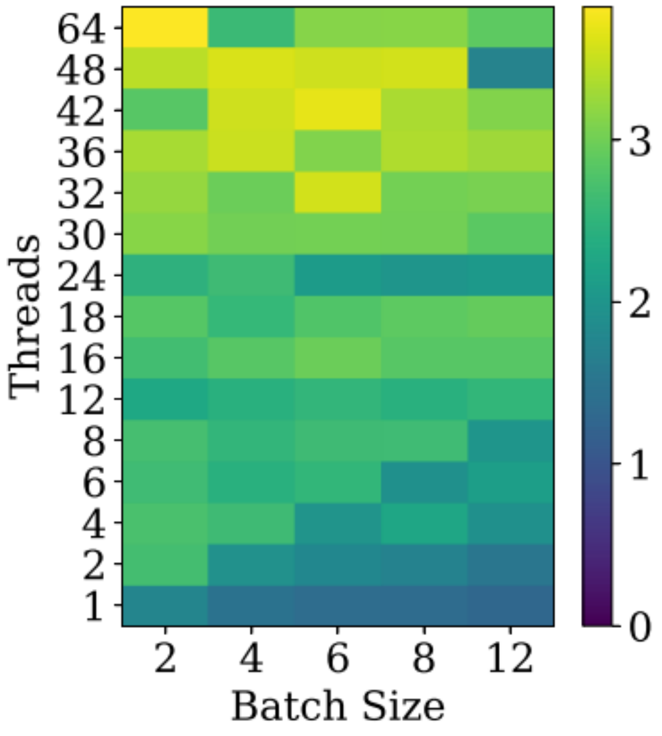


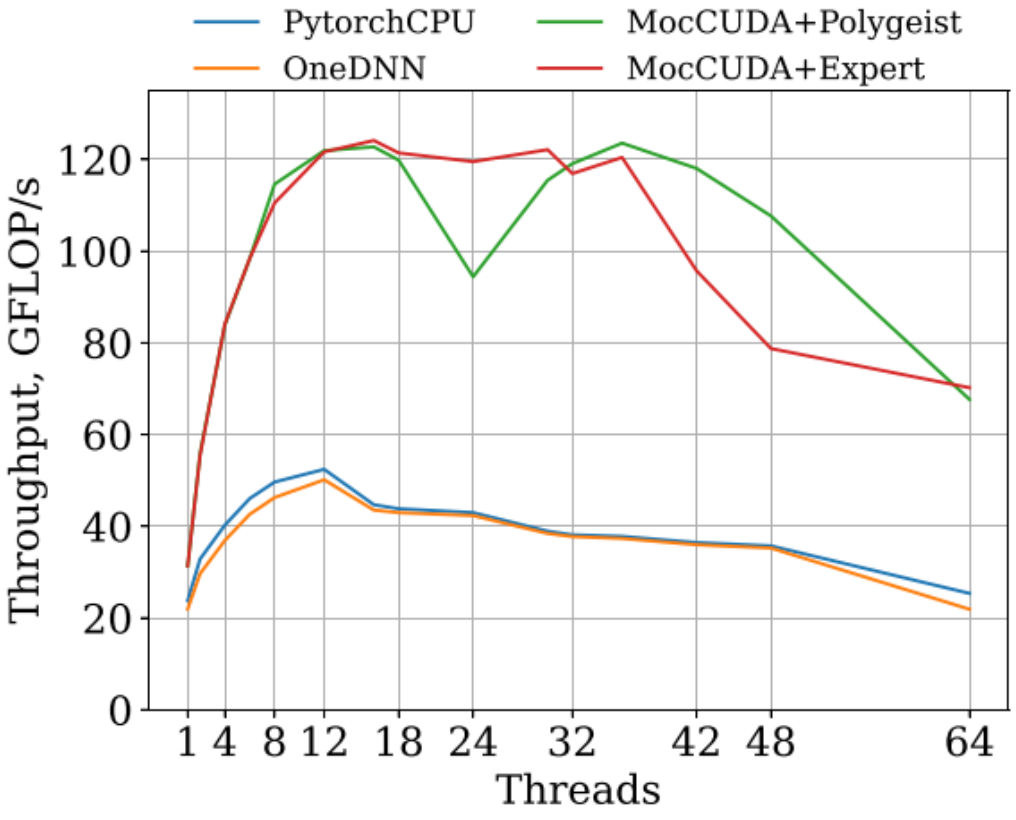
PyTorch Benchmark

- Built compatibility layer called MocCUDA which allows us to overwrite CUDA versions of libraries with CPU versions, including those generated by Polygeist.
- Evaluate training of Resnet-50 on Fugaku supercomputer
- Tested existing Fugaku-tuned CPU backends, as well as expert-written kernels

PyTorch Benchmark

MocCUDA outperforms Fujitsu-tuned oneDNN backend by 2.7x on average across batch sizes/thread counts (ranges 1.2x - 4.5x)





Conclusion

- Extending Polygeist/MLIR, we developed an end-to-end system capable of representing, optimizing, and transpiling CPU and GPU parallel programs.
- Development of a high-level barrier operation, whose behavior is defined by memory semantics, enables interoperability with serial and parallel-specific optimizations.
- Ability to preserve high-level structure, including parallelism, barriers, and control flow enables more efficient lowering to CPU's
- Validate approach by performing GPU to CPU transpilation on Rodinia and a PyTorch Resnet-50, which runs faster than existing CPU backends
- LLVM incubator project, open sourced on Github (github.com/llvm/Polygeist), see polygeist.mit.edu & discuss on Discourse!

Acknowledgements

- Thanks to Valentin Churavy, Albert Cohen, Charles Leiserson, Douglas Kogut, Jiahao Li, Bojan Serafimov, and Cosmin Oancea for thoughtful discussions on this work.
- William S. Moses was supported in part by a DOE Computational Sciences Graduate Fellowship, in part by Los Alamos National Laboratories, and in part by the United States Air Force Research Laboratory. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the United States Air Force or the U.S. Government.
- Johannes Doerfert was supported in part by the Exascale Computing Project (17-SC-20-SC) and in part by the Lawrence Livermore National Security, LLC ("LLNS") via MPO No. B642066, LLNL-CONF-843530.
- This work was supported in part by the Japan Society for the Promotion of Science KAKENHI Grant Number 19H04119 and by the Japanese New Energy and Industrial Technology Development Organization (NEDO).

Conclusion

- Extending Polygeist/MLIR, we developed an end-to-end system capable of representing, optimizing, and transpiling CPU and GPU parallel programs.
- Development of a high-level barrier operation, whose behavior is defined by memory semantics, enables interoperability with serial and parallel-specific optimizations.
- Ability to preserve high-level structure, including parallelism, barriers, and control flow enables more efficient lowering to CPU's
- Validate approach by performing GPU to CPU transpilation on Rodinia and a PyTorch Resnet-50, which runs faster than existing CPU backends
- LLVM incubator project, open sourced on Github (github.com/llvm/Polygeist), see polygeist.mit.edu & discuss on Discourse!

Backup Slides

GPU Synchronization Lowering: Registers

- Registers defined before the synchronization and used after the synchronization must be preserved through an allocation.
- If the memory semantics allow us to more efficiently recompute the value, it doesn't need to be stored.

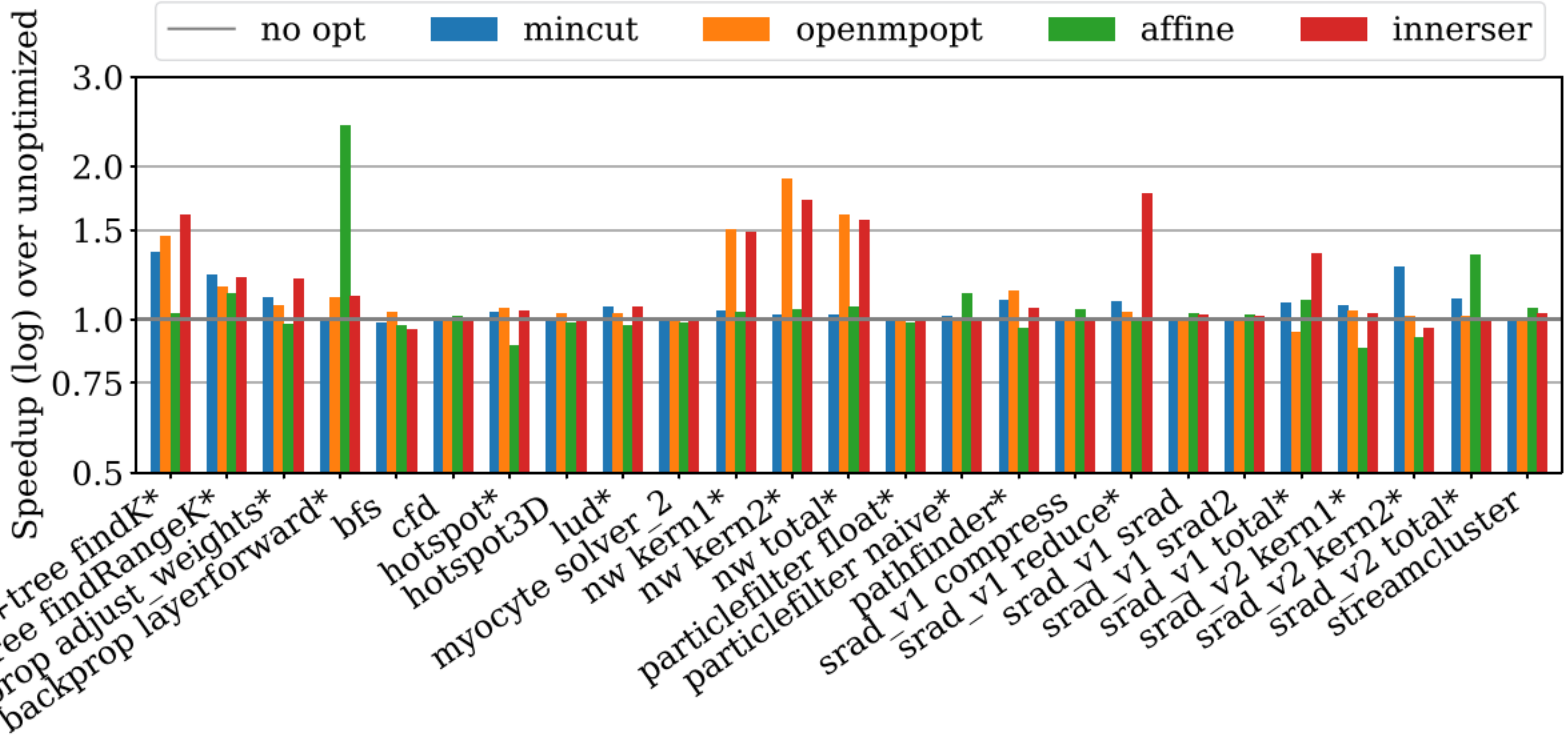
```
parallel_for %i = 0 to N {
  %off = %i + 1
  codeA(%off);
  sync_threads;
  codeB(%off);
}
```

```
%offm = alloca N
parallel_for %i = 0 to N {
  %off = %i + 1
  %offm[%i] = %off
  codeA(%off);
}
parallel_for %i = 0 to N {
  codeB(%off_m%[%i]);
}
```

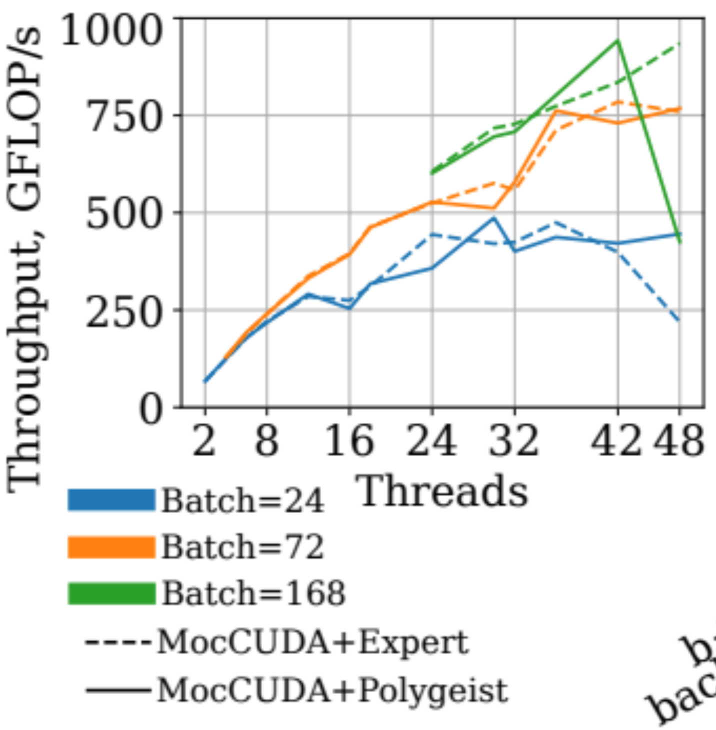
```
parallel_for %i = 0 to N {
  %off = %i + 1
  codeA(%off);
}
parallel_for %i = 0 to N {
  %off = %i + 1
  codeB(%off);
}
```

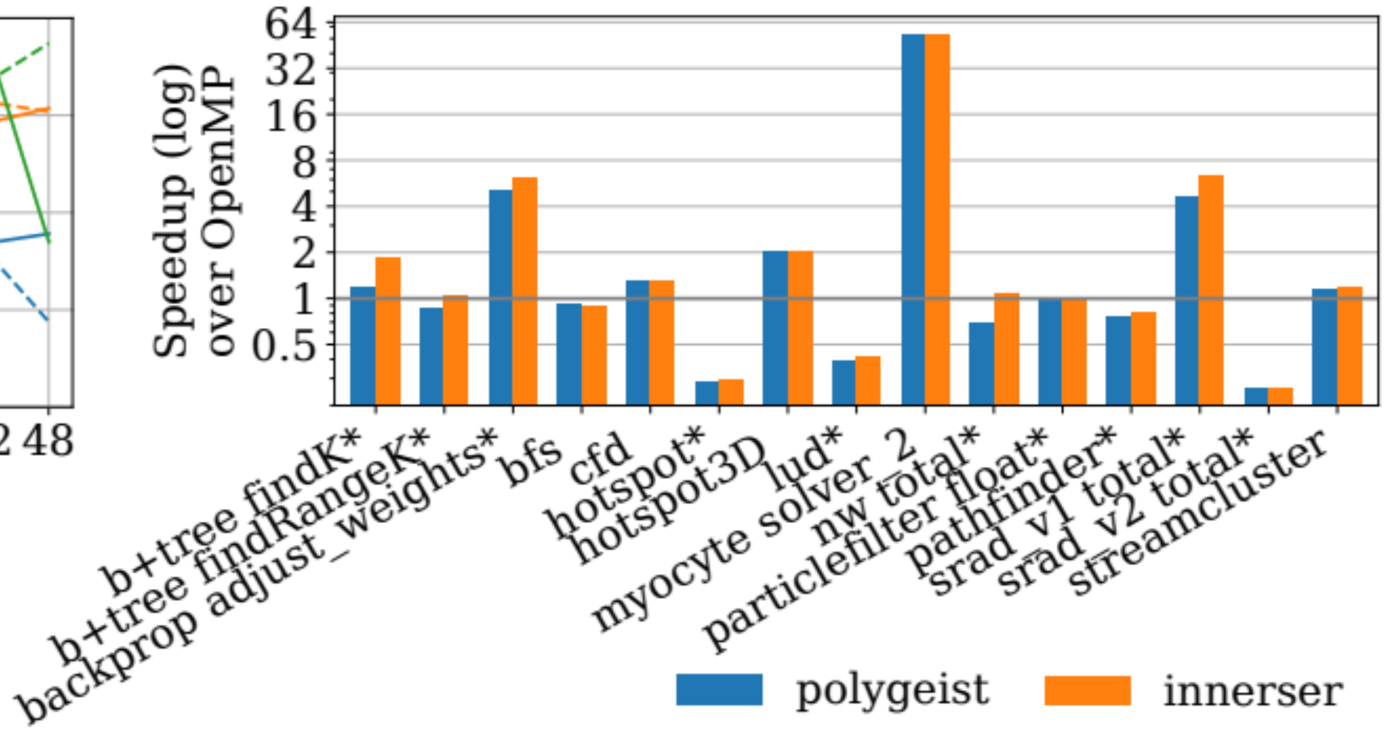
Rodinia Ablation

- Mincut: 5.8%; OpenMPOpt: 10.5%; Affine: 5.4% (2.4x backprop)



PyTorch Scaling





Conclusion

- Optimizable, multi-level operations are key to compiler extensibility and therefore performance
- Polygeist/MLIR is a new Clang-based compiler that allows you to leverage this extensibility
  - C/C++ frontend for MLIR
  - Compiler transformations for raising MLIR to a higher-level
  - Collection of high-level optimization passes (general mem2reg, etc)
  - Polyhedral optimization via novel optimizations and integrating prior tools into MLIR
  - Parallel/GPU optimizations & transformations
- Polygeist beats existing polyhedral tools on sequential and parallel code
- Polygeist can optimize and transcompile your GPU/parallel code
- Supports recognizing and lowering to custom ops/dialects
- LLVM incubator project, open sourced on Github, see https://polygeist.mit.edu and discuss on Discourse!

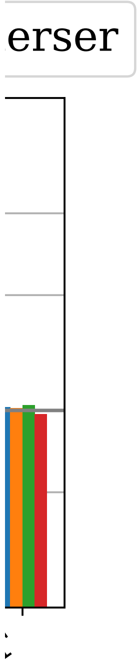
Acknowledgements

- Thanks to Valentin Churavy, Albert Cohen, Henk Corporaal, Tobias Grosser, and Charles Leiserson for thoughtful discussions on this work.
- William S. Moses was supported in part by a DOE Computational Sciences Graduate Fellowship, in part by Los Alamos National Laboratories, and in part by the United States Air Force Research Laboratory. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the United States Air Force or the U.S. Government.
- Lorenzo Chelini is partially supported by the European Commission Horizon 2020
- Ruizhe Zhao is sponsored by UKRI and Corerain Technologies Ltd. The support of the UK EPSRC is also gratefully acknowledged.
- The work was supported in part by JSPS KAKENHI Grant Number JP19H04119 and in part by the Japan Society for the Promotion of Science KAKENHI Grant Number JP19H04119, and in part by the New Energy and Industrial Technology Development Organization (NEDO).

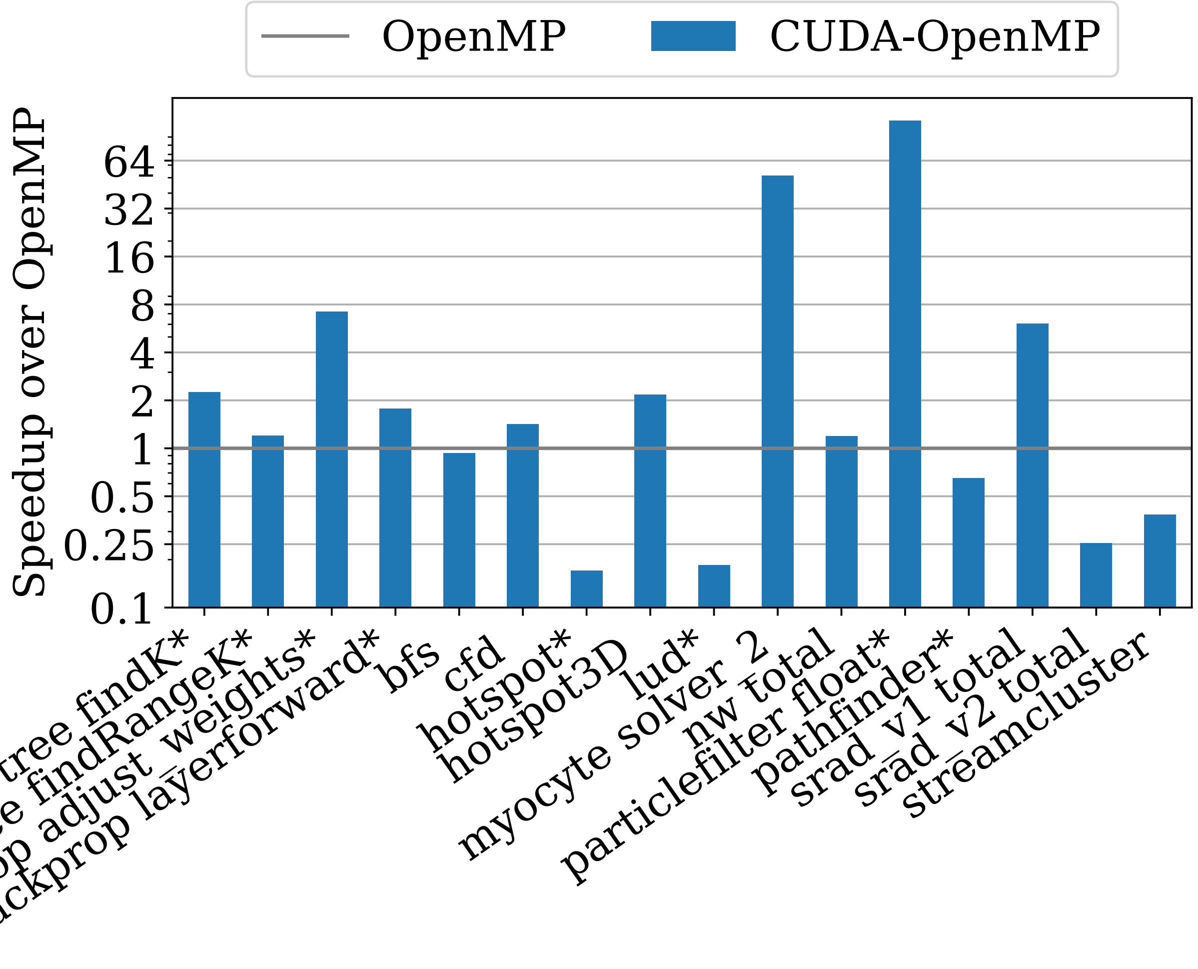
Conclusion

- Optimizable, multi-level operations are key to compiler extensibility and therefore performance
- Polygeist/MLIR is a new Clang-based compiler that allows you to leverage this extensibility
  - C/C++ frontend for MLIR
  - Compiler transformations for raising MLIR to a higher-level
  - Collection of high-level optimization passes (general mem2reg, etc)
  - Polyhedral optimization via novel optimizations and integrating prior tools into MLIR
  - Parallel/GPU optimizations & transformations
- Polygeist beats existing polyhedral tools on sequential and parallel code
- Polygeist can optimize and transcompile your GPU/parallel code
- Supports recognizing and lowering to custom ops/dialects
- LLVM incubator project, open sourced on Github, see https://polygeist.mit.edu and discuss on Discourse!

- Text



erser



OpenMP
CUDA-OpenMP
Speedup over OpenMP
64
32
16
8
4
2
1
0.5
0.25
0.1
b+tree findK*
b+tree findRangeK*
backprop adjust_weights*
backprop layerforward*
bfs
cfd
hotspot*
hotspot3D
lud*
myocyte solver 2
nw total
particlefilter float*
pathfinder*
srad_v1 total
srad_v2 total
streamcluster

GPU Memory Hierarchy

Per Thread	Per Block	Per GPU
Register	Shared Memory	Global Memory
~Bytes	~KBs	~GBs
Use Limits Parallelism	Use Limits Parallelism	

→ Slower, larger amount of memory

Case Study 2: GPUs

GPU Code →

CPU Code →

```
__device__ int sum(int* data, int n);

__global__ void square(int *out, int* in, int n) {
  int tid = blockIdx.x;
  if (tid < n)
    out[tid] = in[tid] / sum(in, n);
}

void launch(int *h_out, int* h_in, int n) {


  int *d_out, *d_in;
  cudaMalloc(&d_out, n*sizeof(n));
  cudaMalloc(&d_in, n*sizeof(n));

  cudaMemcpy(d_in, h_in, n*sizeof(n), cudaMemcpyHostToDevice);

  square<<<(n+31)/32, 32>>>(d_out, d_in, n);


  cudaMemcpy(h_out, h_out, n*sizeof(n), cudaMemcpyDeviceToHost);
}
```

CPU Memory (→ h_out, h_in)

GPU Memory (→ cudaMalloc)

Block (→ (n+31)/32), Thread (→ 32)

A first-class representation of parallelism

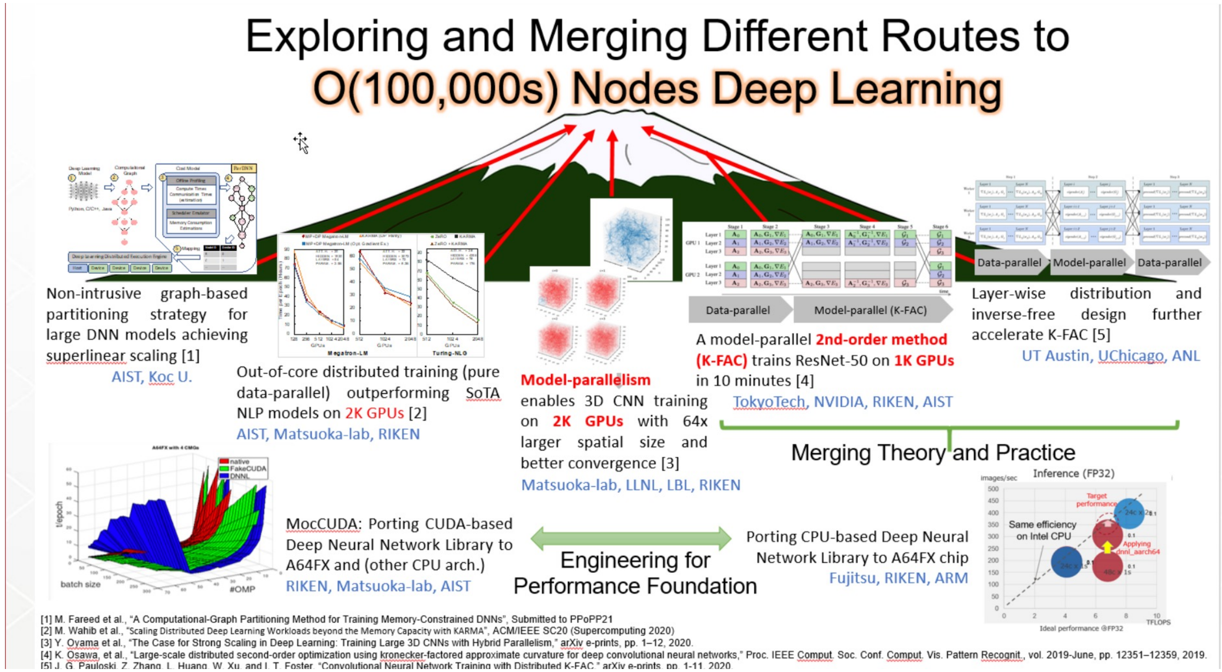
- Current mainstream compilers do not have a good notion or representation of parallelism
- This is accentuated for GPU programs where the kernel is kept in a separate module to allow emission of different assembly

```
target triple = "x86_64-unknown-linux-gnu”

define void @_Z6launchPiS_i(i32* %out, i32* %in, i32 %n)
{
  call i32 @__cudaPushCallConfiguration(…)
  call i32 @cudaLaunchKernel(@_device_stub, …)
  ret void
}
```

```
target triple = ”nvptx”

define void @_Z9normalizePiS_i(i32* %out, i32* %in, i32 %n) {
  %4 = call i32 @llvm.nvvm.read.ptx.sreg.tid.x()
  %5 = icmp slt i32 %4, %n
  br i1 %5, label %6, label %13

6: ; preds = %3
  %8 = getelementptr inbounds i32, i32* %in, i32 %4
  %9 = load i32, i32* %8, align 4
  %10 = call i32 @_Z3sumPii(i32* %in, i32 %n) #5
  %11 = sdiv i32 %9, %10
  %12 = getelementptr inbounds i32, i32* %out, i32 %4
  store i32 %11, i32* %12, align 4
  br label %13

13:
```

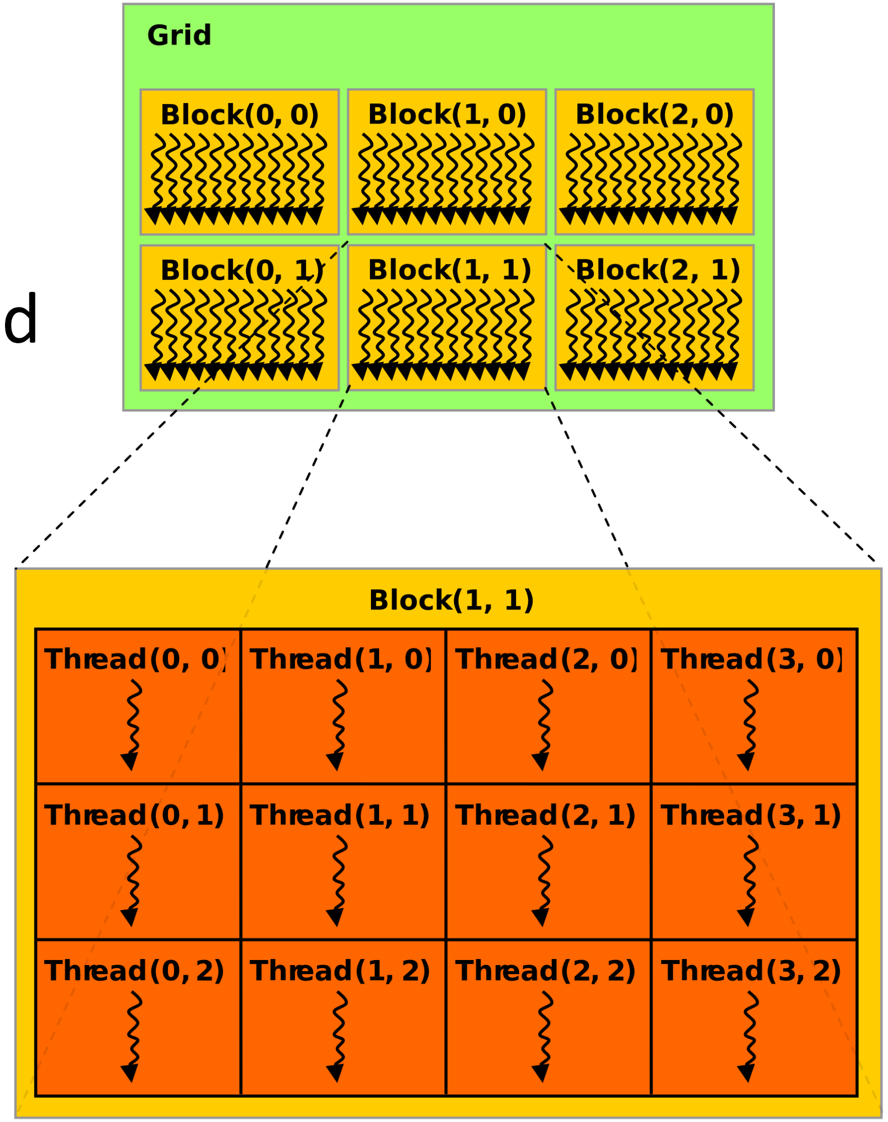
Open Research Directions

- How can we optimize GPU programs?
- Can we convert GPU to CPU (and vice versa)?
  - Working with Riken/Tokyo Tech to port GPU to Fugaku supercomputer
- What advantages can we gain from compiler representations?

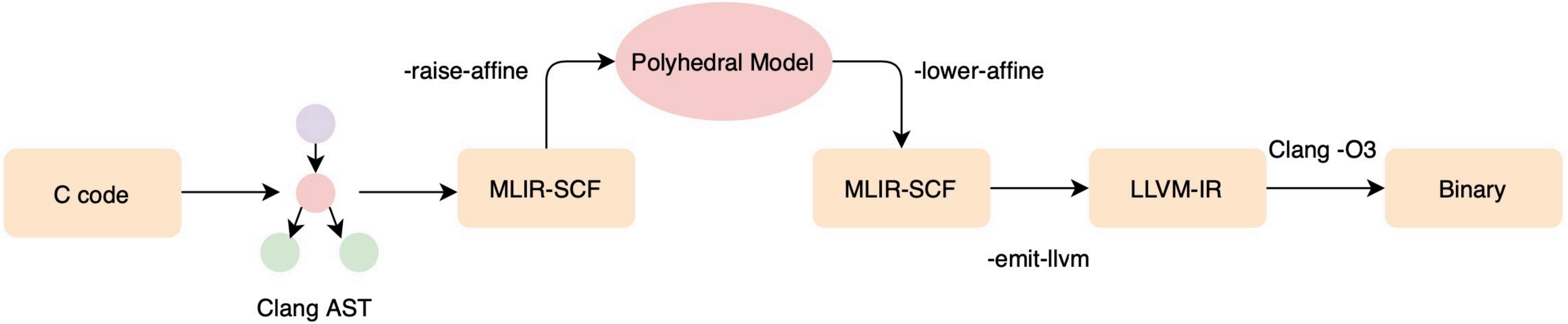


Introduction GPU Programming

- GPU threads are like CPU threads in which they can run in parallel.
- A group of threads (up to 32) are combined in a block
- Threads can share data and/or sync within a block but not between blocks
- All threads in a block are guaranteed to execute at the same time (and may run in lockstep)
- Blocks are not



The Polygeist Compilation Flow



- Generic C or C++ frontend that generates "standard" MLIR
- Raising transformations for transforming "standard" MLIR to polyhedral MLIR (Affine)
- Embedding of existing polyhedral tools (Pluto, CLooG) into MLIR
- Novel transformations (statement splitting, reduction detection) that rely on high-level compiler representation
- End-to-end evaluation of standard polyhedral benchmarks (Polybench)

“Case Study 3”: Your Programs!

- There are already several efforts starting using Polygeist/MLIR to leveraging the benefits of optimizable multi-level operations
  - SYCL
  - Circuit Compilation
  - BLAS Kernels
  - Databases
  - …
- If you’re interested in applying such techniques to your programs, please reach out!

GPU Synchronization Lowering

- Most CPU backends (e.g. Cilk, OpenMP) do not have an equivalent & general synchronization instruction (more akin to a barrier)
- Existing approaches create a heavy-weight state machine of all synchronizations that stores all values

GPU Synchronization Lowering: Registers

- Registers defined before the synchronization and used after the synchronization must be preserved through an allocation.
- If the memory semantics allow us to more efficiently recompute the value, it doesn't need to be stored.

```
parallel_for %i = 0 to N {
  %off = %i + 1
  codeA(%off);
  sync_threads;
  codeB(%off);
}
```

```
%offm = alloca N
parallel_for %i = 0 to N {
  %off = %i + 1
  %offm[%i] = %off
  codeA(%off);
}
parallel_for %i = 0 to N {
  codeB(%off_m%[%i]);
}
```

```
parallel_for %i = 0 to N {
  %off = %i + 1
  codeA(%off);
}
parallel_for %i = 0 to N {
  %off = %i + 1
  codeB(%off);
}
```

GPU Synchronization Lowering: Registers

- Registers defined before the synchronization and used after the synchronization must be preserved through an allocation.
- If the memory semantics allow us to more efficiently recompute the value, it doesn't need to be stored.

- ***[Question] Is distributing the parallelism around the barrier the best approach?***

- ***[Question] How do we minimize the runtime of preserving registers?***
  - Tradeoff parallel recompute vs preserve
  - Min Cut?

GPU Synchronization Lowering: Control Flow

- Synchronization within control flow (for, if, while, etc) can be lowered by splitting around the control flop and interchanging the parallelism.

```
parallel_for %i = 0 to N {
  codeA(%i);
  for %j = … {
    codeB1(%i, %j);
    sync_threads;
    codeB2(%i, %j);
  }
  codeC(%i);
}
```

```
parallel_for %i = 0 to N {
  codeA(%i);
  sync_threads;
  for %j = … {
    codeB1(%i, %j);
    sync_threads;
    codeB2(%i, %j);
  }
  sync_threads;
  codeC(%i);
}
```

```
parallel_for %i = 0 to N {
  codeA(%i);
}
parallel_for %i = 0 to N {
  for %j = … {
    codeB1(%i, %j);
    sync_threads;
    codeB2(%i, %j);
  }
}
parallel_for %i = 0 to N {
  codeC(%i);
}
```

GPU Synchronization Lowering: Control Flow

- Synchronization within control flow (for, if, while, etc) can be lowered by splitting around the control flop and interchanging the parallelism.

```
parallel_for %i = 0 to N {
  codeA(%i);
}
parallel_for %i = 0 to N {
  for %j = … {
    codeB1(%i, %j);
    sync_threads;
    codeB2(%i, %j);
  }
}
parallel_for %i = 0 to N {
  codeC(%i);
}
```

```
parallel_for %i = 0 to N {
  codeA(%i);
}
for %j = … {
  parallel_for %i = 0 to N {
    codeB1(%i, %j);
    sync_threads;
    codeB2(%i, %j);
  }
}
parallel_for %i = 0 to N {
  codeC(%i);
}
```

```
parallel_for %i = 0 to N {
  codeA(%i);
}
for %j = … {
  parallel_for %i = 0 to N {
    codeB1(%i, %j);
  }
  parallel_for %i = 0 to N {
    codeB2(%i, %j);
  }
}
parallel_for %i = 0 to N {
  codeC(%i);
}
```