Whole-Function Vectorization

Ralf Karrenberg
karrenberg@cs.uni-saarland.de

Sebastian Hack
hack@cs.uni-saarland.de

CGO 2011, Chamonix
April 5, 2011

UNIVERSITÄT DES SAARLANDES

CLUSTER OF EXCELLENCE

VISUAL COMPUTING INSTITUTE

Data-Parallel Languages

- Data-parallel languages become more and more popular
  - E.g. OpenCL, CUDA
- Used for a long time in domain-specific environments (e.g. graphcis):
  - RenderMan, Cg, glsl, ...
- Data-parallel execution model
  - Execute one function (called kernel) on n inputs
  - n threads of the same code
  - Order of threads unspecified, can run in parallel
  - Programmer can use barrier synchronization across threads
  - Threads can query their thread id

Our Contribution

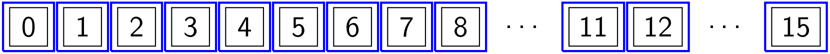
An algorithm to implement the data-parallel execution model for SIMD architectures on arbitrary control flow graphs in SSA form.

Data-Parallel Languages: OpenCL Example

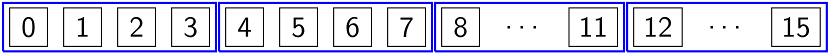
```
__kernel void fastWalshTransform(
        __global float * tArray,
        __const  int    step
)
{
    unsigned int tid = get_global_id(0);

    const unsigned int group = tid%step;
    const unsigned int pair  = 2*step*(tid/step) + group;
    const unsigned int match = pair + step;

    float T1 = tArray[pair];
    float T2 = tArray[match];

    tArray[pair]  = T1 + T2;
    tArray[match] = T1 - T2;
}
```

Data-Parallel Execution Model: Example

CPU (1 core): All threads run sequentially

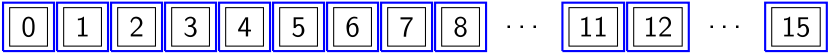


CPU (4 cores): Each core executes 1 thread

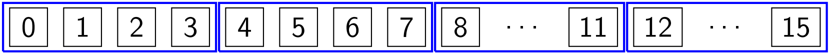


Data-Parallel Execution Model: Example

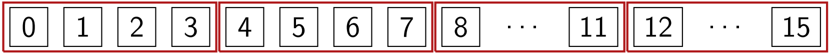
CPU (1 core): All threads run sequentially



CPU (4 cores): Each core executes 1 thread

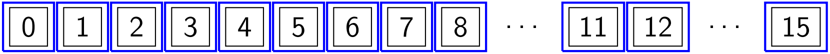


CPU (4 cores, SIMD width 4): Each core executes 4 threads

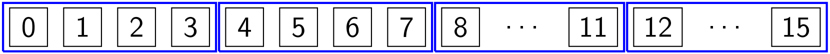


Data-Parallel Execution Model: Example

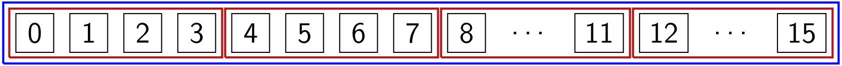
CPU (1 core): All threads run sequentially



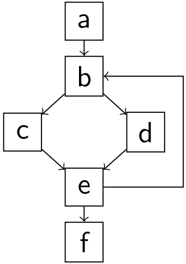
CPU (4 cores): Each core executes 1 thread



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



Diverging Control-Flow

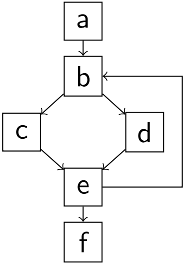


Thread	Trace
1	a b c e f
2	a b d e f
3	a b c e b c e f
4	a b c e b d e f

- Different threads execute different code paths

[1] Allen et al.: "Conversion of Control Dependence To Data Dependence", *POPL '83*

Diverging Control-Flow

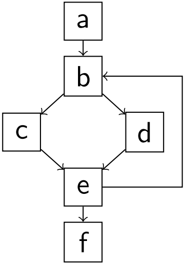


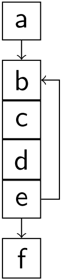
Thread	Trace
1	a b c d e b c d e f
2	a b c d e b c d e f
3	a b c d e b c d e f
4	a b c d e b c d e f

- Different threads execute different code paths
- If merged into one SIMD thread, predication is required

[1] Allen et al.: "Conversion of Control Dependence To Data Dependence", *POPL '83*

Diverging Control-Flow



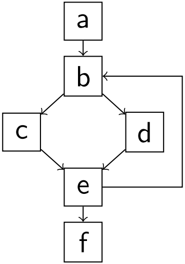


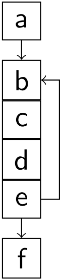
Thread	Trace
1	a b c d e b c d e f
2	a b c d e b c d e f
3	a b c d e b c d e f
4	a b c d e b c d e f

- Different threads execute different code paths
- If merged into one SIMD thread, predication is required
- Execute all code, mask out results of inactive threads [1]

[1] Allen et al.: "Conversion of Control Dependence To Data Dependence", *POPL '83*

Diverging Control-Flow



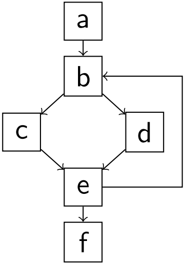


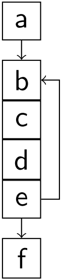
Thread	Trace
1	a b c d e b c d e f
2	a b c d e b c d e f
3	a b c d e b c d e f
4	a b c d e b c d e f

- Different threads execute different code paths
- If merged into one SIMD thread, predication is required
- Execute all code, mask out results of inactive threads [1]
  - Known as if-conversion

[1] Allen et al.: "Conversion of Control Dependence To Data Dependence", *POPL '83*

Diverging Control-Flow





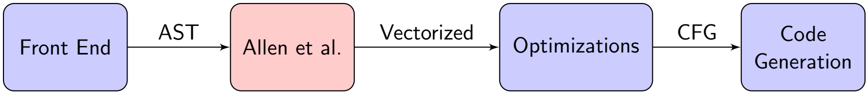
Thread	Trace
1	a b c d e b c d e f
2	a b c d e b c d e f
3	a b c d e b c d e f
4	a b c d e b c d e f

- Different threads execute different code paths
- If merged into one SIMD thread, predication is required
- Execute all code, mask out results of inactive threads [1]
  - Known as if-conversion
  - Use hardware (predicated execution) or mask out manually

[1] Allen et al.: "Conversion of Control Dependence To Data Dependence", *POPL '83*

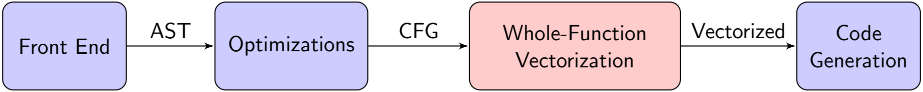
Allen et al.: Control-Flow to Data-Flow Conversion

- Conversion is performed on abstract syntax trees
  - Re-implement it in every front end (language) you want to compile
    - Language-dependent
  - Predicated vector code disturbs common scalar optimizations
    - Control flow is gone
    - Some optimizations not possible anymore (e.g. PRE)
    - Some optimizations confused by vector operations
- All related work on domain-specific languages is AST-based



Our Setting

- We load LLVM bitcode at the runtime of the system
  - Low-level SSA code with control flow graphs
  - Language-independent
  - Leverage scalar optimizations before vectorization



Whole-Function Vectorization: Main Phases

1. Preparatory transformations
2. Vectorization analysis
3. Mask generation
4. Select generation
5. CFG linearization
6. Instruction vectorization

Phase II: Vectorization Analysis

- Memory operations: conservatively have to be split into W guarded scalar operations (scatter/gather)

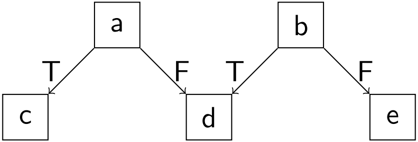
- Attempt to exploit fast SIMD load/store instructions

- Mark instructions that result in *aligned* indices (e.g. [0, 1, 2, 3])
  - Single vector load/store

- Mark instructions that result in *consecutive* indices (e.g. [6, 7, 8, 9])
  - Unaligned load

- Mark instructions that are *uniform* across all threads (e.g. [4, 4, 4, 4])
  - CFG regions marked as *uniform* can be executed in scalar unit

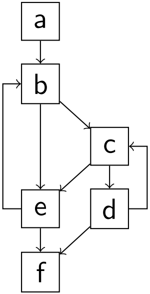
Phases III, IV: Mask & Select Generation



- Mask generation encodes control flow in masks:
  - Mask(c) = mask(a) $\wedge$ condition(a)
  - Mask(d) = mask(a) $\wedge$ $\neg$condition(a) $\vee$ mask(b) $\wedge$ condition(b)
  - Mask(e) = mask(b) $\wedge$ $\neg$condition(b)
- Select generation introduces select operations
  - Create new vector from two incoming ones with appropriate mask

Example: Nested Multi-Exit Loop

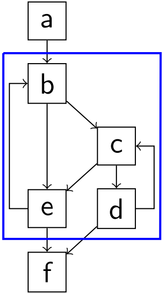
```
float f(float x, float y) {
    float r = x * y;
    for (int i=0; i<x; ++i) {
        for (int j=0; j<y; ++j) {
            --r;
            if (r < 7) goto END;
        }
    }
END:
    return r;
}
```



- Iterate until all threads have left the loop
- Keep track of active & inactive threads
- Remember which thread left through which exit
- Naive: mask out after each operation
- WFV: need only one operation per live value per nested loop

Example: Nested Multi-Exit Loop

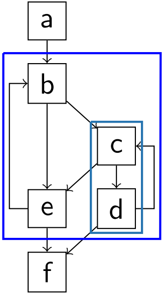
```
float f(float x, float y) {
    float r = x * y;
    for (int i=0; i<x; ++i) {
        for (int j=0; j<y; ++j) {
            --r;
            if (r < 7) goto END;
        }
    }
END:
    return r;
}
```



- Iterate until all threads have left the loop
- Keep track of active & inactive threads
- Remember which thread left through which exit
- Naive: mask out after each operation
- WFV: need only one operation per live value per nested loop

Example: Nested Multi-Exit Loop

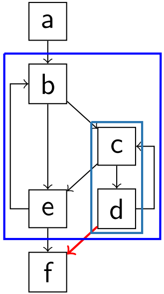
```
float f(float x, float y) {
    float r = x * y;
    for (int i=0; i<x; ++i) {
        for (int j=0; j<y; ++j) {
            --r;
            if (r < 7) goto END;
        }
    }
END:
    return r;
}
```



- Iterate until all threads have left the loop
- Keep track of active & inactive threads
- Remember which thread left through which exit
- Naive: mask out after each operation
- WFV: need only one operation per live value per nested loop

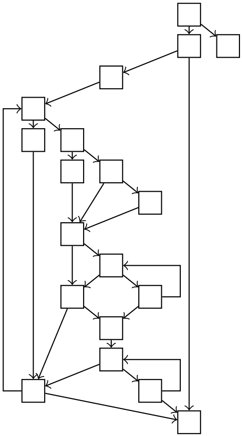
Example: Nested Multi-Exit Loop

```
float f(float x, float y) {
    float r = x * y;
    for (int i=0; i<x; ++i) {
        for (int j=0; j<y; ++j) {
            --r;
            if (r < 7) goto END;
        }
    }
END:
    return r;
}
```



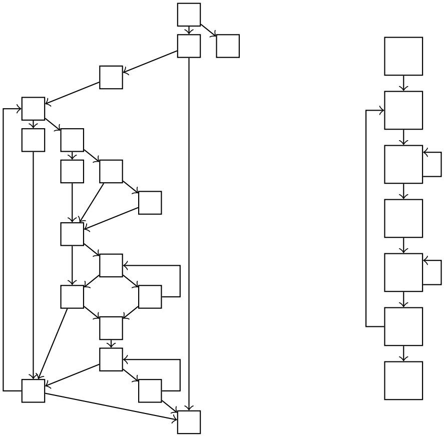
- Iterate until all threads have left the loop
- Keep track of active & inactive threads
- Remember which thread left through which exit
- Naive: mask out after each operation
- WFV: need only one operation per live value per nested loop

Phase V: CFG Linearization



[2]Shin et al.: "Introducing Control Flow into Vectorized Code", *PACT '07*

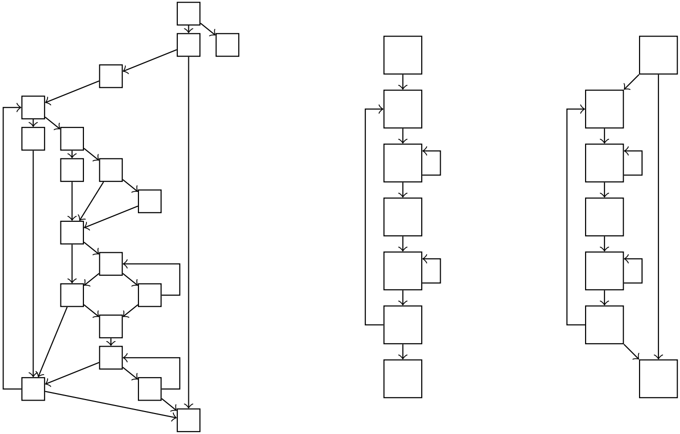
Phase V: CFG Linearization



- Remove all control-flow except for back-branches of loops

[2] Shin et al.: "Introducing Control Flow into Vectorized Code", *PACT '07*

Phase V: CFG Linearization



- Remove all control-flow except for back-branches of loops
- Insert dynamic mask-tests & branches to skip entire paths [2]

[2] Shin et al.: "Introducing Control Flow into Vectorized Code", *PACT '07*

Evaluation I: Vectorized RenderMan Materials

Material	Scalar (fps)	Vectorized (fps)	Speedup
Brick	8.8	31.4	3.6x
Checker	8.8	31.8	3.6x
Glass	0.9	4.5	5.0x
Granite	7.2	24.6	3.4x
Parquet	4.3	18.6	4.3x
Phong	14.1	32.5	2.3x
Screen	4.6	22.7	4.9x
Starball	4.5	20.0	4.4x
Venus	7.6	25.7	3.4x
Wood	4.4	19.1	4.3x
Average	6.5	23.3	3.9x

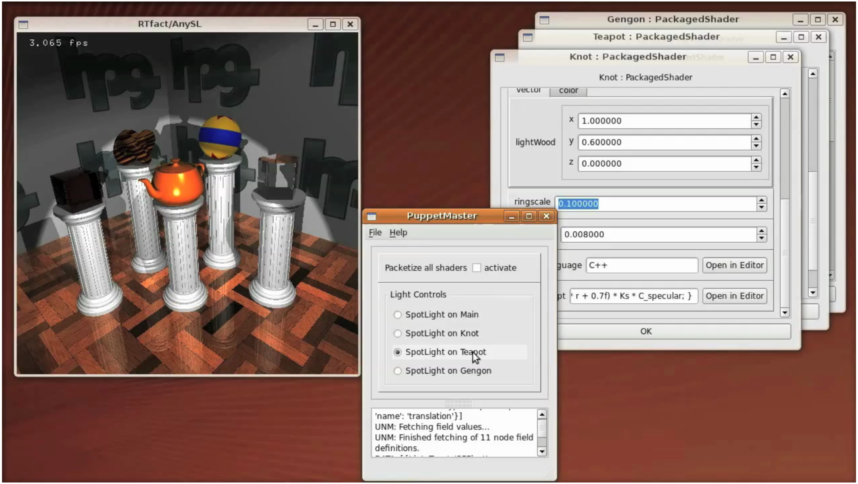
- Performance of SIMD ray tracer in frames per second (fps)
- SIMD width 4
- Material = function that computes colors of an object
- Big impact due to frequent execution

Evaluation I: Vectorized RenderMan Materials

Material	Scalar (fps)	Vectorized (fps)	Speedup
Brick	8.8	31.4	3.6x
Checker	8.8	31.8	3.6x
Glass	0.9	4.5	5.0x
Granite	7.2	24.6	3.4x
Parquet	4.3	18.6	4.3x
Phong	14.1	32.5	2.3x
Screen	4.6	22.7	4.9x
Starball	4.5	20.0	4.4x
Venus	7.6	25.7	3.4x
Wood	4.4	19.1	4.3x
Average	6.5	23.3	3.9x

- Performance of SIMD ray tracer in frames per second (fps)
- SIMD width 4
- Material = function that computes colors of an object
- Big impact due to frequent execution

Vectorized RenderMan Materials: Demonstration



Evaluation II: Vectorized OpenCL Kernels

Application	AMD (ms)	Scalar (ms)	Vectorized (ms)	Speedup
AOBench	23520	37037	24390	1.5x
BlackScholes	280	13	2.4	5.2x
FastWalshTransform	320	80	100	0.8x
Histogram	480	410	710	0.6x
Mandelbrot	291200	4000	1800	2.2x
NBody	200	160	57	2.8x
MatrixTranspose	17600	1220	900	1.4x

- Custom OpenCL CPU driver
- Benchmarks from AMD-ATI StreamSDK
- Single-thread performance, SIMD width 4, average over 100 iterations

Evaluation II: Vectorized OpenCL Kernels

Application	AMD (ms)	Scalar (ms)	Vectorized (ms)	Speedup
AOBench	23520	37037	24390	1.5x
BlackScholes	280	13	2.4	5.2x
FastWalshTransform	320	80	100	0.8x
Histogram	480	410	710	0.6x
Mandelbrot	291200	4000	1800	2.2x
NBody	200	160	57	2.8x
MatrixTranspose	17600	1220	900	1.4x

- Custom OpenCL CPU driver
- Benchmarks from AMD-ATI StreamSDK
- Single-thread performance, SIMD width 4, average over 100 iterations
- Improvement for compute-intensive kernels

Evaluation II: Vectorized OpenCL Kernels

Application	AMD (ms)	Scalar (ms)	Vectorized (ms)	Speedup
AOBench	23520	37037	24390	1.5x
BlackScholes	280	13	2.4	5.2x
FastWalshTransform	320	80	100	0.8x
Histogram	480	410	710	0.6x
Mandelbrot	291200	4000	1800	2.2x
NBody	200	160	57	2.8x
MatrixTranspose	17600	1220	900	1.4x

- Custom OpenCL CPU driver
- Benchmarks from AMD-ATI StreamSDK
- Single-thread performance, SIMD width 4, average over 100 iterations
- Improvement for compute-intensive kernels
- Performance loss for kernels dominated by random memory accesses

Conclusion

- Whole-Function Vectorization exploits data-level parallelism with SIMD instructions
- Targeted at data-parallel languages
- SSA-based, works on any CFG
- Language-independent
- Vectorization analysis helps reducing overhead
- Evaluation shows applicability to real-world scenarios

Conclusion

- Whole-Function Vectorization exploits data-level parallelism with SIMD instructions
- Targeted at data-parallel languages
- SSA-based, works on any CFG
- Language-independent
- Vectorization analysis helps reducing overhead
- Evaluation shows applicability to real-world scenarios

Thank You!

Questions?