Improving Performance of OpenCL on CPUs

Ralf Karrenberg
karrenberg@cs.uni-saarland.de

Sebastian Hack
hack@cs.uni-saarland.de

European LLVM Conference, London
April 12-13, 2012

UNIVERSITÄT DES SAARLANDES

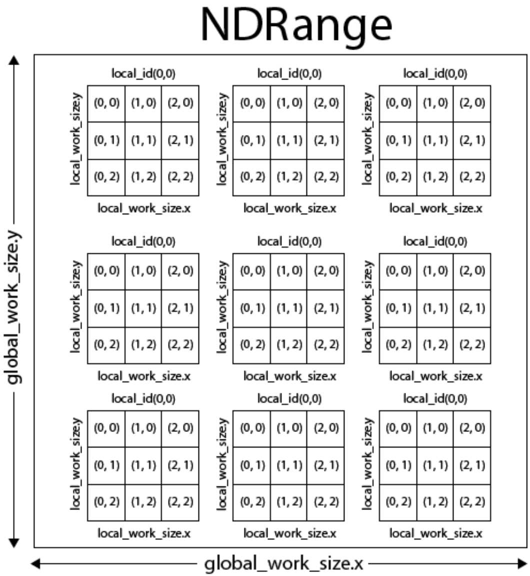
CLUSTER OF EXCELLENCE

VISUAL COMPUTING INSTITUTE

Data-Parallel Languages: OpenCL

```
__kernel void DCT(__global float * output,
                  __global float * input,
                  __global float * dct8x8,
                  __local  float * inter,
                  const    uint    width,
                  const    uint    blockWidth,
                  const    uint    inverse)
{
    uint tidX = get_global_id(0);
    ...
    inter[lidY*blockWidth + lidX] = ...
    barrier(CLK_LOCAL_MEM_FENCE);

    float acc = 0.0f;
    for(uint k=0; k < blockWidth; k++)
    {
        uint index1 = lidX*blockWidth + k;
        uint index2 = (inverse) ? lidY*blockWidth + k :
                                  k*blockWidth + lidY;
        acc += inter[index1] * dct8x8[index2];
    }
    output[tidY*width + tidX] = acc;
}
```

OpenCL: Execution Model



CPU Driver Implementation (2D, Naïve)

```
cl_int
clEnqueueNDRangeKernel(Kernel scalarKernel,
                       TA     argStruct,
                       int*   globalSizes,
                       int*   localSizes)
{
    int groupSizeX = globalSizes[0] / localSizes[0];
    int groupSizeY = globalSizes[1] / localSizes[1];

    // Loop over groups.
    for (int groupX=0; groupX<groupSizeX; ++groupX) {
      for (int groupY=0; groupY<groupSizeY; ++groupY) {
        // Loop over threads in group.
        for (int lidY=0; lidY<localSizes[1]; ++lidY) {
          for (int lidX=0; lidX<localSizes[0]; ++lidX) {
              scalarKernel(argStruct, lidX, lidY,
                           groupX, groupY,
                           globalSizes, localSizes);
        } }
}   } }
```

CPU Driver Implementation (2D, Group Kernel)

```
cl_int
clEnqueueNDRangeKernel(Kernel groupKernel,
                       TA     argStruct,
                       int*   globalSizes,
                       int*   localSizes)
{
    int groupSizeX = globalSizes[0] / localSizes[0];
    int groupSizeY = globalSizes[1] / localSizes[1];

    // Loop over groups.
    for (int groupX=0; groupX<groupSizeX; ++groupX) {
      for (int groupY=0; groupY<groupSizeY; ++groupY) {
        // Loop over threads in group.


        groupKernel(argStruct,
                    groupX, groupY,
                    globalSizes,
                    localSizes);
}   } }
```

CPU Driver Implementation (2D, Group Kernel, OpenMP)

```
cl_int
clEnqueueNDRangeKernel(Kernel groupKernel,
                       TA     argStruct,
                       int*   globalSizes,
                       int*   localSizes)
{
    int groupSizeX = globalSizes[0] / localSizes[0];
    int groupSizeY = globalSizes[1] / localSizes[1];

#pragma omp parallel for
    for (int groupX=0; groupX<groupSizeX; ++groupX) {
      for (int groupY=0; groupY<groupSizeY; ++groupY) {
        // Loop over threads in group.


        groupKernel(argStruct,
                    groupX, groupY,
                    globalSizes,
                    localSizes);
}   } }
```

Group Kernel (2D, Scalar)

```
void groupKernel(TA argStruct, int* groupIDs,
                 int* globalSizes, int* localSizes)
{
    for (int lidY=0; lidY<localSizes[1]; ++lidY) {
      for (int lidX=0; lidX<localSizes[0]; ++lidX) {

        scalarKernel(argStruct, lidX, lidY,
                     groupIDs, globalSizes,
                     localSizes); // to be inlined
}   } }
```

Group Kernel (2D, Scalar, Inlined)

```
void groupKernel(TA argStruct, int* groupIDs,
                 int* globalSizes, int* localSizes)
{
    for (int lidY=0; lidY<localSizes[1]; ++lidY) {

      for (int lidX=0; lidX<localSizes[0]; ++lidX) {
        uint tidX = get_global_id(0);
        ...
        inter[lidY*blockWidth + lidX] = ...
        barrier(CLK_LOCAL_MEM_FENCE);

        float acc = 0.0f;
        for(uint k=0; k < blockWidth; k++)
        {
          uint index1 = lidX*blockWidth + k;
          uint index2 = (inverse) ? lidY*blockWidth + k :
                                    k*blockWidth + lidY;
          acc += inter[index1] * dct8x8[index2];
        }
        output[tidY*width + tidX] = acc;
}   } }
```

Group Kernel (2D, Scalar, Inlined, Optimized (1))

```
void groupKernel(TA argStruct, int* groupIDs,
                 int* globalSizes, int* localSizes)
{
    for (int lidY=0; lidY<localSizes[1]; ++lidY) {

      for (int lidX=0; lidX<localSizes[0]; ++lidX) {
        uint tidX = localSizes[0] * groupIDs[0] + lidX;
        ...
        inter[lidY*blockWidth + lidX] = ...
        barrier(CLK_LOCAL_MEM_FENCE);

        float acc = 0.0f;
        for(uint k=0; k < blockWidth; k++)
        {
          uint index1 = lidX*blockWidth + k;
          uint index2 = (inverse) ? lidY*blockWidth + k :
                                    k*blockWidth + lidY;
          acc += inter[index1] * dct8x8[index2];
        }
        output[tidY*width + tidX] = acc;
}   } }
```

Group Kernel (2D, Scalar, Inlined, Optimized (1))

```
void groupKernel(TA argStruct, int* groupIDs,
                 int* globalSizes, int* localSizes)
{
    for (int lidY=0; lidY<localSizes[1]; ++lidY) {

      for (int lidX=0; lidX<localSizes[0]; ++lidX) {
        uint tidX = localSizes[0] * groupIDs[0] + lidX;
        ...
        inter[lidY*blockWidth + lidX] = ...
        barrier(CLK_LOCAL_MEM_FENCE);

        float acc = 0.0f;
        for(uint k=0; k < blockWidth; k++)
        {
          uint index1 = lidX*blockWidth + k;
          uint index2 = (inverse) ? lidY*blockWidth + k :
                                    k*blockWidth + lidY;
          acc += inter[index1] * dct8x8[index2];
        }
        output[tidY*width + tidX] = acc;
}   } }
```

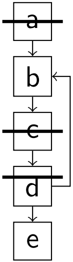
Group Kernel (2D, Scalar, Inlined, Optimized (2))

```
void groupKernel(TA argStruct, int* groupIDs,
                 int* globalSizes, int* localSizes)
{
    for (int lidY=0; lidY<localSizes[1]; ++lidY) {
      uint LIC = lidY*blockWidth;
      for (int lidX=0; lidX<localSizes[0]; ++lidX) {
        uint tidX = localSizes[0] * groupIDs[0] + lidX;
        ...
        inter[LIC + lidX] = ...
        barrier(CLK_LOCAL_MEM_FENCE);

        float acc = 0.0f;
        for(uint k=0; k < blockWidth; k++)
        {
          uint index1 = lidX*blockWidth + k;
          uint index2 = (inverse) ? LIC + k :
                                    k*blockWidth + lidY;
          acc += inter[index1] * dct8x8[index2];
        }
        output[tidY*width + tidX] = acc;
}   } }
```

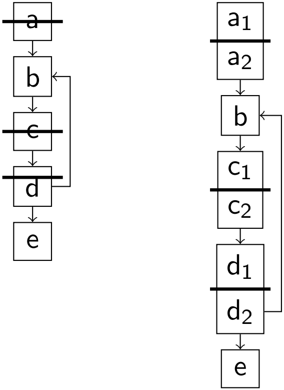
Barrier Synchronization

```
void groupKernel(TA argStruct, int* groupIDs,
                 int* globalSizes, int* localSizes)
{
    for (int lidY=0; lidY<localSizes[1]; ++lidY) {
      uint LIC = lidY*blockWidth;
      for (int lidX=0; lidX<localSizes[0]; ++lidX) {
        uint tidX = localSizes[0] * groupIDs[0] + lidX;
        ...
        inter[LIC + lidX] = ...
        barrier(CLK_LOCAL_MEM_FENCE);

        float acc = 0.0f;
        for(uint k=0; k < blockWidth; k++)
        {
          uint index1 = lidX*blockWidth + k;
          uint index2 = (inverse) ? LIC + k :
                                    k*blockWidth + lidY;
          acc += inter[index1] * dct8x8[index2];
        }
        output[tidY*width + tidX] = acc;
}   } }
```

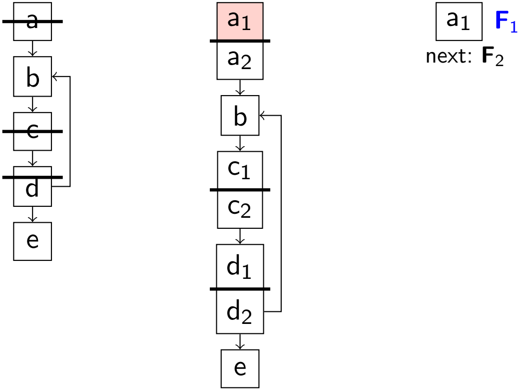
Barrier Synchronization: Example



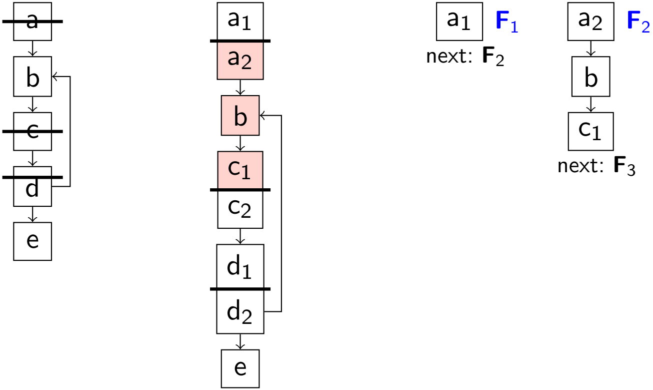
Barrier Synchronization: Example



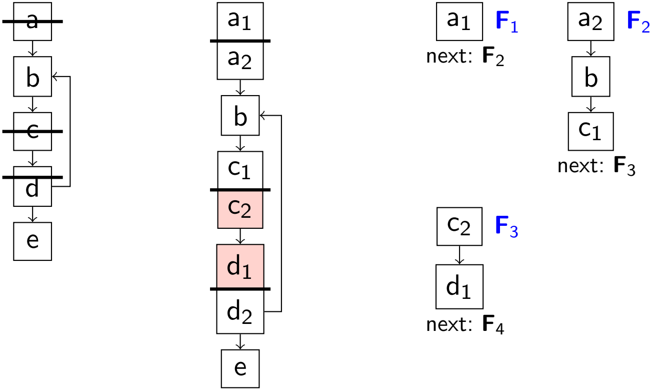
Barrier Synchronization: Example



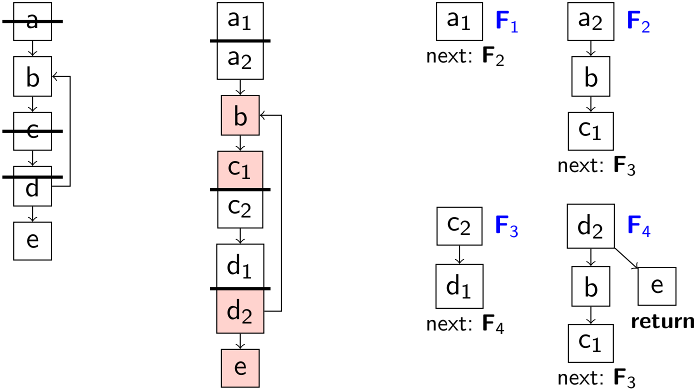
Barrier Synchronization: Example



Barrier Synchronization: Example



Barrier Synchronization: Example



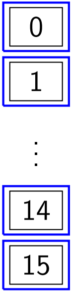
Group Kernel (1D, Scalar, Barrier Synchronization)

```
void groupKernel(TA argStruct, int groupID,
                 int globalSizes, int localSize, ...)
{
    void* data[localSize] = alloc(localSize*liveValSize);

    int next = BARRIER_BEGIN;
    while (true) {
      switch (next) {
        case BARRIER_BEGIN:
          for (int i=0; i<localSize; ++i)
            next = F1(argStruct, tid, ..., &data[i]); // B2
          break;
        ...
        case B4:
          for (int i=0; i<localSize; ++i)
            next = F4(tid, ..., &data[i]); // B3 or END
          break;
        case BARRIER_END: return;
}   } }
```

OpenCL: Exploiting Parallelism on CPUs

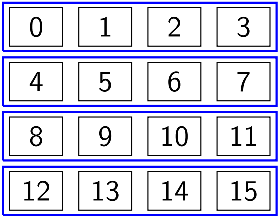
CPU (1 core):

All threads run sequentially



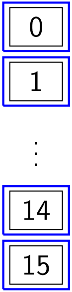
CPU (4 cores):

Each core executes 1 thread

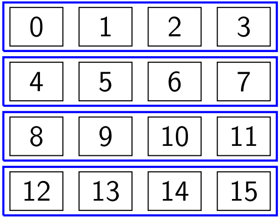


OpenCL: Exploiting Parallelism on CPUs

CPU (1 core):
All threads run sequentially



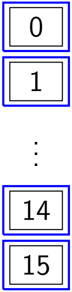
CPU (4 cores):
Each core executes 1 thread



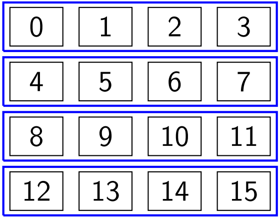
CPU (4 cores, SIMD width 4): Each core executes 4 threads

OpenCL: Exploiting Parallelism on CPUs

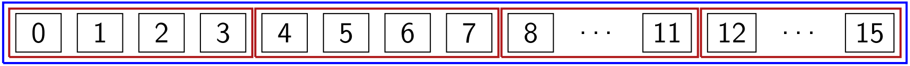
CPU (1 core):
All threads run sequentially



CPU (4 cores):
Each core executes 1 thread



CPU (4 cores, SIMD width 4): Each core executes 4 threads

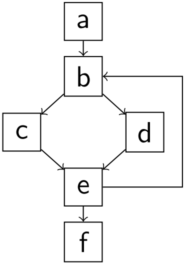


Group Kernel (2D, SIMD)

```
void groupKernel(TA argStruct, int* groupIDs,
                 int* globalSizes, int* localSizes)
{
    for (int lidY=0; lidY<localSizes[1]; ++lidY) {
      for (int lidX=0; lidX<localSizes[0]; lidX+=4) {
        __m128i lidXV = <lidX,lidX+1,lidX+2,lidX+3>;
        simdKernel  (argStruct, lidXV, lidY,
                     groupIDs, globalSizes,
                     localSizes); // to be inlined
}   } }
```

- Whole-Function Vectorization (WFV) of kernel code
- New kernel computes 4 "threads" at once using SIMD instruction set
- Challenge: diverging control flow

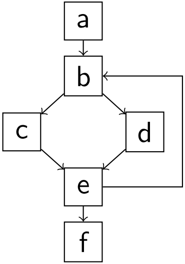
Diverging Control Flow

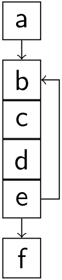


Thread	Trace
1	a b c e f
2	a b d e f
3	a b c e b c e f
4	a b c e b d e f

- Different threads execute different code paths

Diverging Control Flow

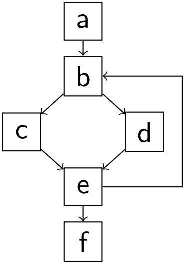


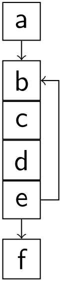


Thread	Trace
1	a b c d e b c d e f
2	a b c d e b c d e f
3	a b c d e b c d e f
4	a b c d e b c d e f

- Different threads execute different code paths
- Execute everything, mask out results of inactive threads (using predication, blending)
- Control flow to data flow conversion on ASTs [Allen et al. POPL'83]
- Whole-Function Vectorization on SSA CFGs [K & H CGO'11]

Diverging Control Flow





Thread	Trace
1	a b c d e b c d e f
2	a b c d e b c d e f
3	a b c d e b c d e f
4	a b c d e b c d e f

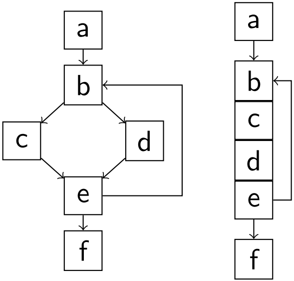
- Overhead for maintaining & updating of predicates
- Overhead for operations with side-effects (e.g. load/store/call)
- Expensive but rarely executed paths are now always executed
- Linearization increases register pressure ☞ more spilling
- Works well for kernels with mostly straight-line code

DCT Kernel: Non-Divergent Control Flow

```
__kernel void DCT(__global float * output,
                  __global float * input,
                  __global float * dct8x8,
                  __local  float * inter,
                  const    uint    width,
                  const    uint    blockWidth,
                  const    uint    inverse)
{
    uint tidX = get_global_id(0);
    ...
    inter[lidY*blockWidth + lidX] = ...
    barrier(CLK_LOCAL_MEM_FENCE);

    float acc = 0.0f;
    for(uint k=0; k < blockWidth; k++)
    {
        uint index1 = lidX*blockWidth + k;
        uint index2 = (inverse) ? lidY*blockWidth + k :
                                  k*blockWidth + lidY;
        acc += inter[index1] * dct8x8[index2];
    }
    output[tidY*width + tidX] = acc;
} // Compiled to LLVM bitcode.
```

Non-Divergent Control Flow

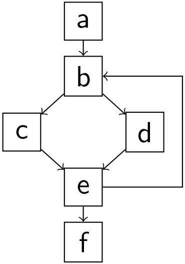
- Idea: optimize cases where threads do not diverge

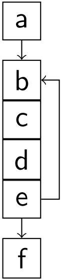


Thread	Trace
1	a b c e f
2	a b c e f
3	a b c e b d e f
4	a b c e b d e f

Non-Divergent Control Flow

- Idea: optimize cases where threads do not diverge

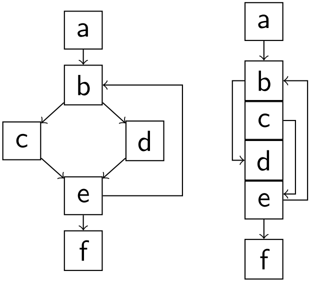




Thread	Trace
1	a b c e b d e f
2	a b c e b d e f
3	a b c e b d e f
4	a b c e b d e f

Non-Divergent Control Flow

- Idea: optimize cases where threads do not diverge

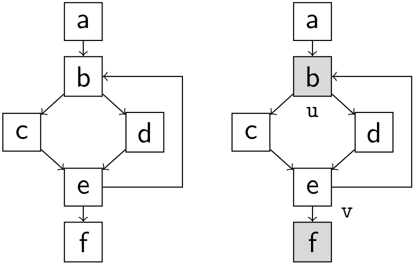


Thread	Trace
1	a b c e b d e f
2	a b c e b d e f
3	a b c e b d e f
4	a b c e b d e f

- Option 1: Insert dynamic predicate-tests & branches to skip paths
  - "Branch on superword condition code" (BOSCC) [Shin et al. PACT'07]
  - Additional overhead for dynamic test
  - Does not help against increased register pressure

Non-Divergent Control Flow

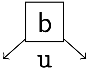
- Idea: optimize cases where threads do not diverge

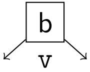


Thread	Trace
1	a b c e b d e f
2	a b c e b d e f
3	a b c e b d e f
4	a b c e b d e f

- Option 2: Statically prove non-divergence of certain blocks
  - Non-divergent blocks can be excluded from linearization
  - Less executed code, less register pressure
  - More conservative than dynamic test ☞ exploit both!

Uniform/Varying Branches





Either all threads entering b go left or right

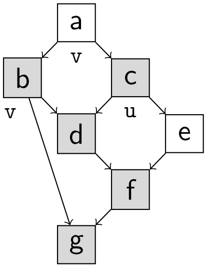
```
if (blockWidth % 2 == 0)
{
     ...
}

for(uint k=0; k < blockWidth; k++)
{
     ...
}
```

From the $p+q$ threads entering b, p go left, q go right

```
if (tid % 2 == 0)
{
     ...
}

for(uint k=0; k < tid; k++)
{
     ...
}
```

When does a block diverge?

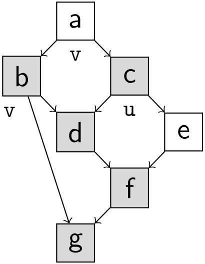
Informally



A block *b* is divergent if:

- *b* might execute less (not provably 0) threads than its predecessor.
  That is: it is a successor of a varying branch
- Two disjoint paths from the same varying branch rejoin at *b*
- (Additional criterion for loops)

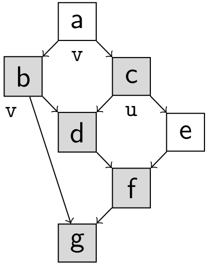
CFG Linearization w/ Non-Divergent Blocks: Example

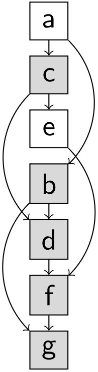


(a)

(a) Original CFG

CFG Linearization w/ Non-Divergent Blocks: Example

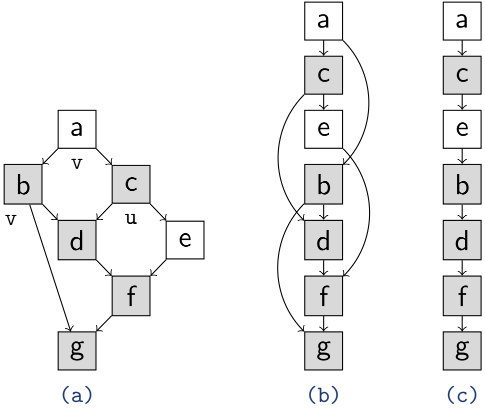




(a) Original CFG

(b) Topological order (by data dependencies)

CFG Linearization w/ Non-Divergent Blocks: Example

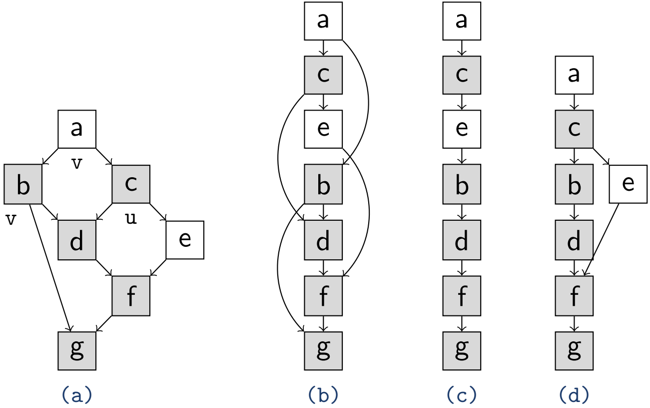


(a) Original CFG

(b) Topological order (by data dependencies)

(c) Naïve: Rewire all edges to next block ☞ all blocks are always executed

CFG Linearization w/ Non-Divergent Blocks: Example



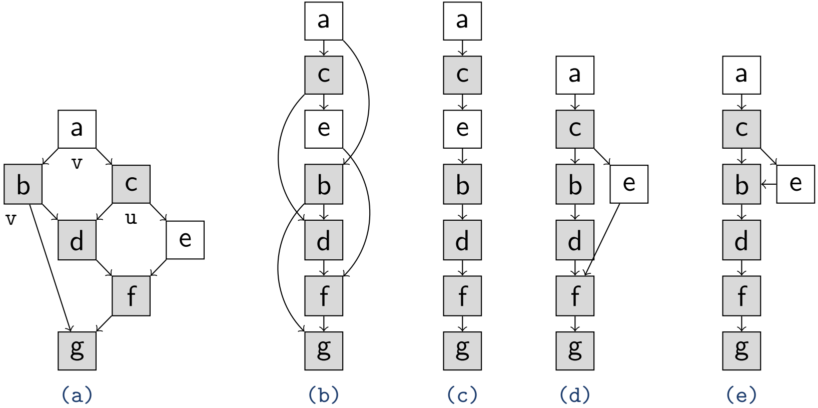
(a) Original CFG

(b) Topological order (by data dependencies)

(c) Naïve: Rewire all edges to next block ☞ all blocks are always executed

(d) Invalid: Edges to/from non-divergent block remain ☞ `b` and `d` can be skipped

CFG Linearization w/ Non-Divergent Blocks: Example



(a) Original CFG

(b) Topological order (by data dependencies)

(c) Naïve: Rewire all edges to next block ☞ all blocks are always executed

(d) Invalid: Edges to/from non-divergent block remain ☞ b and d can be skipped

(e) Valid: Rewire edges to divergent blocks to next in list ☞ only e can be skipped

Evaluation I: WFV vs. Sequential Execution Comparison

Application	Naïve	UniVal	BOSCC	UniCF
BitonicSort	3.0	3.2	3.3	3.2
BlackScholes	3.9	4.1	4.1	4.1
DCT	0.67	0.85	0.85	1.78
FastWalshTransform	0.74	0.73	0.73	0.73
FloydWarshall	0.11	0.12	0.13	0.12
Histogram	0.92	1.08	1.07	1.24
Mandelbrot	0.51	2.4	2.4	2.4
MatrixTranspose	0.97	1.44	1.44	1.44
NBody	1.8	2.67	2.67	3.64
AVG	1.4	1.84	1.85	2.07

- SIMD width 4, median of 100 iterations, no warm-up, confidence level 95%

Evaluation I: WFV vs. Sequential Execution Comparison

Application	Naïve	UniVal	BOSCC	UniCF
BitonicSort	3.0	3.2	3.3	3.2
BlackScholes	3.9	4.1	4.1	4.1
DCT	0.67	0.85	0.85	1.78
FastWalshTransform	0.74	0.73	0.73	0.73
FloydWarshall	0.11	0.12	0.13	0.12
Histogram	0.92	1.08	1.07	1.24
Mandelbrot	0.51	2.4	2.4	2.4
MatrixTranspose	0.97	1.44	1.44	1.44
NBody	1.8	2.67	2.67	3.64
AVG	1.4	1.84	1.85	2.07

- SIMD width 4, median of 100 iterations, no warm-up, confidence level 95%
- Naïve WFV is often inferior to sequential execution

Evaluation I: WFV vs. Sequential Execution Comparison

Application	Naïve	UniVal	BOSCC	UniCF
BitonicSort	3.0	3.2	3.3	3.2
BlackScholes	3.9	4.1	4.1	4.1
DCT	0.67	0.85	0.85	1.78
FastWalshTransform	0.74	0.73	0.73	0.73
FloydWarshall	0.11	0.12	0.13	0.12
Histogram	0.92	1.08	1.07	1.24
Mandelbrot	0.51	2.4	2.4	2.4
MatrixTranspose	0.97	1.44	1.44	1.44
NBody	1.8	2.67	2.67	3.64
AVG	1.4	1.84	1.85	2.07

- SIMD width 4, median of 100 iterations, no warm-up, confidence level 95%
- Naïve WFV is often inferior to sequential execution
- Dynamic analysis (BOSCC) has almost no effect for these benchmarks

Evaluation I: WFV vs. Sequential Execution Comparison

Application	Naïve	UniVal	BOSCC	UniCF
BitonicSort	3.0	3.2	3.3	3.2
BlackScholes	3.9	4.1	4.1	4.1
DCT	0.67	0.85	0.85	1.78
FastWalshTransform	0.74	0.73	0.73	0.73
FloydWarshall	0.11	0.12	0.13	0.12
Histogram	0.92	1.08	1.07	1.24
Mandelbrot	0.51	2.4	2.4	2.4
MatrixTranspose	0.97	1.44	1.44	1.44
NBody	1.8	2.67	2.67	3.64
AVG	1.4	1.84	1.85	2.07

- SIMD width 4, median of 100 iterations, no warm-up, confidence level 95%
- Naïve WFV is often inferior to sequential execution
- Dynamic analysis (BOSCC) has almost no effect for these benchmarks
- Static analysis (UniCF) is beneficial for suitable kernels

Evaluation I: WFV vs. Sequential Execution Comparison

Application	Naïve	UniVal	BOSCC	UniCF
BitonicSort	3.0	3.2	3.3	3.2
BlackScholes	3.9	4.1	4.1	4.1
DCT	0.67	0.85	0.85	1.78
FastWalshTransform	0.74	0.73	0.73	0.73
FloydWarshall	0.11	0.12	0.13	0.12
Histogram	0.92	1.08	1.07	1.24
Mandelbrot	0.51	2.4	2.4	2.4
MatrixTranspose	0.97	1.44	1.44	1.44
NBody	1.8	2.67	2.67	3.64
AVG	1.4	1.84	1.85	2.07

- SIMD width 4, median of 100 iterations, no warm-up, confidence level 95%
- Naïve WFV is often inferior to sequential execution
- Dynamic analysis (BOSCC) has almost no effect for these benchmarks
- Static analysis (UniCF) is beneficial for suitable kernels
- Kernels dominated by random memory access are not suited for WFV

Evaluation II: WFVOpenCL vs. Intel/AMD (milliseconds)

Application	WFVOpenCL	Intel	AMD	Speedup vs Intel
BitonicSort	164	1,170	47,271	7.13×
BlackScholes	241	329	717	1.37×
DCT	201	350	693	1.74×
FastWalshTransform	4,944	6,661	8,601	1.35×
FloydWarshall	934(148*)	525*	471	0.56×(3.55×*)
Histogram	387	1,178	527	3.07×
Mandelbrot	632	1,930	29,045	3.05×
MatrixTranspose	1,072	2,933	10,748	2.74×
NBody	343	676	1,253	1.97×

- 4 cores, SIMD width 4, median of 100 iterations, no warm-up, confidence level 95%
- Intel OpenCL SDK v1.1 / AMD APP SDK v2.5
- Average speedup: 2.5× (Intel), 40× (AMD)
- *WFV disabled – Intel driver does not vectorize FloydWarshall

LLVM: Benefits and Drawbacks

- We heavily rely on JIT code generator ☞ no disappointment!
- LLVM IR allows convenient expression of vector computations
  - Vector-select and type legalization
- MOVMASK still requires an intrinsic
- Would be great: a way to express predication in IR

Outlook

- More optimizations for WFV
- Integration of WFV into LLVM mainline?
  - Should integrate nicely with Hal's BasicBlock vectorization
  - Combine with loop dependency analysis / Polly for "classic" loop vectorization
- Support for architectures w/ predicated execution (e.g. LRBni)

Conclusion

- OpenCL benefits from "group kernel"-based implementation:
  - Optimize uniform expressions & access to `tid` etc.
  - Enable continuation-based barrier synchronization
- OpenCL benefits from both multi-threading and WFV on CPUs
- Divergence analysis improves WFV:
  - Reduce amount of executed code
  - Reduce register pressure
  - Reduce overhead for maintaining & updating of predicates
- Evaluation shows importance of advanced vectorization techniques
- Sources available: https://github.com/karrenberg

Conclusion

- OpenCL benefits from "group kernel"-based implementation:
  - Optimize uniform expressions & access to `tid` etc.
  - Enable continuation-based barrier synchronization
- OpenCL benefits from both multi-threading and WFV on CPUs
- Divergence analysis improves WFV:
  - Reduce amount of executed code
  - Reduce register pressure
  - Reduce overhead for maintaining & updating of predicates
- Evaluation shows importance of advanced vectorization techniques
- Sources available: https://github.com/karrenberg

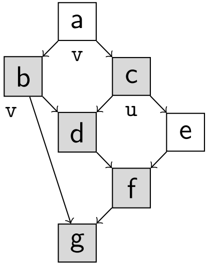
Thank You!

Questions?

CFG Linearization w/ Non-Divergent Blocks

- Combine divergent blocks to divergent regions with DFS:
  - Non-uniform branch found: create new region, set as active
  - Post-dominator of region found: finish region, set last unfinished one as active
  - Add divergent blocks to active region
  - Merge overlapping regions
- Linearize regions recursively (inner before outer regions):
  - Order blocks topologically by data dependencies (inner regions treated as single blocks)
  - Schedule blocks in this order by visiting all outgoing edges:
    - ⋆ Rewire all edges that target a divergent block
    - ⋆ New target: next divergent, unscheduled block of region

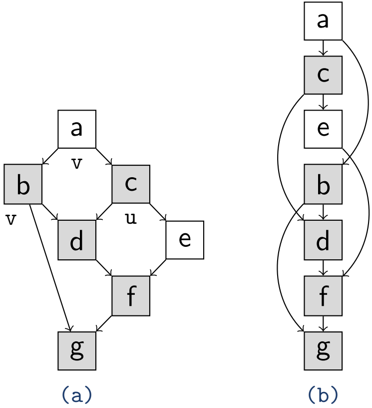
CFG Linearization w/ Non-Divergent Blocks: Example



(a)

(a) Original CFG

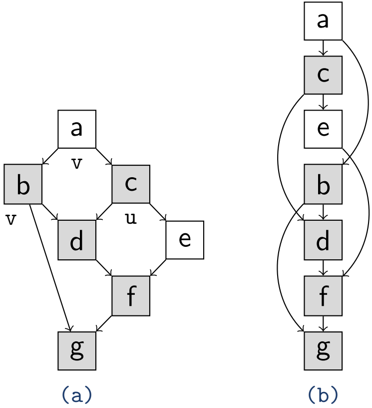
CFG Linearization w/ Non-Divergent Blocks: Example



(a) Original CFG

(b) Topological order (by data dependencies)

CFG Linearization w/ Non-Divergent Blocks: Example

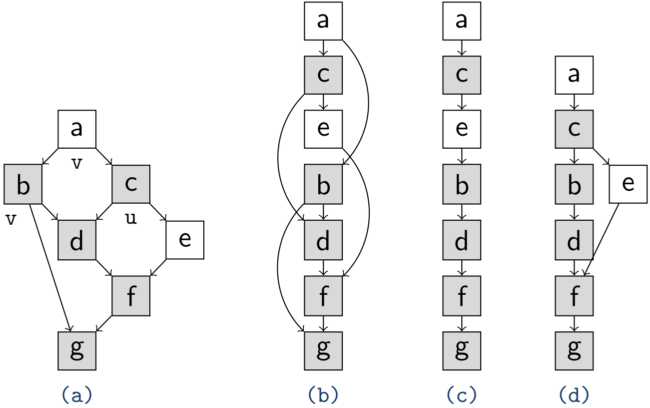


(a) Original CFG

(b) Topological order (by data dependencies)

(c) Naïve: Rewire all edges to next block ☞ all blocks are always executed

CFG Linearization w/ Non-Divergent Blocks: Example



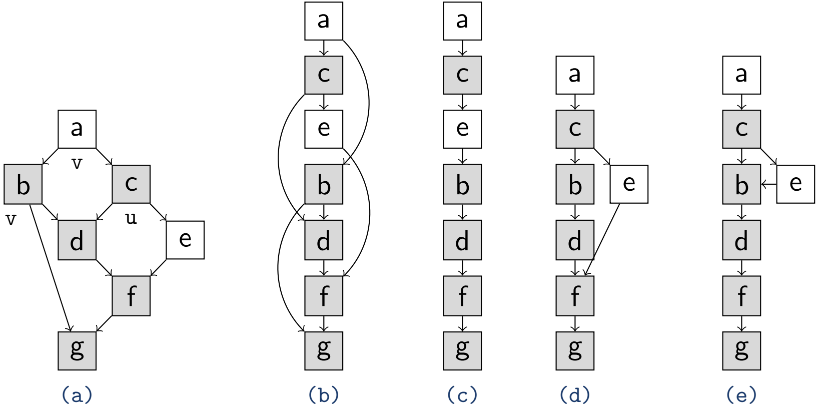
(a) Original CFG

(b) Topological order (by data dependencies)

(c) Naïve: Rewire all edges to next block ☞ all blocks are always executed

(d) Invalid: Edges to/from non-divergent block remain ☞ `b` and `d` can be skipped

CFG Linearization w/ Non-Divergent Blocks: Example



(a) Original CFG

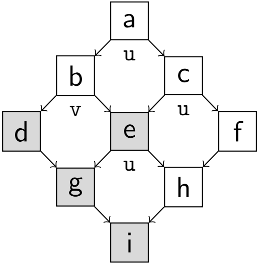
(b) Topological order (by data dependencies)

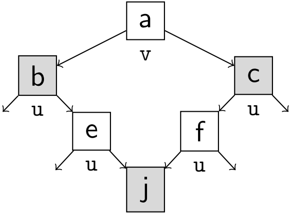
(c) Naïve: Rewire all edges to next block ☞ all blocks are always executed

(d) Invalid: Edges to/from non-divergent block remain ☞ `b` and `d` can be skipped

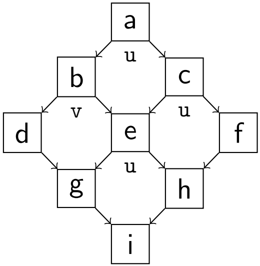
(e) Valid: Rewire edges to divergent blocks to next in list ☞ only `e` can be skipped

Examples

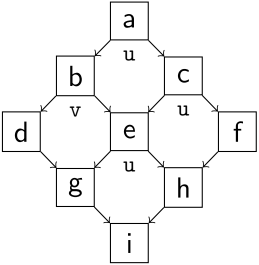


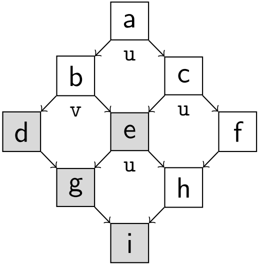


Retaining Control Flow: Complex Example

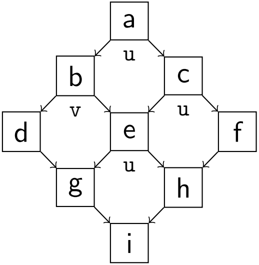


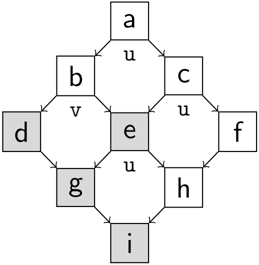
Retaining Control Flow: Complex Example

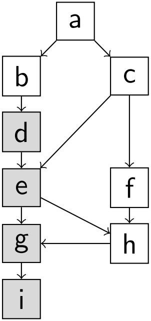




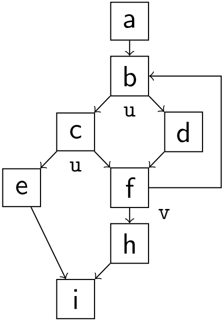
Retaining Control Flow: Complex Example



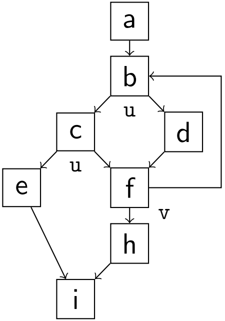


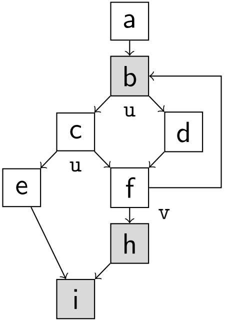


Retaining Control Flow: Loop Example



Retaining Control Flow: Loop Example





Retaining Control Flow: Loop Example