Creating an SPMD Vectorizer for OpenCL with LLVM

LLVM 2015 Tutorial

Pierre-André Saulais
<pierre-andre@codeplay.com>

Codeplay Software
@codeplaysoft

October 29, 2015

What this tutorial is about

- Vectorizing
  - Transform whole functions using LLVM [1, 2]
  - "Horizontal" vectorization (not loop-based)
- SPMD (Single Program, Multiple Data) Kernels
  - Data-parallel execution model
  - Compute frameworks like OpenCL™ and CUDA™
- for CPUs, DSPs,
  - Explicitly programmed SIMD unit(s)
  - Can execute both scalar and vector instructions
- An introduction
  - Introduce concepts needed to create a basic vectorizer
  - Starting point, not finished product

Overview

Part 1: Background

- SPMD Execution Model
- Vectorization

Part 2: Implementing a SPMD Vectorizer

- Overview
- Packetization Stage
- Scalarization Stage
- Control Flow Conversion Stage

Part 1

Background

Part 1: Background

- SPMD Execution Model
- Vectorization

SPMD Execution Model

- Single program
  - Scalar form, but implicit SIMD execution
- Multiple instances running in parallel
  - Each instance working on a different data chunk
- On GPU
  - Divide work between lanes of SIMD units (fine division)
  - SIMD execution in lockstep
- On CPU
  - Divide work between cores (coarse division)
  - Sequential execution within a core (naive approach)

Single Program

- Kernel function
  - Entry point for the computation
- Executed once per work-item
  - As if there was a loop around it (but no dependency between iterations)
  - Access to the iteration counter using `get_global_id(0)`

```
kernel void add_uniform(int *dst,
                        int *src,
                        int alpha) {
  int tid = get_global_id(0);
  dst[tid] = src[tid] + (alpha - 1);
}
```

```
int *dst = ...;
int *src = ...;
int alpha = ...;
for (int tid = 0; tid < num_items; tid++) {
  dst[tid] = src[tid] + (alpha - 1);
}
```

Division of Work

- Work-item
  - Unit of work
  - One instance of a program
  - Executed in parallel by Execution Units (threads)
- Iteration space
  - 1D (array shape), used in this tutorial
  - 2D (grid shape)
  - ...

1D Iteration Space

Part 1: Background

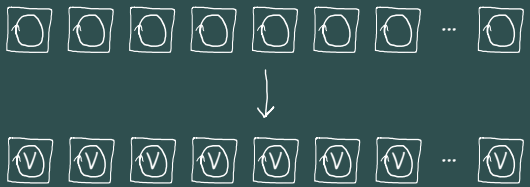
- SPMD Execution Model
- Vectorization

Why Vectorize?

- Many executions units each executing one instance of a single program
  - Works well on GPU (many hardware threads)
  - Not so much on CPU (very few cores)
  - CPU has to execute many work-items sequentially
- Speed up this sequential computation using SIMD units
  - Vertical Vectorization
  - Horizontal Vectorization

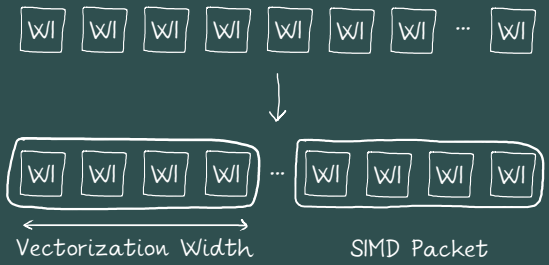
Vertical Vectorization

- Patterns within a single work-item
  - e.g. loops within a kernel
- Using the LLVM Loop Vectorizer or SLP Vectorizer
- However, not all kernels contain vectorizable patterns



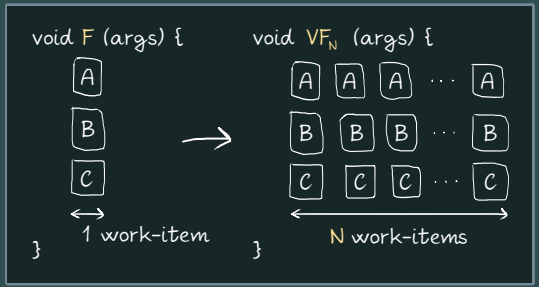
Horizontal Vectorization

- Across work-items
  - Compute multiple work-items at the same time
  - Take advantage of the execution model (single program, multiple data)
- Does not depend on special code patterns like loops
- SPMD Vectorizer



SPMD Vectorizer

- Vectorizes a SPMD program's entry point function
- Given a function F and vectorization factor N, produces a function VF_N
  - Calling VF_N is like calling F, but N times (on consecutive work-items)
  - F and VF_N have the same signature
- Vectorization may be allowed to fail
  - Work on cloned function
  - Use original function on failure



Implementation Level: IR or MI?

- IR
  - \+ Use-def graph and RAUW make for straightforward graph transformations
  - \+ Easy to target multiple platforms
  - **?** Generally higher-level (simpler implementation?)
  - – Platform-specific features more difficult to use
  - – SIMD predication only for a few operations (select, load/stores)
- MachineInstr (i.e. backend level)
  - \+ Easy to use platform-specific features (e.g. predication, mask registers)
  - **?** Generally lower-level (more powerful?)
  - – More platform-specific code
  - – Graph-based transformations not as straightforward
- Both?
  - – Transformations at IR level, generating CFG-specific metadata
  - – Use metadata in backend to do MI-level predication

Glossary

Work-item: Unit of computation to execute in parallel
Instance: State associated with one work-item
SIMD Lane: Execution of one instance of a vectorized kernel
SIMD Group: Contains all lanes that can be executed in parallel (at the same time)
SIMD Width: Number of parallel lanes, equal to vectorization factor (N)
Packet: Maps 1 value in the original function to N values, one per SIMD lane

Part 2

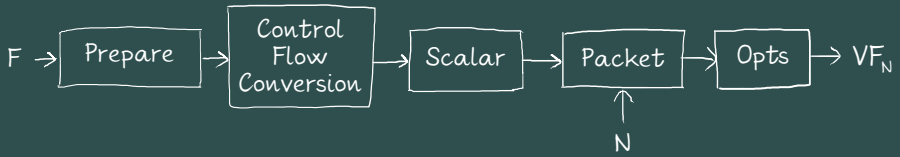
Implementing a SPMD Vectorizer

Part 2: Implementing a SPMD Vectorizer

- Overview
- Packetization Stage
- Scalarization Stage
- Control Flow Conversion Stage

Structure

- Pipeline design
  - F is repeatedly transformed by different stages
  - Stages are independent of each other
  - Each stage consists of one or more IR passes
  - Most stages require some analysis
- Analyses
  - Capture information about the IR to vectorize
  - May need updating after a stage (stale information)
  - May depend on other analyses



Analysis Examples

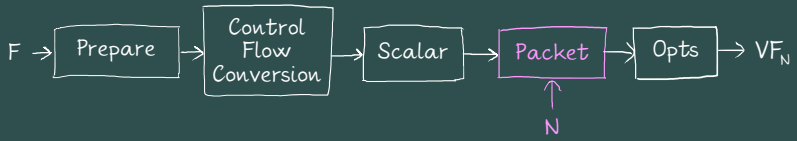
- Uniform Value Analysis (UVA)
  - Marks values as either uniform or varying
  - Uniform: Packet where values are identical for all lanes
  - Varying: Packet where values are not identical for all lanes
- Divergence Analysis
  - Determines which branches, which basic blocks are divergent
  - Divergent branch: Some lanes take one side, remaining lanes the other side
  - Divergent block: Not all lanes are active when executing the block
  - Depends on UVA
- SIMD Width Analysis
  - Chooses a 'good' width N based on register/instruction usage
  - Depends on UVA
- ...

Part 2: Implementing a SPMD Vectorizer

- Overview
- Packetization Stage
- Scalarization Stage
- Control Flow Conversion Stage

Packetization Overview

- Stage that does the actual vectorization: $F \xrightarrow{N} VF_N$
  - Calling VF_N is like calling F, but N times (N: SIMD width)
  - Straightforward thanks to preparation from previous stages
- This is done per-instruction, for the whole function
  - Instructions that define a value: define N values, one for each instance
  - Instructions with side effects: perform side effects for each instance
- Only varying instructions need packetization
  - Uniform instructions can remain scalar, executed once per work-group
  - Depends on UVA to know which instructions to vectorize



Uniform Value Analysis

Example that combines uniform and varying values:

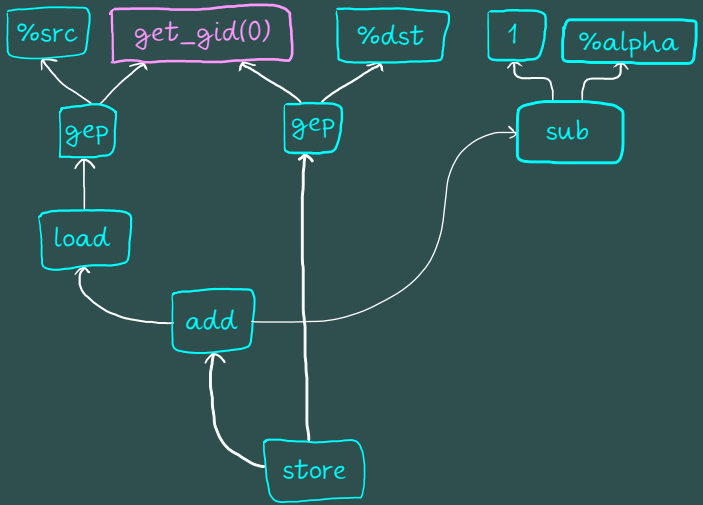
```
kernel void add_uniform(int *dst, int *src, int alpha) {
    int tid = get_global_id(0);
    dst[tid] = src[tid] + (alpha - 1);
}
```

```
define void @add_uniform(i32* %dst, i32* %src, i32 %alpha) {
entry:
  %tid = i32 @get_global_id(i32 0)
  %arrayidx = getelementptr i32* %src, i32 %tid
  %tmp = load i32* %arrayidx, align 4
  %sub = sub i32 %alpha, 1
  %add = add i32 %sub, %tmp
  %arrayidx2 = getelementptr i32* %dst, i32 %tid
  store i32 %add1, i32* %arrayidx2, align 4
  ret void
}
```

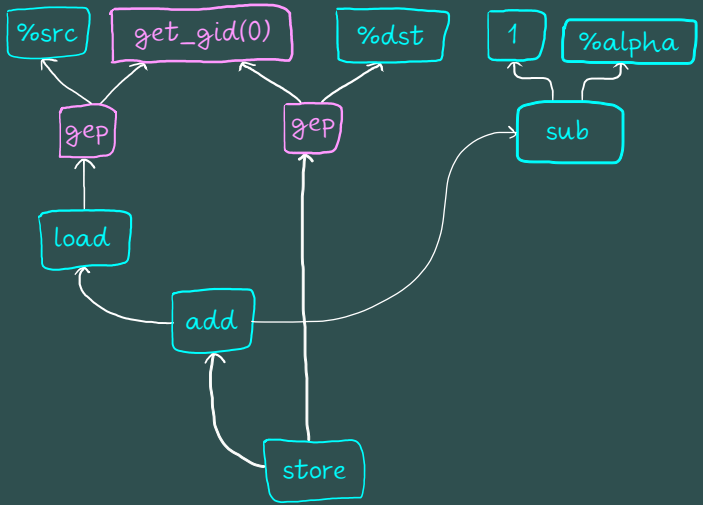
Uniform Value Analysis

- Finds 'root' values
  - Varying values with no varying operand
  - Example: get_global_id(0) has a different value for each instance
- Marks each IR value as uniform or varying
  - All values start as uniform
  - Marking a value as varying causes all users to also be marked varying
  - Marking is done recursively, starting with roots
  - Values are marked before their users, to avoid cycles (phi nodes)

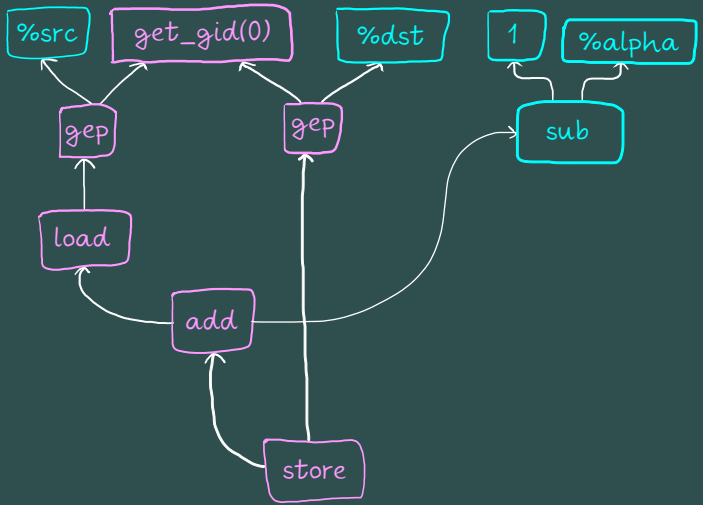
UVA Example: Start



UVA Example: Propagation

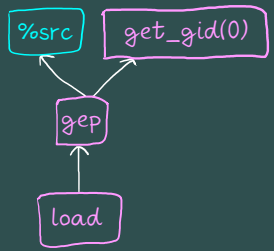


UVA Example: End



Memory Addressing

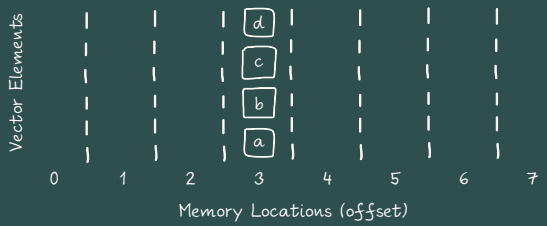
- Packetization depends on the addressing mode
  - Each memory operation can access N elements
  - Address usually has the form `base + offset'
  - Need to evaluate the offset for each of the N lanes
- How are these elements laid out in memory?
  - The layout affects how operations are packetized
  - Most layouts can be described with a single *stride*
- Stride is the distance between successive elements
  - Expressed in number of elements
  - One means elements are consecutive
  - Negative means memory offsets are decreasing



Uniform Memory Addressing

- Packetized offset is uniform (e.g. $< 3, 3, 3, 3 >$)
- Constant $Stride = 0$
- Transformed to a regular *scalar* load or store

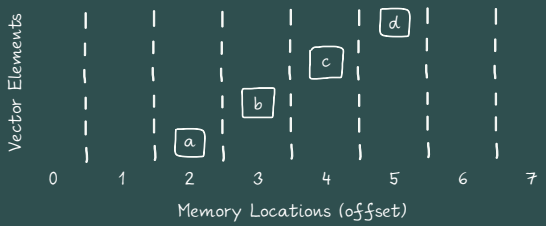
```
int *src;
int x = src[3];
```



Sequential Memory Addressing

- Packetized offset is a sequence like $< 2, 3, 4, 5 >$
- Constant $Stride = 1$
- Transformed to a regular *vector* load or store

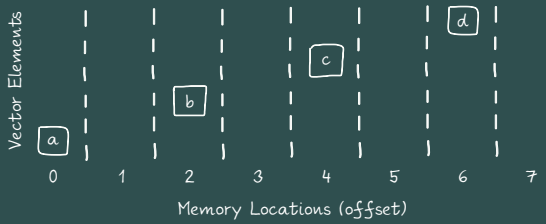
```
int *src;
int tid = get_global_id(0);
int x = src[tid + 2];
```



Interleaved Memory Addressing

- Packetized offset is a sequence like $< 0, 2, 4, 6 >$
- Constant $Stride > 1$
- Transformed to an *interleaved* load or store

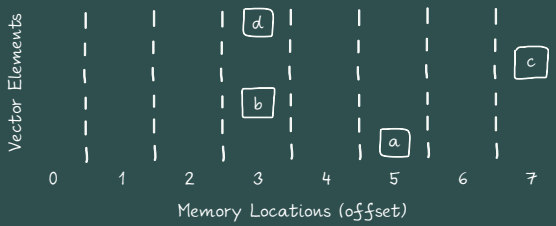
```
int *src;
int tid = get_global_id(0);
int even = src[tid * 2];
int odd = src[(tid * 2) + 1];
```



Arbitrary Memory Addressing

- Packetized offset can be any sequence (e.g. $< 5, 3, 7, 3 >$)
- Variable stride
- Transformed to a *gather* load or *scatter* store

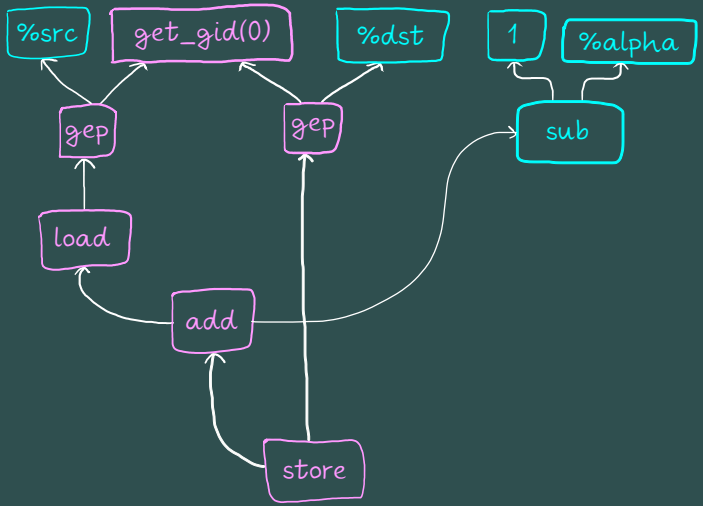
```
int *src;
int *map; // {5, 3, 7, 3};
int tid = get_global_id(0);
int x = src[map[tid]];
```



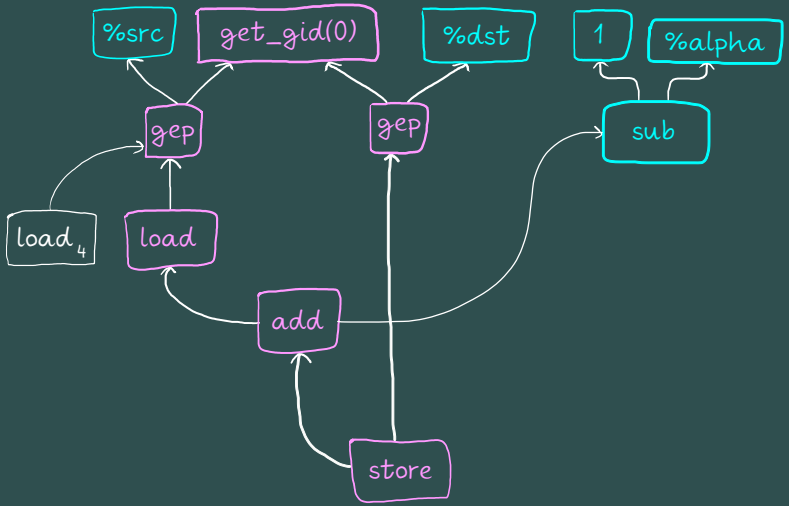
Packetization Process

- Find leaves
  - Leaves allow varying values to 'escape' from the function, they are:
  - Store instructions (varying operand)
  - Call instructions (varying operand, or call has no use)
  - Return instructions
- Recursively packetize leaves and their operands
  - Broadcast uniform values (e.g. argument, constants)
  - Replace get_global_id(0) with a vector of IDs
  - Packetize operands first, then instruction (top-down)
  - Cache packetized values to prevent duplication
- Delete original scalar instructions if dead

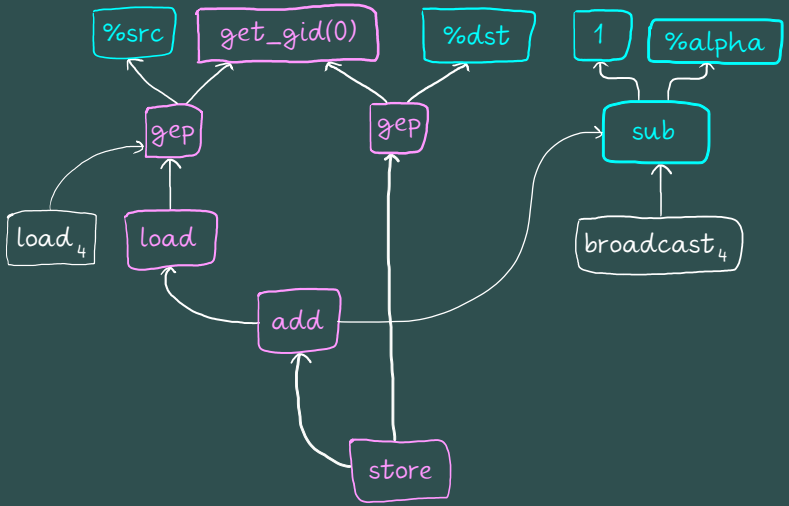
Packetization Example



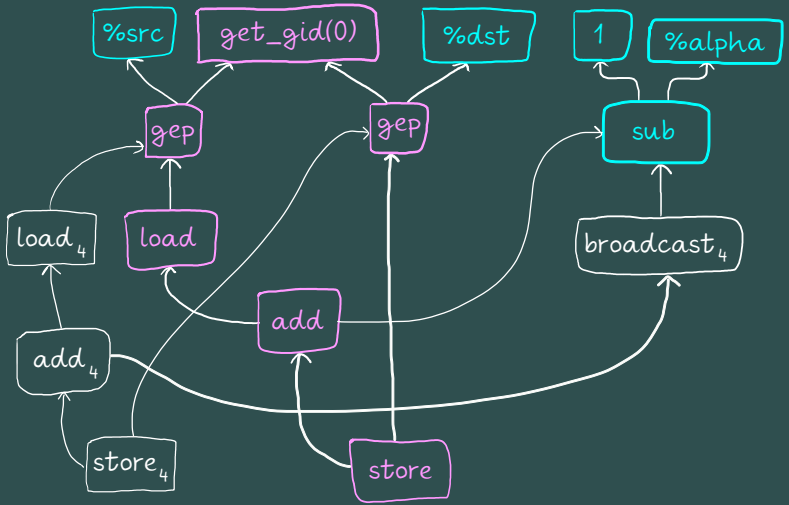
Packetization Example



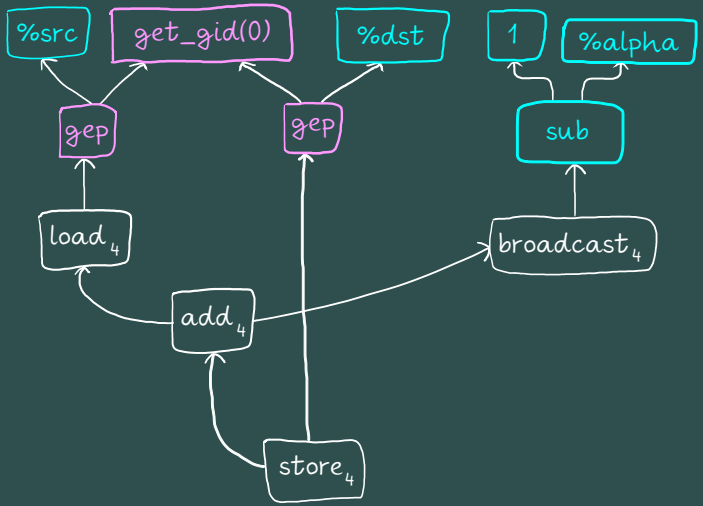
Packetization Example



Packetization Example



Packetization Example



Packetization Example

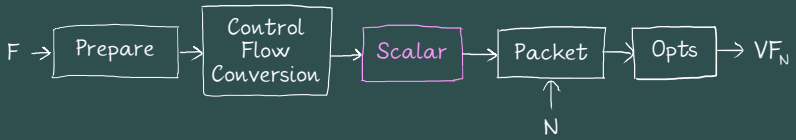
```
define void @__v4_add_uniform(i32* %dst, i32* %src, i32 %alpha) {
entry:
  %tid = call i32 @get_global_id(i32 0)
  %arrayidx = getelementptr i32* %src, i32 %tid
  %0 = bitcast i32* %arrayidx to <4 x i32>*
  %1 = load <4 x i32>* %0, align 4

  ; Broadcast (alpha - 1) to a vector
  %sub = sub i32 %alpha, 1
  %insert = insertelement <4 x i32> undef, i32 %sub, i32 0
  %broadcast_sub = shufflevector <4 x i32> %insert, ...

  %add = add nsw <4 x i32> %broadcast_sub, %1

  %arrayidx2 = getelementptr i32* %dst, i32 %tid
  %2 = bitcast i32* %arrayidx2 to <4 x i32>*
  store <4 x i32> %add, <4 x i32>* %2, align 4
  ret void
}
```

Part 2: Implementing a SPMD Vectorizer

- Overview
- Packetization Stage
- Scalarization Stage
- Control Flow Conversion Stage

Scalarization Overview

- Eliminates vector operations from the source function
- Vector types used likely to be narrower than the native SIMD width
- To be combined with packetization
  - Generate vector instructions with the native SIMD width
- On its own, does not change the the behaviour of the code



Scalarization Example

- Example: Extract audio samples from left and right channels, scale by 2
- Scalarizing n-element loads and stores introduces a stride of n
  - Results in interleaved loads and stores after packetization

Before Scalarization (fragment):

```
int2 *src, int *left, int *right;
int tid = get_global_id(0);


int2 sample = src[tid];


left[tid] = (sample.x << 1);
right[tid] = (sample.y << 1);
```

After Scalarization (reconstructed):

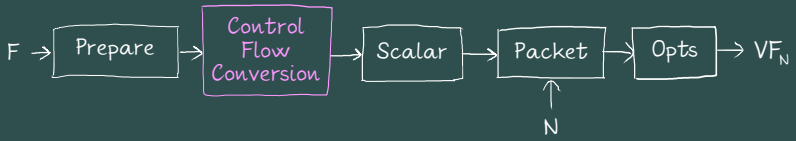
```
int2 *src, int *left, int *right;
int tid = get_global_id(0);

int *srcScalar = ((int *)src);
int sampleLeft = srcScalar[(tid * 2) + 0];
int sampleRight = srcScalar[(tid * 2) + 1];

left[tid] = (sampleLeft << 1);
right[tid] = (sampleRight << 1);
```

Part 2: Implementing a SPMD Vectorizer

- Overview
- Packetization Stage
- Scalarization Stage
- Control Flow Conversion Stage

Control Flow Conversion: Overview

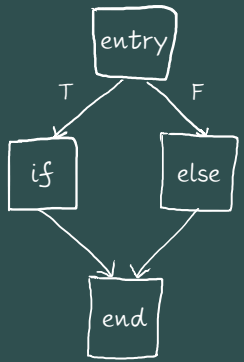
- Linearizes functions that have divergent control flow
  - Conversion from control flow to data flow
  - All basic blocks are executed
  - Program semantics are preserved using predication (masking)
- Why is it needed?
  - SIMD unit does not support 'vector' (divergent) branches
  - Single program counter per SIMD group
- Requires some passes to be run in the 'Prepare' stage
  - Functions should have a single return block
  - Loops should be in 'simple form'



Control Flow Conversion: if

- Divergent branch condition: cond
- Instruction with side-effects: load

```
kernel void copy_if_even(int *src, int *dst) {
  int tid = get_global_id(0);
  int cond = (tid & 1) == 0;
  int result;
  if (cond) {
    result = src[tid];
  } else {
    result = -1;
  }
  dst[tid] = result;
}
```

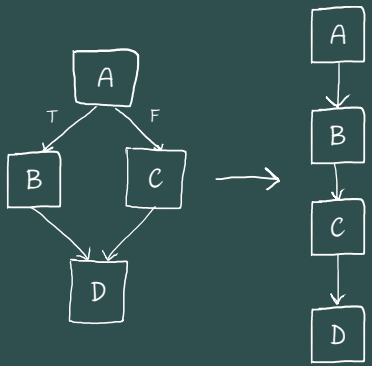


Control Flow Conversion: Main Steps

- Divergence Analysis
- Generate Masks
- Freeze Loop Live Variables
- Apply Masks
  - Instructions with side-effects
- Convert Phi Nodes
  - Preserves data flow
- CFG Linearization

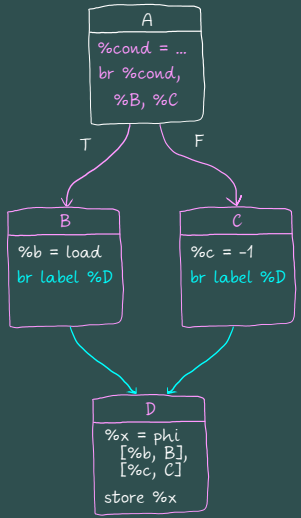
CFG Linearization Overview

- Flattens the CFG
  - All blocks are executed
  - Regardless of branch conditions
- Steps
  - Order blocks
  - Rewrite branches
- More on this later



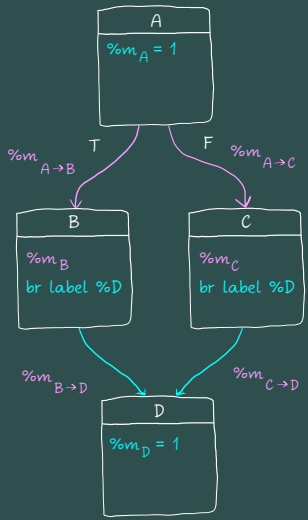
Basic Divergence Analysis

- Determines which basic blocks need predication (i.e. are divergent)
- BB is divergent if:
  - Any predecessor has a branch with a varying condition
  - Any predecessor is divergent (naive)
- Process:
  - Start with the entry BB
  - Mark successors divergent or not
  - Visit all successors recursively
  - Visit each BB only once (cycles)



Mask Generation

- Mask: N-bit field (1-bit pre-packetization)
  - Per-instance, 'active' bit for predication
- Each edge $A \rightarrow B$ has a mask: $m_{A \rightarrow B}$
  - Which lanes take the branch to B?
  - $m_{A \rightarrow B} = m_A \cap bcond_{A \rightarrow B}$
  - Given branch condition $bcond_{A \rightarrow B}$
- Each block B has an entry mask: m_B
  - Which lanes execute B?
  - $m_B = \bigcup_{i=0}^{n} m_{P_i \rightarrow B}$
  - Given predecessors $P_0 \cdots P_n$
- Start by generating return mask m_D
  - Depends on all other masks



Applying Masks

- Each block B is executed regardless of whether F executes it or not
- Each instruction I that has side-effects need 'guarding'
- Such instructions are predicated using m_B
  - I has side-effects for lane L if $m_B[L]$ is true
  - Loads and stores are turned into masked loads and stores
  - Calls to functions with side-effects: m_B is passed as an argument
  - Floating-point instructions that raise exceptions (e.g. DIV0)
  - ...
- Unsupported masked operations can be expanded
  - For each lane L, generate: `if` $(m_B[L])$ { $V_L = I_L(\ldots)$; }
  - Need to create many basic blocks

Phi Conversion

- A phi node:
  - Takes incoming blocks $P_0 \cdots P_n$, values $V_0 \cdots V_n$
  - Evaluates to V_i if the incoming block was P_i
- Does not work after linearization
  - Each block B has a single predecessor after linearization
  - Actual incoming block: find P_i, so that $m_{P_i \to B}$ is true
- Need to convert phi nodes into n select instructions
  - Using $m_{P_0 \to B} \cdots m_{P_n \to B}$ to select V from $V_0 \cdots V_n$

```
A:
  %cmp = ...
  br i1 %cmp, label %B, label %C
B:
  %x = load i32* %idx
  br label %C
C:
  %v = phi i32 [%x, %B], [-1, %A]
```

```
A:
  %cmp = ... // = mAB = mB = mBC
  br label %B
B:
  %x = masked_load i32* %idx, i1 %cmp
  br label %C
C:
  %v = select i1 %cmp, i32 %x, i32 -1
```

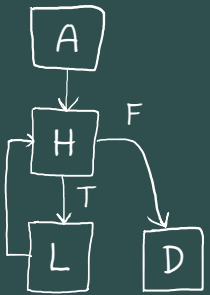
CFG Conversion Result: if

```
define void @__v4_copy_if_even(i32* %in, i32* %out) {
entry:
  %call = call spir_func i64 @get_global_id(i32 0)
  %.splatinsert = insertelement <4 x i64> undef, i64 %call, i32 0
  %.splat = shufflevector <4 x i64> %.splatinsert, <4 x i64> undef, <4 x i32> zeroinit
  %0 = add <4 x i64> %.splat, <i64 0, i64 1, i64 2, i64 3>
  %and1 = and <4 x i64> %0, <i64 1, i64 1, i64 1, i64 1>
  %cmp2 = icmp eq <4 x i64> %and1, zeroinitializer

; if.then:
  %arrayidx = getelementptr inbounds i32* %in, i64 %call
  %1 = call <4 x i32> @masked_load4(i32* %arrayidx, <4 x i1> %cmp2)

; if.end:
  %2 = select <4 x i1> %cmp2, <4 x i32> %1, <4 x i32> <i32 -1, i32 -1, i32 -1, i32 -1>
  %arrayidx1 = getelementptr inbounds i32* %out, i64 %call
  %3 = bitcast i32* %arrayidx1 to <4 x i32>*
  store <4 x i32> %2, <4 x i32>* %3, align 4
  ret void
}
```

Control Flow Conversion: Loops

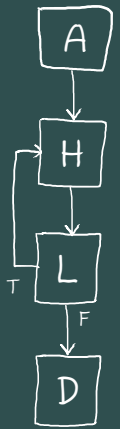
- More difficult to convert
- More masks to compute
- Loop condition may be varying

```
kernel void while_loop(int *src,
                       int *dst,
                       int step) {
  int tid = get_global_id(0);
  int x = src[tid];
  while (x < 0) {
    x += step;
  }
  dst[tid] = x;
}
```



Loop Exit and Active Masks

- Different instances may iterate a different number of times
  - Because the loop condition is varying
  - Keep iterating as long as any instance is inside the loop
- Loop Exit mask m_{exit}
  - Keeps track of which instances exited the loop
  - Used as entry mask for loop exits (m_D)
  - Needs a phi node since this changes over iterations
- Loop Active mask m_{active}
  - $m_{active} = m_{header} \cap \neg m_{exit}$
  - True: Branch from loop latch L back to loop header H
  - False: Exit the loop from loop latch L

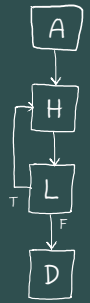


Freezing Loop Live Variables

- Variables that are either:
  - Used in a subsequent loop iteration (through a phi node)
  - Used outside of the loop
  - In the example: x
- Once an instance exits, need to freeze live variables
  - Otherwise it will have the wrong value after the loop
- Create a select instruction x_{frozen} that returns either:
  - The new value from this iteration x_{new} (instance is active)
  - The value from the previous iteration x_{prev} (instance exited)
  - m_B selects the right value, where B contains x
- Replace all uses of x_{new} with x_{frozen}
- Replace outside-loop uses of x_{prev} with x_{frozen}
  - This happens in 'while' and 'for' loops

Loop Execution Example

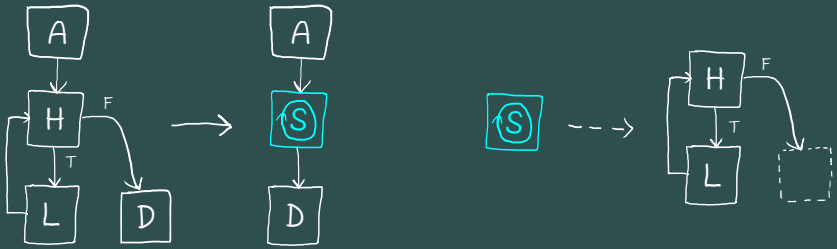
- Execute the loop header H
  - $m_{exit} \leftarrow m_{exit} \cup (x_{prev} \geq 0)$
  - $m_{active} \leftarrow m_{entry} \cap \neg m_{exit}$
- Execute the loop body L
  - $x_{new} \leftarrow x_{prev} + 2$
- Freeze live value x
  - $x_{frozen} \leftarrow select(m_{active}, x_{new}, x_{prev})$
- Branch to H if $any(m_{active})$, or exit loop to D



Iteration	x_{prev}				m_{entry}	m_{exit}	m_{active}	x_{new}				x_{frozen}			
0	7	-2	-3	-5	●●○●	●○○○	○●○●	9	0	-1	-3	7	0	-3	-3
1	7	0	-3	-3	○●○●	●●○○	○○○●	9	2	-1	-1	7	0	-3	-1
2	7	0	-3	-1	○○○●	●●○○	○○○●	9	2	-1	1	7	0	-3	1
3	7	0	-3	1	○○○●	●●○●	○○○○	9	2	-1	3	7	0	-3	1

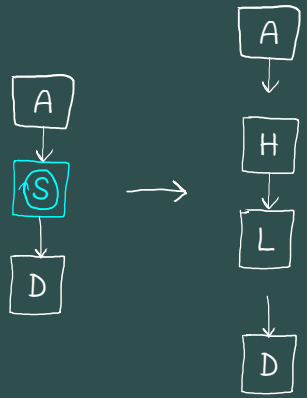
Basic Linearization: Graph Creation

- Create a CFG-like graph
  - Where a loop's blocks are replaced by a single loop node
  - The loop node (S) contains a sub-graph
- Sub-graphs can contain blocks and loops
  - Allows recursive processing of loop nests



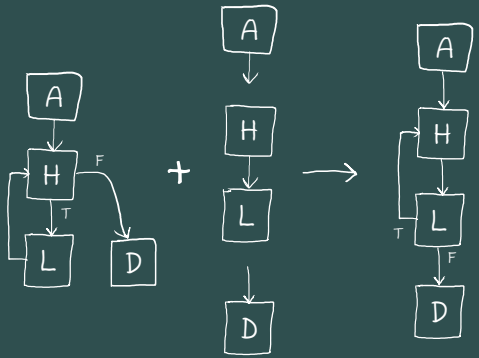
Basic Linearization: Block Ordering

- Bottom-up topological sort of the graph
  - Result is an ordered list of graph nodes
  - Loop nodes expand to sub-lists
  - Topological sort of loop nodes starts at the loop latch
- Naive approach that linearizes everything
  - Increases register pressure
- Only works for reducible control flow



Basic Linearization: Branch Rewriting

- Visit each block in the ordered list
- Rewrite their branch target
- For most blocks:
  - Always branch to the next block in the list
- For the loop latch L:
  - How many active instances in the loop?
  - ≥ 1: Branch to the loop header H
  - 0: Branch to next block
  - Uses $any(m_{active})$
- CFG conversion is done



Conclusion

- Explained basic concepts
  - Data-parallel execution model
  - Whole-function vectorization
  - N instances of every instruction
  - Uniform vs varying values
  - Divergent control flow, masking
- Many things were not covered in this talk
  - 2D, 3D iteration spaces
  - Loops with multiple exits
  - More advanced analyses
  - Optimizations
  - ...
- Should be enough to create a functional vectorizer

References and Resources

1. Automatic Packetization [Ralf Karrenberg, Saarland University '09]
2. Whole-Function Vectorization (Ralf Karrenberg, Sebastian Hack, CGO '11)
3. Intel® OpenCL™ Implicit Vectorization Module [Nadav Rotem, LLVM '11]
4. Branching in Data-Parallel Languages using Predication with LLVM [Marcello Maggioni, EuroLLVM '14]
5. Exploring the Design Space of SPMD Divergence Management on Data-Parallel Architectures [Yunsup Lee et al., MICRO '14]

Thank you!

- Q&A
  - Happy to answer questions by email too: pierre-andre@codeplay.com
- ...
- Happy vectorizing!

Part 3

Going Further

SIMD Width Detection

- Basic process
  - Visit varying nodes
  - Record width W of widest type used
  - Given vector register width V, $N = \frac{V}{W}$
- Improve analysis using register pressure information
  - Max register usage $< \frac{1}{p}$? Multiply N by p
  - Result in p times the number of native vector operations
- Use a cost model
  - Your target may only support some vector operations on specific widths

Instantiation

- Not all instructions can be packetized
  - External function with side-effcts (e.g. `printf`)
  - Atomic builtins
- Solution: instantiate the instruction for all lanes ($i \in [0; N)$)
  - Duplicate scalar instructions N times
  - Replace tid with $tid + i$ (e.g. calls to `get_global_id`)
  - Need to extract packetized operands N times
- Happens during the packetization stage
- Can be an alternative to scalarization
  - Instantiation then becomes a stage of its own
  - Analysis determines when to instatiate, when to scalarize

Packetizing Builtin Function Calls

- Requires a 'builtin function database'
  - Which functions are builtins
  - Argument types, unless encoded in the mangled function name (e.g. OpenCL)
  - Properties (e.g. returns an item ID, has side-effects, pointer return, etc)
- Map scalar builtin to vector equivalent (e.g. `tan(float)` to `tan(float4)`)
  - And the other way around, for scalarization
  - Assumes the equivalent builtin is already packetized
- Simply create call to vector equivalent with packetized operands
- However not all OpenCL builtins have vector equivalents (e.g. `float dot(float4)`)
  - Can inline these builtins before scalarization

Packetizing Builtin Functions

- Clone builtin function, updating signature (vector return value, arguments)
- Generate argument placeholders (see example below), replacing all uses
- UVA treats arguments as roots
- `ret` instructions are leaves
- Packetizing a placeholder replaces `extractelement` by the argument

IR after cloning, before packetization:

```
define <4 x i32> @__v4__Z7isequalff(<4 x float> %x, <4 x float> %y) {
entry:
  %placeholder_x = extractelement <4 x float> %x, i32 0
  %placeholder_y = extractelement <4 x float> %y, i32 0
  %cmp.i = fcmp oeq float %placeholder_x, %placeholder_y
  %conv.i = zext i1 %cmp.i to i32
  ret i32 %conv.i
}
```

Packetizing User Functions (No Side-Effects)

- Vectorizer has no intrinsic knowledge of which arguments are varying
  - Need to analyze this for each call site
  - Generate a packetized function for each combination
- Arguments may also be uniform (e.g. arrays)
- Otherwise similar to packetizing builtins

Packetizing User Functions (Side-Effects)

- The vectorized function needs to take an extra mask argument, m_{entry}
  - Determines which lanes are enabled when entering the function
- When applying masks, pass m_B to function calls
  - Where B is the block where the call instruction is
- Might be simpler to just inline such calls

CFG Specialization

- Duplicate part of the CFG
  - With the assumption that all lanes are enabled
  - Avoids CFG conversion and predication for the specialized part
  - Increases code size
- Need to generate an extra branch (guard) to specialized code
  - e.g. 'b i1 $all(m_{A->B})$, label $\%B_{spec}$, label $\%B$'

```
kernel void convolution(float *src, float *dst) {
  int x = get_global_id(0);
  int width = get_global_size(0);
  float sum = 0.0f;
  if ((x >= FILTER_SIZE) && (x < (width - FILTER_SIZE))) {
    /* Loop that computes sum, using an uniform condition */
  }
  dst[x] = sum;
}
```

CFG Conversion: Single Lane

- Branch always taken by a single lane
  - e.g. `'if (tid == 0) { /* write back result */ }'`
  - Often used with reductions
- No need for CFG conversion
  - Keep the conditional branch
- No need for packetization

Interleaved Memory Optimizations

- Scalarizing vector loads and stores results in many interleaved loads and stores
- Most targets do not support this efficiently
  - Resulting in even more scalar loads, stores, vector extractions and insertions
- Grouping these instructions often helps
  - ARM supports `vld.[2-4]` and `vst.[2-4]` for some vector types
  - Can replace n-group with n memory operations and $n \times n$-transposition
- To find groups, look for a common base pointer and increasing offsets

AoS to SoA Conversion

- Scalarizing then packetizing vector loads and stores implicitely performs Array-of-Structures to Structure-of-Arrays conversion
- Common 'load(s)-compute-store(s)' pattern inside kernels
  - Computation is done per-element, without shuffling elements
  - Interleaved loads and stores generated due to scalarization
  - Can replace with regular vector loads and stores to avoid the conversion
- Analysis needed to show that no shuffling or single-lane accesses occur
- Resulting code likely to be much more friendly to most targets

Implementation Strategy

- Create test kernels
  - Start with very simple kernels (e.g. copy buffer, add two buffers)
  - Gradually add more features (e.g. non-sequential memory accesses, vector instructions, etc)
- Suggested implementation order
  - Preparation and packetization first (required for simplest kernels)
  - Then easier features: builtins, memory addressing, scalarization, instantiation
  - More complex features last: control flow, optimizations

Scalarization Process

- Look for vector varying instructions such as:
  - Leaves that define vector values, vector stores
  - Vector extractions
  - Vector -> scalar bitcasts
- Recursively scalarize until we reach a scalar value
  - Operands before instructions
  - Re-create instructions for each vector element
  - Vector lane ≠ SIMD instance!

Scalarization Example

After Scalarization:

```
kernel void extract_lr(int2 *src, int *left, int *right) {
    int tid = get_global_id(0);
    int sampleLeft = *((int *)&src[tid] + 0);
    int sampleRight = *((int *)&src[tid] + 1);
    left[tid] = (sampleLeft >> 1);
    right[tid] = (sampleRight >> 1);
}
```

After Packetization:

```
kernel void extract_lr(int2 *src, int *left, int *right) {
    int tid = get_global_id(0);
    int4 samplesLeft  = interleaved_load_int4((int *)&src[tid] + 0, 2);
    int4 samplesRight = interleaved_load_int4((int *)&src[tid] + 1, 2);
    vstore4(samplesLeft >> 1, tid, (int *)left);
    vstore4(samplesRight >> 1, tid, (int *)right);
}
```