“Fallback” of load/store into gather/scatter in LLVM-IR

Euro LLVM 2023 Developers’ Meeting

Quick Talk by Omer Aviram

Agenda

- Motivation
- “Fallback” utility overview
- Usage example
- Cost model
- Performance and robustness conclusions

Motivation – Overcoming memory accessing limitations

Architectures employing **hardware-controlled loops with zero-overhead,** supporting both memory patterns:

- load/store units - with pre-configured strides based on compiler analysis - controlling loop execution.
- scatter/gather units for indirect accesses calculated in runtime.

Motivation – Overcoming memory accessing limitations

Architectures employing **hardware-controlled loops with zero-overhead,** supporting both memory patterns:

- load/store units - with pre-configured strides based on compiler analysis - controlling loop execution.
- scatter/gather units for indirect accesses calculated in runtime.

Hardware resources

- Finite number of load/store units

Indirect index accesses

- `arr[idx_arr[i]]` – Unable to pre-configure memory access stride.

“Fallback” utility overview

LLVM-IR utility designed to convert ("fallback") memory accesses to sequential data (such as vectorized load/store) into indirect accesses (scatter/gather)

“Fallback” utility overview

LLVM-IR utility designed to convert ("fallback") memory accesses to sequential data (such as vectorized load/store) into indirect accesses (scatter/gather)

Semantics reminder (LLVM LangRef):

- load instruction recieves **a single pointer** (to scalar/vector) from memory

```
declare <8 x float>  @llvm.masked.load.v8f32(ptr <ptr>, i32 <alignment>, <8 x i1> <mask>,
 <8 x float> <passthru>)
```

- gather instruction receives **a vector of pointers** to arbitrary memory locations and gathers them into a vector.

```
declare <8 x float> @llvm.masked.gather.v8f32.v8(<8 x ptr> <ptrs>, i32 <alignment>, <8 x
i1> <mask>, <8 x float> <passthru>)
```

Explaining the transformation (1)

```
void foo(int4 *base, int *idx_arr, int4 *out) {
  for(int x = 0; x < WIDTH; x++)
    out[x] = base[idx_arr[x]];
}
```

`base[idx_arr[x]]` is an indirect memory access – compiler is **unable to pre-configure its stride** into the load/store unit .

Instead - "fallback" it into an indirect masked gather instruction.

Explaining the transformation (2) – LLVM IR

```
%orig_gep = getelementptr inbounds <4 x i32>, ptr %base, i32 %loaded.idx
%loadVec4 = load <4 x i32>, ptr %orig_gep, align 8
```

Fallback involves manipulating the GEP -

from **a pointer to vector** of sequential data

into a **vector of pointers** to consecutive elements

Explaining the transformation (2) – LLVM IR

```
%orig_gep = getelementptr inbounds <4 x i32>, ptr %base, i32 %loaded.idx
%loadVec4 = load <4 x i32>, ptr %orig_gep, align 8
```

Fallback involves manipulating the GEP -

from **a pointer to vector** of sequential data

into a **vector of pointers** to consecutive elements

- Transforming `base[idx_arr[x]]`

```
%mul.by.vf = mul i32 %loaded.idx, 4
%splatinsert = insertelement <4 x i32> poison, i32 %mul.by.vf, i32 0
%splat.mul.idx.vf = shufflevector <4 x i32> %.splatinsert, <4 x i32>
poison, <4 x i32> zeroinitializer
%vec.idx = add <4 x i32> %splat.mul.idx.vf, <i32 0, i32 1, i32 2, i32 3>
%gather_gep = getelementptr <4 x i32>, ptr %base, i32 0, <4 x i32> %vec.idx
%fallback.gather = call <4 x i32> @llvm.masked.gather.v4i32.v4((<4 x ptr>))
%gather_gep, i32 8, <4 x i1> %true_mask, <4 x i32> %passthrough)
```

Some fallback transformations are less trivial

Pointer arithmetic resulting in a “chain” of GEPs -

```
;; Original load using a "chain of GEPs" instructions:
%ptr = getelementptr inbounds i8, ptr %base, i32 %indices1
%ptr2 = getelementptr inbounds i8, ptr %ptr, i32 %indices2
%ptr3 = getelementptr inbounds i8, ptr %ptr2, i32 %indices3
load i8, ptr addrspace(3) % ptr3, align 1

;; Converted gather with a single GEP instruction:
%add.indices = add i32 %mul_indices1_by_vf, %mul_indices2_by_vf
%add.indices2 = add i32 %indices3, %add.indices
%vecrozied_index = insertelement <1 x i32> poison, i32
%add.indices2, i32 0
%folded_gep = getelementptr inbounds i8, ptr %base, <1 x i32>
%vecrozied_index
call <1 x i8> @llvm.masked.gather.v1i8.v1(<1 x ptr> %folded_gep,
i32 1, <1 x i1> <i1 true>, <1 x i8>
```

```
base = base_ptr + i * 8 + j * 4;
char res = base[x+y*YStride];
```

OpenCL

Type reinterpretation -

```
;; Load from a different type returned by GEP.
%ptr = getelementptr inbounds <4 x i8>, ptr %in1, i32 %indices
%0 = load <2 x i16>, ptr %ptr, align 4

;; Converted to a gather followed by bitcast.
%vec.idx = add <4 x i32> %mul.idx.by.vf, <i32 0, i32 1, i32 2, i32 3>
%gep = getelementptr inbounds <4 x i8>, ptr %in1, i32 0, <4 x i32>
%vec.idx
%fallback.gather = call <4 x i8> @llvm.masked.gather.v4i8.v4(<4 x ptr
addrspace(3)> %gep, i32 4, <4 x i1> <i1 true, i1 true, i1 true, i1
true>, <4 x i8> poison)
%fix.dt = bitcast <4 x i8> %fallback.gather to <2 x i16>
```

```
for(int x = 0; x < height; x++)
    out[x] = as_short2(in1[in2[x]]);
}
```

OpenCL

Utilizing hardware resources

- Given a VLIW architecture with limited hardware resources:
    - 1 load unit
    - 1 store unit
    - 1 scatter/gather unit

```
for(int x = 0; x < height; x++) {
   out[x] = base1[x] + base2[2 * x + 7];
}
```

Utilizing hardware resources

- Given a VLIW architecture with limited hardware resources:
  - 1 load unit
  - 1 store unit
  - 1 scatter/gather unit

```
for(int x = 0; x < height; x++) {
   out[x] = base1[x] + base2[2 * x + 7];
}
```

Converting one of the load instructions (`base1[x]` or `base2[2 * x + 7]`) into a gather will better utilize hardware resources.

But which load is best to "fallback"?

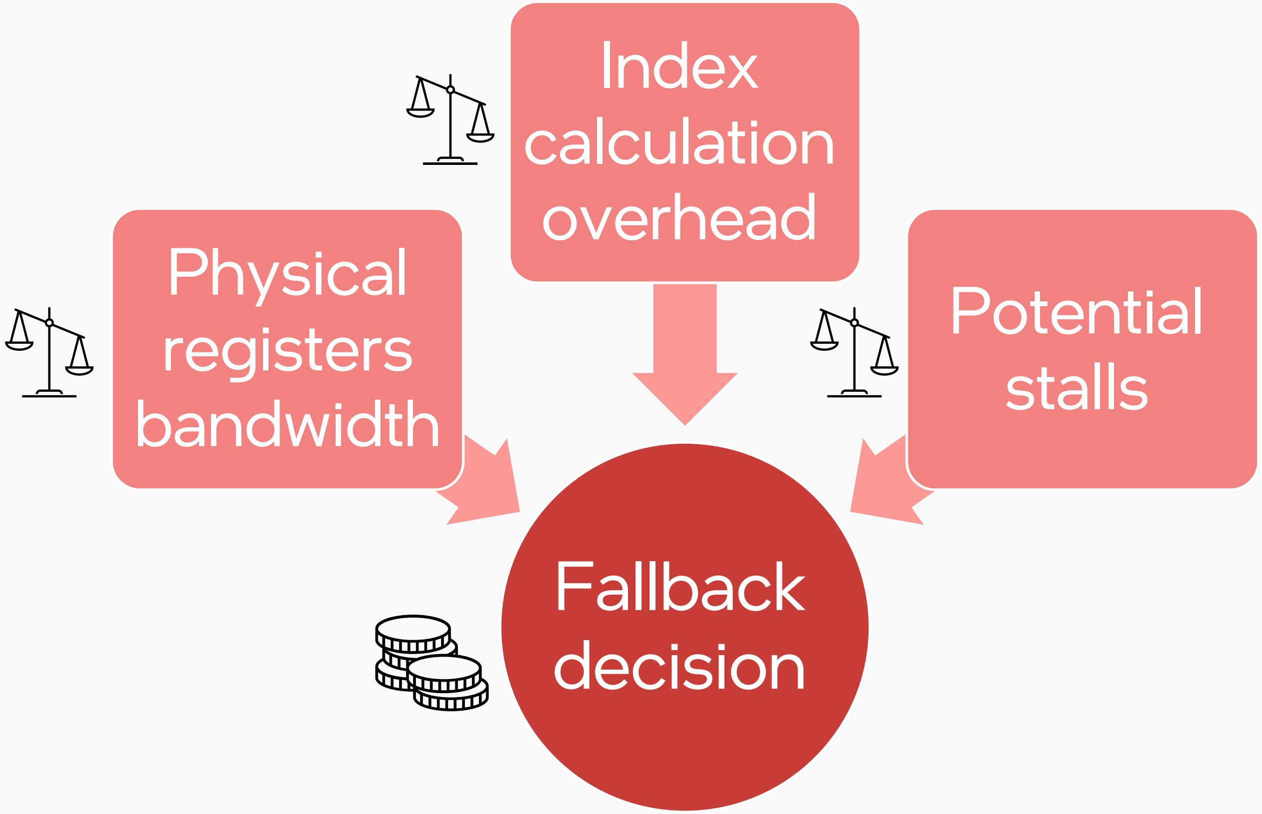
Target supported cost model

Targets may implement **architecture-based cost model**, to decide which memory access to “fallback” in order to maximize performance.

```
for(int x = 0; x < height; x++) {
   out[x] = base1[x] + base2[2 * x + 7];
}
```

Target supported cost model

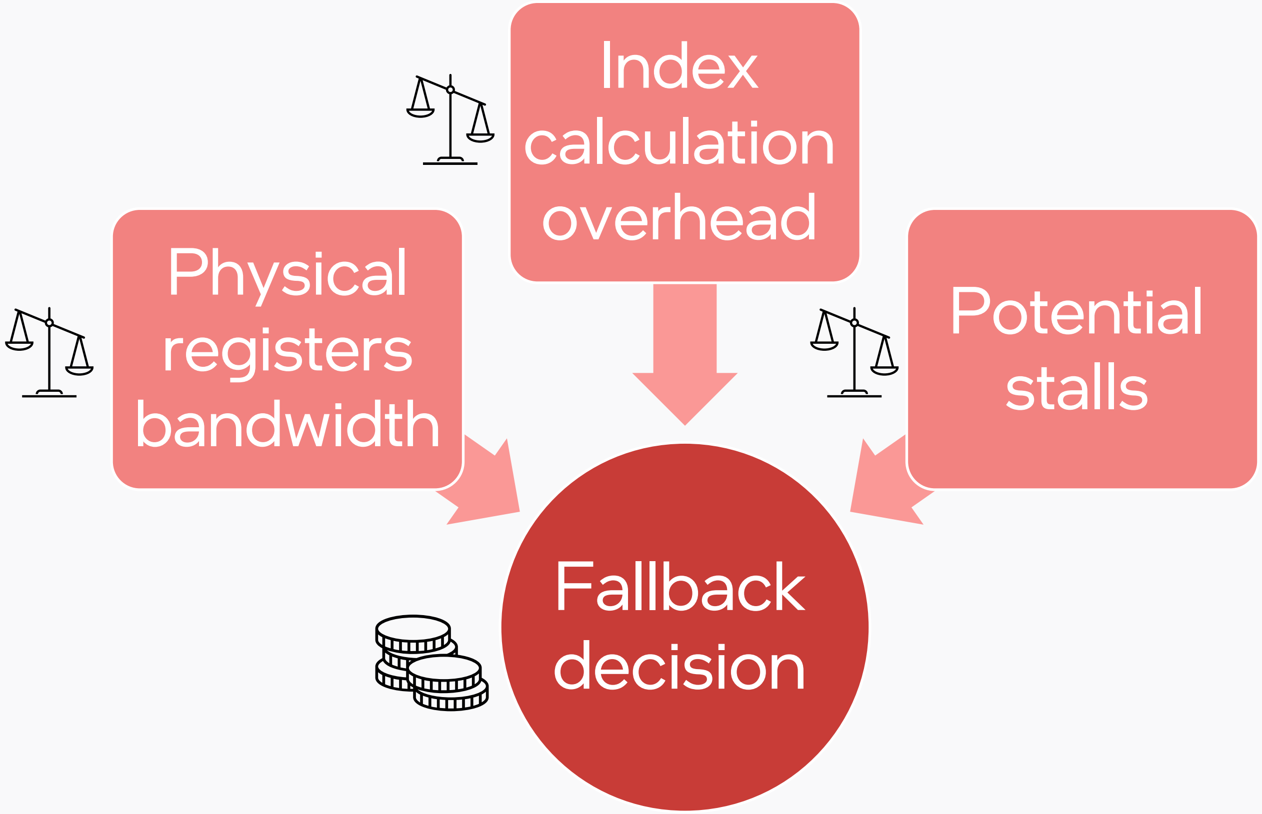
Targets may implement **architecture-based cost model**, to decide which memory access to “fallback” in order to maximize performance.




```
for(int x = 0; x < height; x++) {
    out[x] = base1[x] + base2[2 * x + 7];
}
```

Target supported cost model

Targets may implement **architecture-based cost model**, to decide which memory access to “fallback” in order to maximize performance.




```
for(int x = 0; x < height; x++) {
   out[x] = base1[x] + base2[2 * x + 7];
}
```

base1[x] requires less runtime index calculation than base2[2 * x + 7] -> better fallback performance

Performance/Robustness conclusions

- Compiler robustness – overcome hardware limitations
  - ~5% more tests compiled for target successfully.
- Non-optimized naïve code has a better chance to compile successfully – better user experience for compiler customers.
- Performance may improve thanks to balancing unit pressure between load/store compared to gather/scatter.

Thank you!