Intro to register pressure in AMD compilers

Alessandro Fanfarillo, Nicholas Curtis

August 2022

Register pressure and performance

GPUs are throughput machines, appropriate for highly parallel workloads

Parallelism allows one to hide the high cost of memory accesses with computation

The performance of GPU kernels is highly dependent on “occupancy” (parallel running wavefronts)

Occupancy depends on the amount of resources (e.g., registers, LDS, etc.) used by a wavefront

Registers are scarce resources, register usage in each thread is an important factor that determines occupancy

If a single wavefront uses fewer registers, the hardware scheduler can run more wavefronts in parallel

The number of registers used by a wavefront is determined by the compiler’s register allocation phase

Register allocation depends on the instruction order produced by the instruction scheduler

The instruction order determines register pressure (RP)

Instruction scheduling must minimize RP to maximize occupancy

Register pressure and occupancy

Num VGPRs	Occupancy per EU	Occupancy per CU
<= 64	8 waves	32 waves
<= 72	7 waves	28 waves
<= 80	6 waves	24 waves
<= 96	5 waves	20 waves
<= 128	4 waves	16 waves
<= 168	3 waves	12 waves
<= 256	2 waves	8 waves
> 256 (+ spilling to scratch)	1 waves	4 waves

Compromises for register allocation

Register allocation is an NP-Hard problem

NP-Hard problems can usually be solved by compromising speed, generality or correctness (optimality, find the best feasible solution)

Compilers compromise on correctness by applying “heuristics” that produce mostly correct solutions

The compromise goes from “find best solution on most cases” to “find close to best solution on most cases”

Helping the compiler means massaging a “bad” case into one that the compiler optimizes reasonably well

Compilers not only optimize for occupancy but also for instruction-level parallelism (ILP, minimizing the schedule length)

The two are in conflict: higher ILP requires more registers, which decreases occupancy

In codes with high register pressure, obtaining a good ILP can make a difference on the final performance

Scratch Use

What is scratch, and why does my program use it?

Scratch is an address space on AMD GPUs that is roughly equivalent to “local memory” in CUDA (i.e., thread-local global memory with interleaved addressing) that is used for register spills/stack space

Where does scratch use come from?

Register spills: register allocation of variables is done at compile/linking time. If enough registers are not available, variables will be "spilled" to scratch memory. The compiler keeps track of variable liveness and tries to maximize reuse and minimize spilling (NP-Hard problem)

- Spilling is not always "bad". Enough occupancy can hide the extra, more costly, memory access

Memory objects: Any variable is naturally a memory object. The compiler may attempt to place it to registers or LDS, but this may not succeed (e.g., due to size of an object)

How to Identify Scratch Use and Spilling (1)

Looking at the ISA (assembly code) is the simplest way

roc-obj / disassembly *may* give you some insight but lacks metadata that can determine a spill versus scratch use

"--save-temps" generates ISA files with valuable statistics on register & scratch usage, and spills. However, in some (albeit rare) cases use of --save-temps can change the generated ISA

Direct indicators of spills (in --save-temps):

In the *hip-amdgcn-amd-amdhsa-gfxXXX.s file, look for:

- .sgpr_spill_count and .vgpr_spill_count fields
- When compiling with debugging symbols ("-g –ggdb"), look for comments (e.g., "; **4-Byte Folded Spill**")

Direct indicators of scratch use (in --save-temps):

- Look for a non-zero "; ScratchSize XX" in the metadata *after* the kernel in question

Indirect indicators of scratch use (also available in disassembly):

- Look for instructions like “buffer_store_dword v18, off, s[0:3], 0 offset:160”

How to Identify Scratch Use and Spilling (2)

An LLVM patch to print kernel usage information during the compilation phase is accepted https://reviews.llvm.org/D123878

Example: hipcc **-Rpass-analysis=kernel-resource-usage** --amdgpu-target=gfx906

targets/mnmd_smem_u_7r_3d_32x8x4/minimod.cu.hip:240:1: remark: Kernel Name: _Z24kernel_add_source_kernelPfxf [-Rpass-analysis=kernel-resource-usage]

__global__ void kernel_add_source_kernel(float *g_u, llint idx, float source) {

^

targets/mnmd_smem_u_7r_3d_32x8x4/minimod.cu.hip:240:1: remark: SGPRs: 7 [-Rpass-analysis=kernel-resource-usage]

targets/mnmd_smem_u_7r_3d_32x8x4/minimod.cu.hip:240:1: remark: VGPRs: 2 [-Rpass-analysis=kernel-resource-usage]

targets/mnmd_smem_u_7r_3d_32x8x4/minimod.cu.hip:240:1: remark: ScratchSize [bytes/thread]: 0 [-Rpass-analysis=kernel-resource-usage]

targets/mnmd_smem_u_7r_3d_32x8x4/minimod.cu.hip:240:1: remark: Occupancy [waves/SIMD]: 10 [-Rpass-analysis=kernel-resource-usage]

targets/mnmd_smem_u_7r_3d_32x8x4/minimod.cu.hip:240:1: remark: SGPRs Spill: 0 [-Rpass-analysis=kernel-resource-usage]

targets/mnmd_smem_u_7r_3d_32x8x4/minimod.cu.hip:240:1: remark: VGPRs Spill: 0 [-Rpass-analysis=kernel-resource-usage]

targets/mnmd_smem_u_7r_3d_32x8x4/minimod.cu.hip:240:1: remark: LDS Size [bytes/block]: 0 [-Rpass-analysis=kernel-resource-usage]

For OpenMP: export LIBOMPTARGET_KERNEL_TRACE=1

How to Reduce Register Spilling (1)

- Avoid allocating data on the stack in a kernel

  Memory allocated on the stack lives in scratch, and *may* be optimized into registers

- Avoid passing big object as argument of kernels

  Function arguments are allocated on the stack and *may* be optimized into registers as an optimization

- Avoid writing large kernels with many function calls (including math functions and assertions)
  - All device functions are currently inlined. A kernel calling many device functions can become very big
- Keep loop unrolling under control
- Move variables declaration/assignment close to their use
- Manually spill to LDS

How to Reduce Register Spilling (2)

- Use __launch_bounds__ to suggest the compiler how many work items will be used for a particular kernel

Example: __global__ void __launch_bounds__ (256, 1) my_kernel(…)

First argument: MAX_THREADS_PER_BLOCK (default 1024)

Second argument: MIN_WARPS_PER_EU (default 1)

- Try to understand which specific class of registers is spilling (e.g., SGPRs vs VGPRs) and try to fix to root cause in the code.
  - Vector registers (VGPR): Storage for variables with (potentially) unique values for each thread in a wave. Almost all local variables will end up in VGPRs
  - Scalar registers (SGPR): Storage for variables that are uniform across all threads in a wave (e.g., a kernel argument, a pointer, constant buffers, etc.)
  - SGPRs can only spill into VGPRs. If SGPRs count is high, fixing the root cause may reduce spilling in scratch coming from VGPRs

Is Register Spilling Always Bad?

Low occupancy is just one factor that may impact performance

There are cases where accepting some register spilling can dramatically improve performance

Example SW4CK:

- Kernel 1 takes about **12 ms** to run on 1 GCD of MI250X, register pressure is high (232 VGPRs), occupancy of 2 waves/EU with no spilling.
- Unroll hot loop to let the compiler avoid superfluous computation. Increased register pressure even more: 256 VGPRs + small spilling to scratch.
- Performance improved by almost 50%. New time **6.81 ms**

Conclusions

- Scratch use is unavoidable in most cases; compiler tries to place objects in registers and LDS as an optimization
  - Determine if scratch use is actually your bottleneck
- Passing big structs/objects (structs of structs with statically allocated object/arrays) as arguments should be avoided
- Use __launch_bounds__ for problematic kernels to allow the compiler to use more registers per thread (need to balance occupancy and spilling)
  - Setting this too restrictive can cause the compiler to run out of registers and fail
  - In practice it is often useful to identify "classes" of kernels and apply launch bounds to groups of kernels as opposed to tuning individual kernels
  - Default is 1024. 256 or 512 are generally the options to try for high scratch kernels
- Keep in mind that device functions are currently always inlined and may increase register pressure/spilling
- Loop unrolling influences register pressure
- Move variables declaration/assignment close to their use
- Manually spill to LDS variables with longest liveness

Disclaimer

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions, and typographical errors. The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. Any computer system has risks of security vulnerabilities that cannot be completely prevented or mitigated. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

THIS INFORMATION IS PROVIDED 'AS IS." AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS, OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION. AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY RELIANCE, DIRECT, INDIRECT, SPECIAL, OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Third-party content is licensed to you directly by the third party that owns the content and is not licensed to you by AMD. ALL LINKED THIRD-PARTY CONTENT IS PROVIDED "AS IS" WITHOUT A WARRANTY OF ANY KIND. USE OF SUCH THIRD-PARTY CONTENT IS DONE AT YOUR SOLE DISCRETION AND UNDER NO CIRCUMSTANCES WILL AMD BE LIABLE TO YOU FOR ANY THIRD-PARTY CONTENT. YOU ASSUME ALL RISK AND ARE SOLELY RESPONSIBLE FOR ANY DAMAGES THAT MAY ARISE FROM YOUR USE OF THIRD-PARTY CONTENT.

The OpenMP name and the OpenMP logo are registered trademarks of the OpenMP Architecture Review Board.


AMD