Apple LLVM GPU Compiler: Embedded Dragons

Charu Chandrasekaran, Apple
Marcello Maggioni, Apple
Agenda

• How Apple uses LLVM to build a GPU Compiler
• Factors that affect GPU performance
• The Apple GPU compiler
  • Pipeline passes
  • Challenges
How Apple uses LLVM

• Live on Trunk and merge continuously
• Benefit from latest improvements on trunk
• Identify any regressions immediately and report back
• Minimize changes to open source llvm code
• Reuse as much as possible
Continuous Integration

Year 1 production compiler

LLVM Trunk

GPU Compiler
Continuous Integration

Year 1 production compiler
Year 2 production compiler
Year 3 production compiler

LLVM Trunk
GPU Compiler
Testing

- Regression testing involves:
  - register count
  - instruction count
  - FileCheck: correctness
  - compile time
  - compiler size
  - runtime performance
The GPU SW Stack

iOS / watchOS / tvOS Process

Metal Framework, GPU Driver

App

Shader

XPC Service

Metal-FE

XPC Service

Backend

User

Interacts

Result

Result

.metadata

.exec

IR

.obj
About GPUs
GPU are massively parallel vector processors

Threads are grouped together and execute in lockstep (they share the same PC)
The parallelism is implicit, a single thread looks like normal CPU code

```c
float kernel(float a, float b) {
    float c = a + b;
    return c;
}
```
The parallelism is implicit, a single thread looks like normal CPU code

```c
float8 kernel(float8 a, float8 b) {
    float8 c = add_v8(a, b);
    return c;
}
```
About GPUs: Latency hiding

Multiple groups of threads are resident on the GPU at the same time for latency hiding.

```c
float kernel(struct In_PS ) {
    float4 color = texture_fetch();
    float4 c = In_PS.a * In_PS.b;
    ...
    float4 d = c + color;
    ...
}
```
The GPU picks up work from the various different groups of threads to hide the latency from the other groups

```c
float kernel(struct In_PS ) {
    float4 color = texture_fetch();
    float4 c = In_PS.a * In_PS.b;
    ...
    float4 d = c + color;
    ...
}
```
About GPUs: Latency hiding

The GPU picks up work from the various different groups of threads to hide the latency from the other groups.

```c
float kernel(struct In_PS ) {
    float4 color = texture_fetch();
    float4 c = In_PS.a * In_PS.b;
    ... 
    float4 d = c + color;
    ... 
}
```
The GPU picks up work from the various different groups of threads to hide the latency from the other groups.

```c
float kernel(struct In_PS ) {
    float4 color = texture_fetch();
    float4 c = In_PS.a * In_PS.b;
    ...
    float4 d = c + color;
    ...
}
```
About GPUs: Latency hiding

The GPU picks up work from the various different groups of threads to hide the latency from the other groups.

```c
float kernel(struct In_PS ) {
    float4 color = texture_fetch();
    float4 c = In_PS.a * In_PS.b;
    ...
    float4 d = c + color;
    ...
}
```
The groups of threads share a big register file that is split between the threads.
About GPUs: Register file

The number of registers used per-thread impact the number of resident group of threads on the machine (occupancy)
This in turn will impact the latency hiding capability

**About GPUs: Register file**

**VERY IMPORTANT!**
About GPUs: Spilling

The huge register file and number of concurrent threads makes spilling pretty costly
About GPUs: Spilling

Example (spilling 1 register): 1024 threads x 32-bit register = 4 KB!

The huge register file and number of concurrent threads makes spilling pretty costly.

Spilling is typically not an effective way of reducing register pressure to increase occupancy and should be avoided at all costs.
Pipeline
We support function calls and we try to exploit them.

Like most GPU programming models though, we can inline everything if we want.
Inlining

Not inlining showed significant speedup on some shaders where big functions were called multiple times.
Inlining

Dead Arg Elimination

Get rid of dead arguments to functions
Inlining

Dead Arg Elimination

Argument Promotion

Convert to pass by value as many objects as we can
Proceed to the actual inlining
Inlining decision based on standard LLVM inlining policy + custom threshold + additional constraints
Inlining

Objective of our inlining policy is to be very conservative while trying exploit cases where we can keep a function call can benefit us potentially a lot.

Custom policies try to minimize the impact that not inlining could have on other key optimizations for performance (SROA, Buffer preloading).

```plaintext
int function(int addrspace(stack)* v) {
    ...
}

int function(int addrspace(constant)* v) {
    ...
}
```

We force inline these cases.
The new IPRA support in LLVM has been key in avoiding pointless calling convention register store/reload.
We run it multiple times in our pipeline in order to be sure that we promote as many allocas to register values as possible.
int function(int i) {
    int a[4] = { x, y, z, w };  
    ...  
    ... = a[i];  
}
int function(int i) {
    int a[4] = { x, y, z, w };  
    ... = i == 0 ? x : (i == 1 ? y : i == 2 ? z : w);
}
Loop Unrolling

- SROA
- Alloca Opt
- Loop Unrolling
Loop Unrolling

```c
int a[5] = { x, y, z, w, q };  // int a[5] = { x, y, z, w, q };
int b = 0;               // int b = x;
for (int i = 0; i < 5; ++i) {
    b += a[i];         // b += y;
}                        // b += z;
                      // b += w;
                      // b += q;
```

Completely unrolling loops allows SROA to remove stack accesses

If we have dynamic memory access to stack or constant memory that we can promote to uniform memory we want to greatly increase the unrolling thresholds
We also keep track of register pressure

Our scheduler is very eager to try and help latency hiding by moving most of memory accesses at the top of the shader (and is difficult to teach it otherwise) so we limit unrolling when we detect we could blow up the register pressure

```c
for (int i = 0; i < 5; ++i) {
    float4 a = texture_fetch();
    float4 b = texture_fetch();
    float4 c = texture_fetch();
    float4 d = texture_fetch();
    float4 e = texture_fetch();

    // Math involving the above
}
```
Loop Unrolling

We allow partial unrolling if we detect a static loop count and the loop would be bigger than our unrolling threshold.
Speculation helps in creating bigger blocks for the scheduler to do a better job and reduces the total overhead introduced by small blocks.
Speculation helps in creating bigger blocks for the scheduler to do a better job
and reduces the total overhead introduced by small blocks.
GPUs are massively parallel, but often some computation in shader can be statically determined to be the same for all the threads.

Some of these patterns are really convenient or difficult for the shader writer to extract from the program.
void kernel(constant float4 *A,  
    constant bool *b  
    global float *C) {  
    float4 f_vec = *b ? *A : float4(1.0);  
    ... = f_vec * C[tid];  
}
Uniformity Hoisting

We can move such computation to a program that runs at a lower rate (once)

Even one instruction is a lot of parallel work saved
void kernel(constant float4 *A, constant bool *b
   global float *C) {
   const int a[5] = { 3, 2, 1, 4, 2 };  
   ... = a[i];
   }

Some stack arrays that are initialized and never stored to (and haven’t been optimized away previously) can be turned into global loads instead
File scope constants can be initialized more efficiently before running the program.

In the stack also the array is replicated for every thread, while in global memory the array memory is shared by all the threads.
When control-flow is unstructured (e.g., a block is controlled by multiple predecessors) execution on GPUs require some special handling
Our backend supports full execution of unstructured control-flow handled at MI-level with little overhead.

So we need only limited structurization (we require loops to be transformed in LoopSimplify form though).
For relatively small unstructured blocks employ structurization based on duplication
We thought about employing the LLVM StructurizeCFG pass, but the way it translated control-flow wasn’t optimal for us (Higher register pressure on avg, more control instructions)
We run a bunch of optimizations (multiple times) in between passes
Instruction Selection is one of the most expensive steps of our compilation pipeline.

We use lots of custom combines to extract performance from our hardware.
Instruction Selection

Takes between 15% to 35% of our compile time!

Instruction Selection is one of the most expensive steps of our compilation pipeline.

We use lots of custom combines to extract performance from our hardware.
Instruction Selection

Instruction Selection is one of the most expensive steps of our compilation pipeline.

We use lots of custom combines to extract performance from our hardware.
Plan is to switch to GlobalSel in the near future as our main compiler ISel.

The switch should give us a better infrastructure while improving compile time.
Scheduling

Scheduling is key for exploiting ILP, improve latency hiding and reducing power consumption by reducing register accesses

We try to achieve the above while being very careful at not to cause register pressure problems
Adding unrelated after memory accesses helps with in-thread latency hiding so that other instructions can be executed while the load or texture fetch results are ready.
Scheduling

load r1
load r2
add r5, r0, r3
mul r7, r3, r4
sub r6, r5, r4
add r3, r1, r6
mul r4, r2, r3

Interleaving independent operations to improve ILP

Forwarding instruction results help reducing register file traffic (lower power)

This is pretty standard scheduling
Many other target specific policies are enforced, all aimed at improving ILP, latency hiding and power (for example grouping instructions by type), all of this while battling with register pressure.

We are willing to spend a lot of compile time on scheduling.
Challenges
Compile-time and being a JIT

• Being JITs GPU compilers care about compile time very much
Compile-time and being a JIT

- Being JITs GPU compilers care about compile time very much
- We optimize our pipeline to obtain the best results with the least wasted compile time
Compile-time and being a JIT

Main offenders:

- Instruction Selection: 15% - 35% compile-time
- Scheduling: 5% - 15% compile-time
- Instruction combining: ~10% compile-time
- Register Allocation/Register Coalescing: ~10% compile-time
• Being JITs GPU compilers care about compile time very much

• We optimize our pipeline to obtain the best results with the least wasted compile time

• Having a custom pipeline often times creates problems as changing the order of the passes can unveil nasty bugs that used to be hidden …
Compile-time and being a JIT

• Being JITs GPU compilers care about compile time very much

• We optimize our pipeline to obtain the best results with the least wasted compile time

• Having a custom pipeline often times creates problems as changing the order of the passes can unveil nasty bugs that used to be hidden …

• We also reuse a single compiler instance for multiple compilations … this also uncovered some nasty bugs!
Register definitions

Tuples of 2

Tuples of 4
Some instructions support complex input/output operands loaded in contiguous registers

GPUs typically support register tuples with overlapping tuple elements sharing many RUs
This kind of register hierarchy generates a substantial amount of LLVM register definitions (one per each element of each tuple)

Tuples can go up to 16-wide on some architectures!
Algorithms that scale with the number or registers or iterate over all the registers containing a RU can take a hit

We had problem with IPRA implementation for example where in our case for determining the registers used by a function was $O(N^2)$ on the number of registers
Register pressure awareness

- LLVM has limited support for register pressure awareness
- IR passes largely ignore register pressure (example LICM)
- Machine-level has some register pressure estimation, but most passes care only if they are running out of registers
- For us increasing register pressure is potentially bad even if we don’t end up spilling as it reduces occupancy