Profiling guide – Novice level

Alessandro Fanfarillo

AMD
together we advance_

Introduction

- Main material taken from blog post: https://rocm.blogs.amd.com/software-tools-optimization/profiling-guide/novice/README.html
- Assumptions for “Novice” level:
  1. The app leverages a GPU for computation, and you understand the basic purpose of each GPU kernel in your code
  2. You recognize that moving data between the CPU and GPU has a cost, even if terms like “latency-bound” or “memory-bound” are unfamiliar
- Expected outcome of this training talk:
  - Learn how to measure where your application spends time on the GPU, including kernel execution and data transfers
  - Discover how to pinpoint basic performance limitations (e.g., poor GPU occupancy, high memory traffic)
  - Begin exploring why performance may differ across GPU architectures, even when specifications seem similar

The code

- 2D Shallow-water solver, finite-difference for spatial derivatives, RK4 for time stepping
- Single precision
- Accuracy checks
- Performance metric returned at the end
- Designed as a playground, not real application

Initial performance:

Domain: 512x512, steps=500, dt=0.0728643

Elapsed: 0.154 s | Throughput (including RK4 stages): **3409.97 MCUPS**

Mass: initial=2.626466547e+05, final=2.626464585e+05, rel.err=7.471e-07

Min(h) after run: 0.981776

First step: rocprofv3

rocprofv3 --kernel-trace --stats -S -T -d outdir -o shallow -- ./shallow

NAME	DOMAIN	CALLS	DURATION (nsec)	AVERAGE (nsec)	PERCENT (INC)	MIN (nsec)	MAX (nsec)	STDDEV
compute_rhs	KERNEL_DISPATCH	2000	26948810	1.347e+04	33.591140	12480	15200	3.689e+02
update_stage	KERNEL_DISPATCH	1500	21177442	1.412e+04	26.397248	13120	16000	4.198e+02
apply_reflect_bc	KERNEL_DISPATCH	2001	17278248	8.635e+03	21.536983	7520	11680	4.578e+02
final_update	KERNEL_DISPATCH	500	14813126	2.963e+04	18.464258	28640	32000	5.007e+02
init_gaussian	KERNEL_DISPATCH	1	8320	8.320e+03	0.010371	8320	8320	0.000e+00

From this step, we learn where time gets spent, and kernel names. Now let's see how busy the CUs are:

rocprofv3 --pmc OccupancyPercent -T -d outdir -o shallow -- ./shallow

1,1,1,1,3069691,3069691,262144,5,"init_gaussian",256,0,0,12,4,32,"OccupancyPercent",46.601604,1011230466808020,1011230466816020

2,2,1,1,3069691,3069691,512,4,"apply_reflect_bc",256,0,0,24,0,32,"OccupancyPercent",1.32989986e-01,1011230466910098,1011230466919058

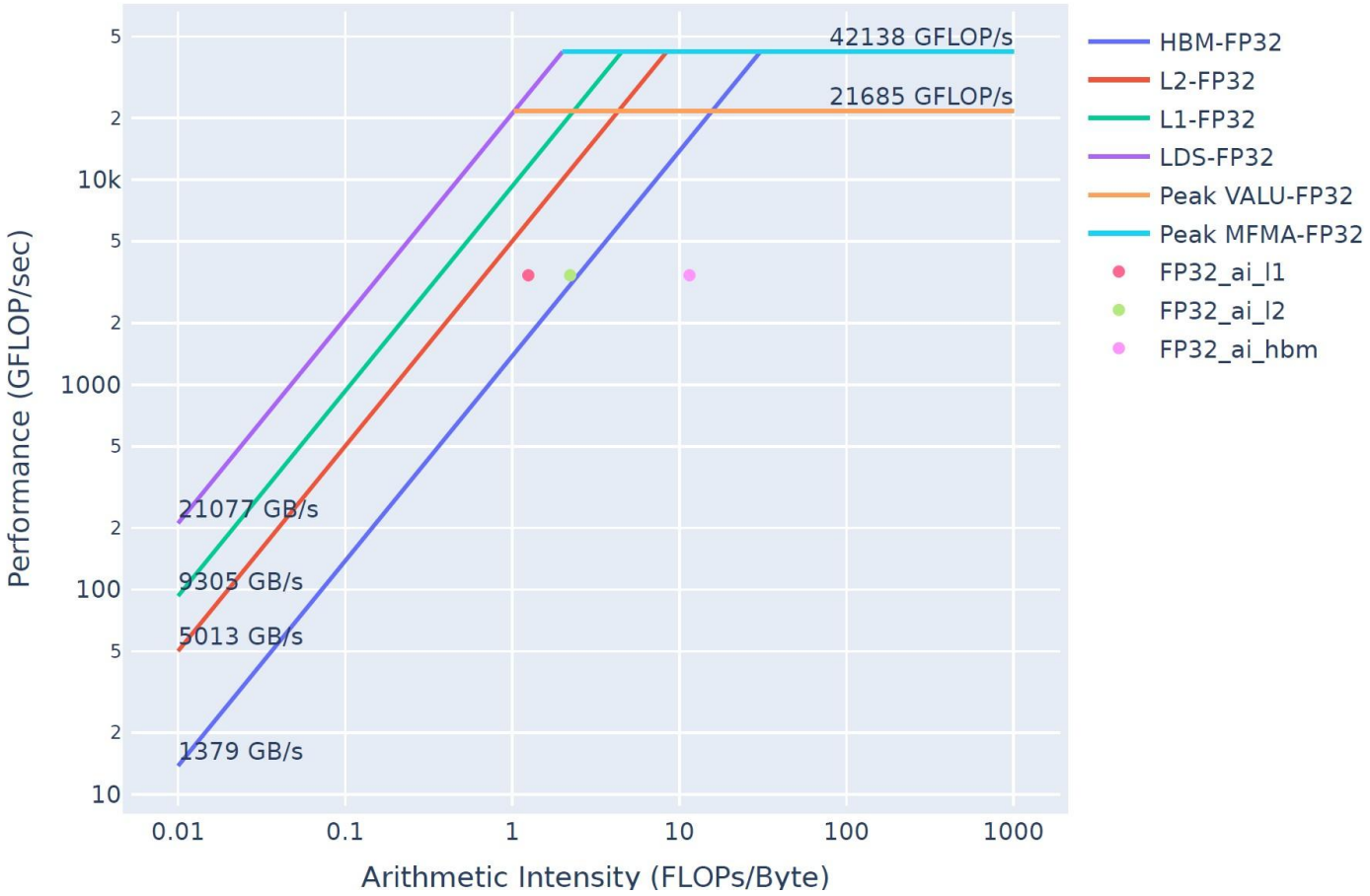
3,3,1,1,3069691,3069691,512,4,"apply_reflect_bc",256,0,0,24,0,32,"OccupancyPercent",1.19206811e-01,1011230483265617,1011230483277457

4,4,1,1,3069691,3069691,262144,3,"compute_rhs",256,0,0,32,0,32,"OccupancyPercent",51.798800,1011230483422097,1011230483436977

5,5,1,1,3069691,3069691,262144,2,"update_stage",256,0,0,16,0,32,"OccupancyPercent",50.337635,1011230483568977,1011230483584337

6,6,1,1,3069691,3069691,512,4,"apply_reflect_bc",256,0,0,24,0,32,"OccupancyPercent",1.38614879e-01,1011230483711377,1011230483721297

First roofline of compute_rhs



Compute_rhs has some compute but it is still memory bound

The distance from roofline communicates a series of possible things:

1. Inefficient memory accesses
2. Insufficient parallelism to hide latency

The low occupancy and the roofline strongly suggest that there is not enough parallelism to hide the latency costs of accessing memory

Let's try to increase the domain size

From 512x512 to 2048x2048

Check Occupancy after domain size increase

1,1,1,1,3072041,3072041,4194304,5,"init_gaussian",256,0,0,12,4,32,"OccupancyPercent",62.256694,1011333016314232,1011333016402392

2,2,1,1,3072041,3072041,2048,3,"apply_reflect_bc",256,0,0,32,0,32,"OccupancyPercent",5.61374392e-01,1011333016467032,1011333016477912

3,3,1,1,3072041,3072041,2048,3,"apply_reflect_bc",256,0,0,32,0,32,"OccupancyPercent",4.95447153e-01,1011333044785613,1011333044800653

4,4,1,1,3072041,3072041,4194304,4,"compute_rhs",256,0,0,32,0,32,"OccupancyPercent",87.123946,1011333044846093,1011333045024333

5,5,1,1,3072041,3072041,4194304,2,"update_stage",256,0,0,16,0,32,"OccupancyPercent",91.281051,1011333045067053,1011333045257454

6,6,1,1,3072041,3072041,2048,3,"apply_reflect_bc",256,0,0,32,0,32,"OccupancyPercent",5.47154444e-01,1011333045296014,1011333045306894

7,7,1,1,3072041,3072041,4194304,4,"compute_rhs",256,0,0,32,0,32,"OccupancyPercent",86.724880,1011333045351214,1011333045527854

By increasing the domain size, we provide enough work to hide the latency from memory access

Domain: 2048x2048, steps=500, dt=0.0728643

Elapsed: 0.965 s | Throughput (including RK4 stages): **8695.79 MCUPS**

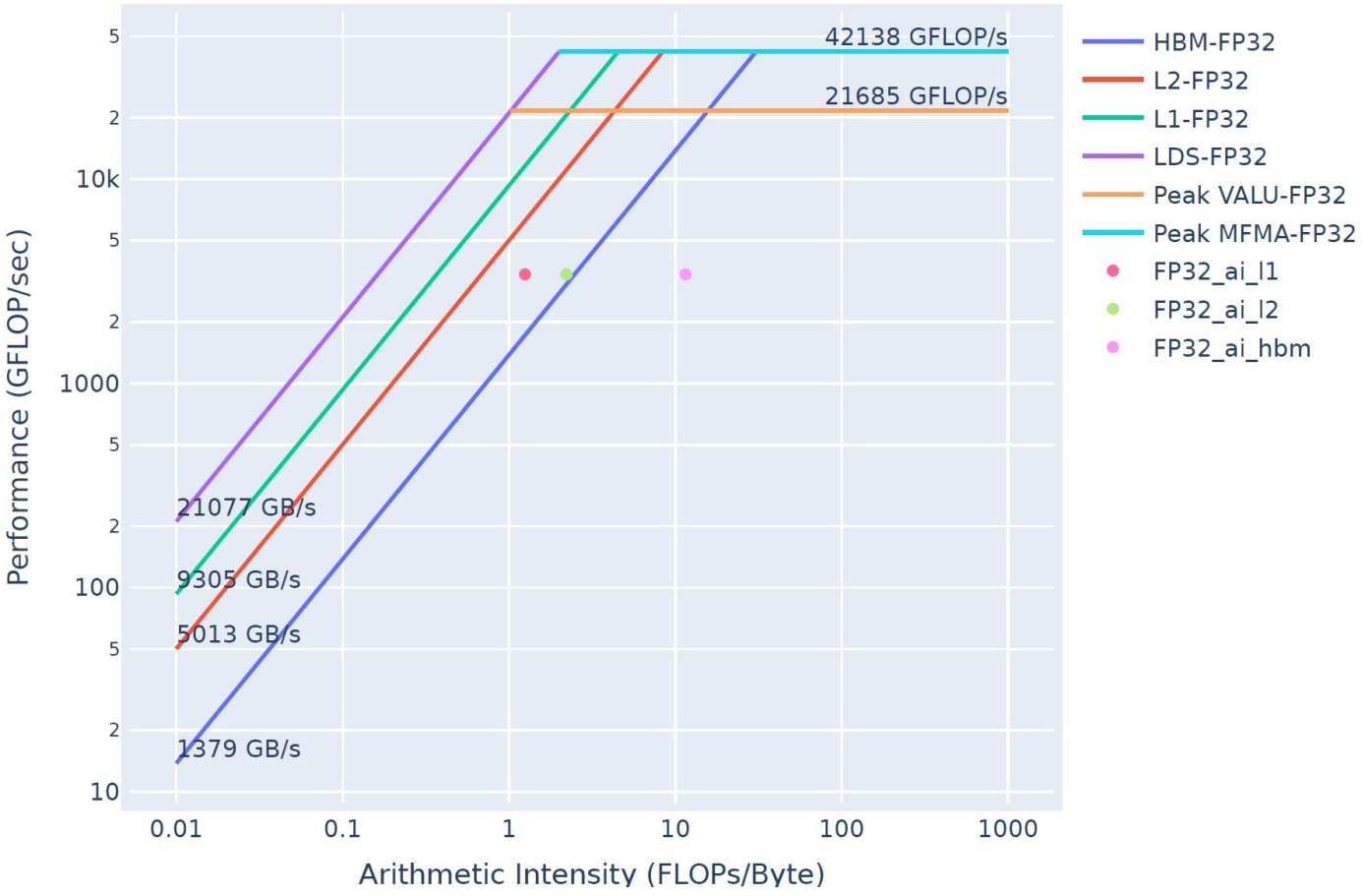
Mass: initial=4.194806655e+06, final=4.194806458e+06, rel.err=4.678e-08

Min(h) after run: 0.981776

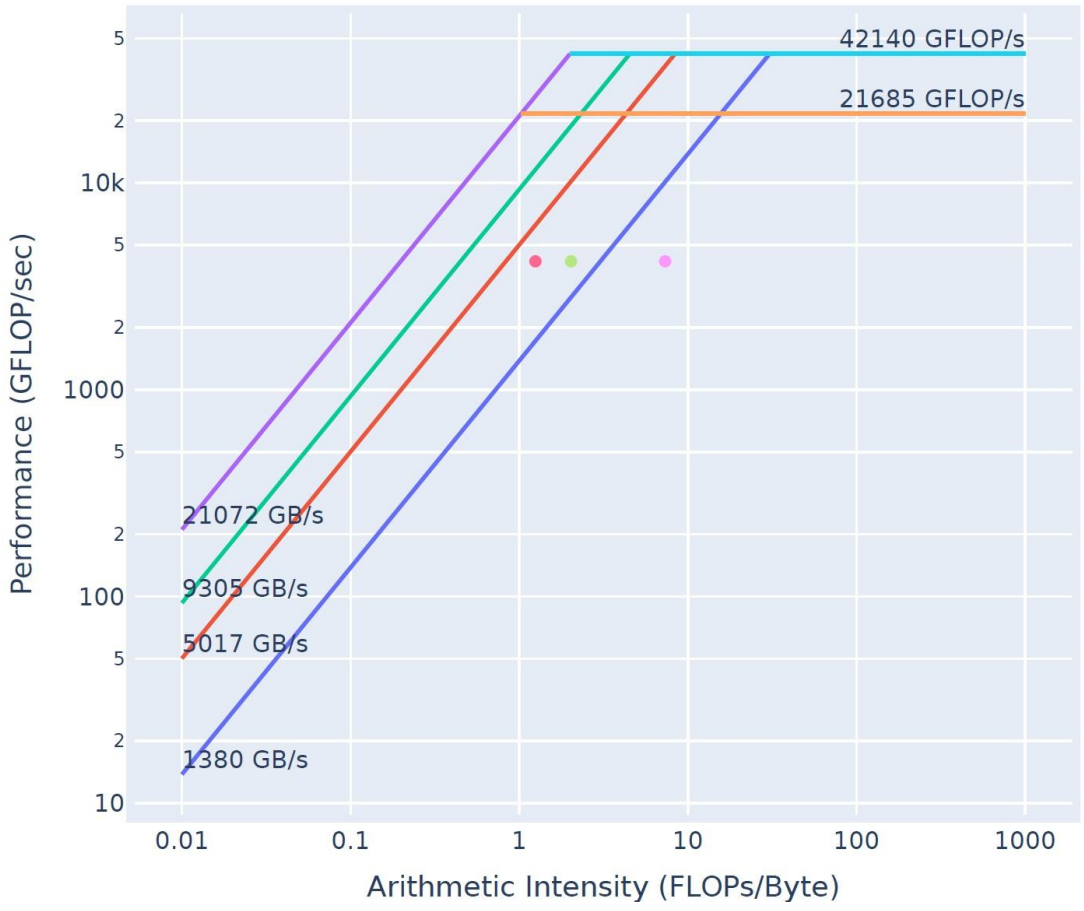
Increased from 3049 to 8696 MCUPS – 2.85x faster

Roofline before and after domain size increase

Before



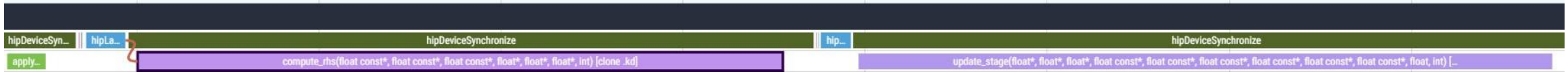
After



Check hiptrace from rocprofv3

For an application developer, it is very helpful to visually see the host runtime API activity and all device activities on a timeline trace

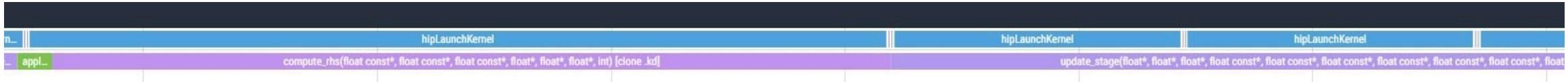
rocprofv3 --kernel-trace --hip-trace --output-format pftrace -d outdir -o shallow -- ./shallow



These gaps after kernels are due to the presence of hipDeviceSynchronize() and they add up to the global runtime

Because we are only working on a single stream, the kernels' executions are already sequentially executed

We can completely remove the hipDeviceSynchronize() after each kernel launch



This change improves the overall performance without impacting correctness

New performance:

Domain: 2048x2048, steps=500, dt=0.0728643

Elapsed: 0.887 s | Throughput (including RK4 stages): **9452.20 MCUPS**

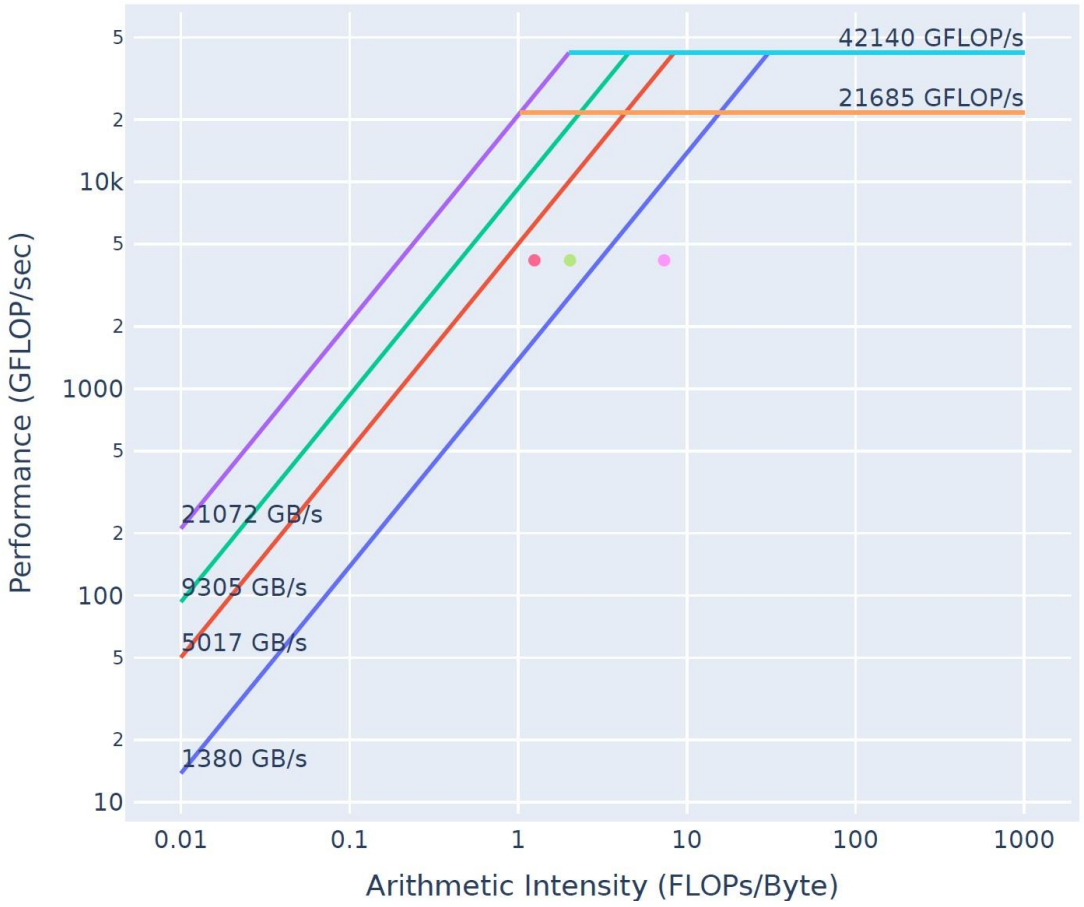
Mass: initial=4.194806655e+06, final=4.194806458e+06, rel.err=4.678e-08

Min(h) after run: 0.981776

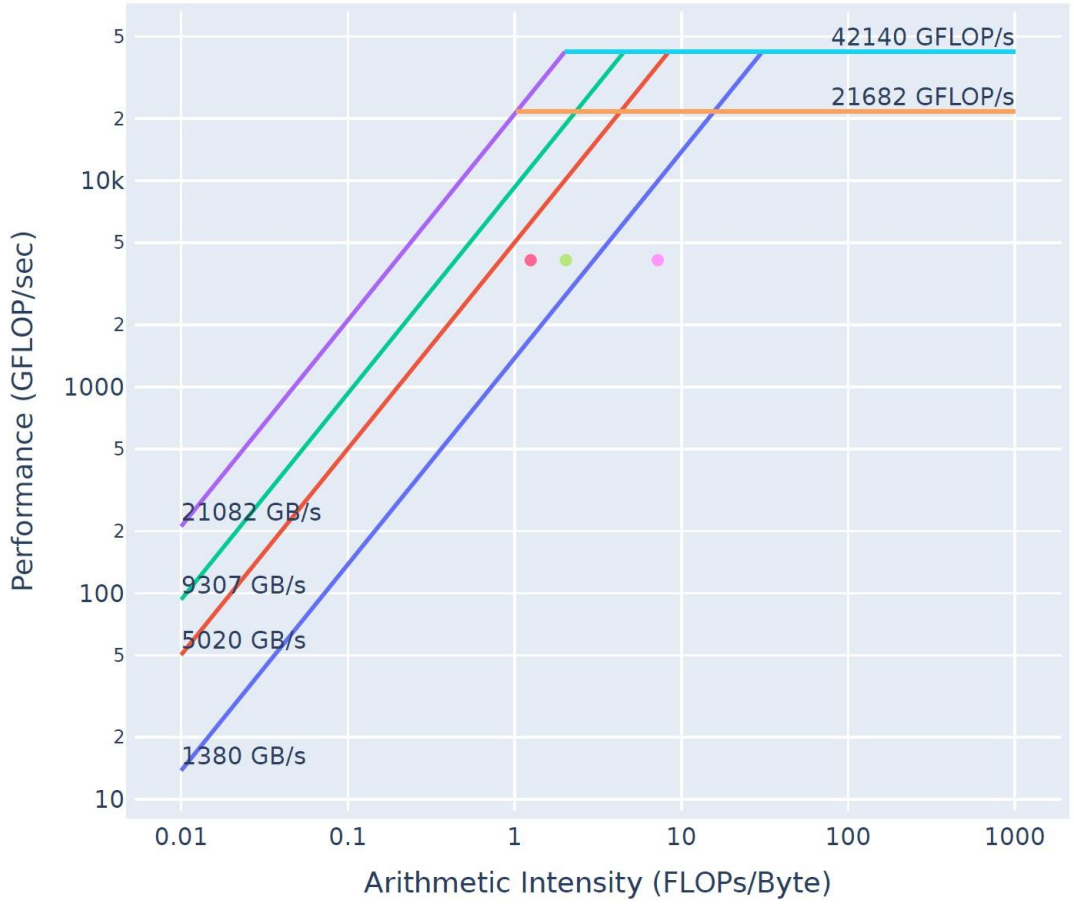
From 8696 to 9452 MCUPS – 1.09x faster

Check roofline after removing hipDeviceSynchronize()

Before



After



No change. Next step is to optimize the code. Let's look at more metrics

Check VALUBusy and new performance

1,1,1,1,818317,818317,4194304,5,"init_gaussian",256,0,0,12,4,32,"VALUBusy",37.622438,1606927748825612,1606927748894892

2,2,1,1,818317,818317,2048,4,"apply_reflect_bc",256,0,0,32,0,32,"VALUBusy",6.43171047e-02,1606927748956652,1606927748968012

3,3,1,1,818317,818317,2048,4,"apply_reflect_bc",256,0,0,32,0,32,"VALUBusy",4.93171973e-02,1606927776662430,1606927776676990

4,4,1,1,818317,818317,4194304,3,"compute_rhs",256,0,0,32,0,32,"VALUBusy",54.070862,1606927776708350,1606927776884831

5,5,1,1,818317,818317,4194304,2,"update_stage",256,0,0,16,0,32,"VALUBusy",7.308252,1606927776918751,1606927777110431

6,6,1,1,818317,818317,2048,4,"apply_reflect_bc",256,0,0,32,0,32,"VALUBusy",6.62305718e-02,1606927777143871,1606927777154591

VALUBusy measures the percentage of GPUTime vector ALU instructions are processed (ideally close to 100%)

One way of potentially improving the performance of a stencil kernel is to use a large thread block (better caching, etc.)

Use tiles of 32x32 instead of 16x16. New performance:

Domain: 2048x2048, steps=500, dt=0.0728643

Elapsed: 0.735 s | Throughput (including RK4 stages): **11417.18** MCUPS

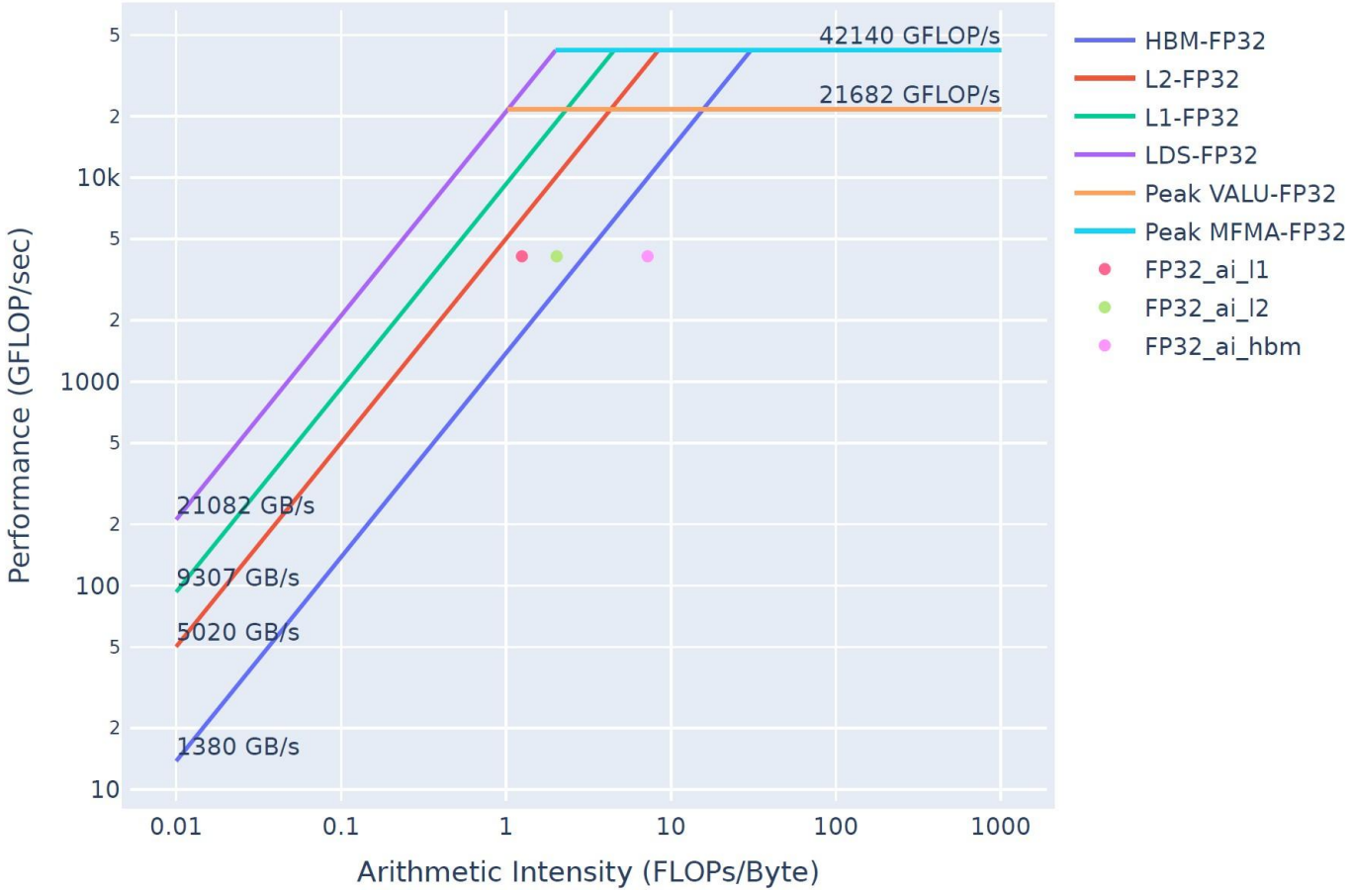
Mass: initial=4.194806655e+06, final=4.194806458e+06, rel.err=4.678e-08

Min(h) after run: 0.981776

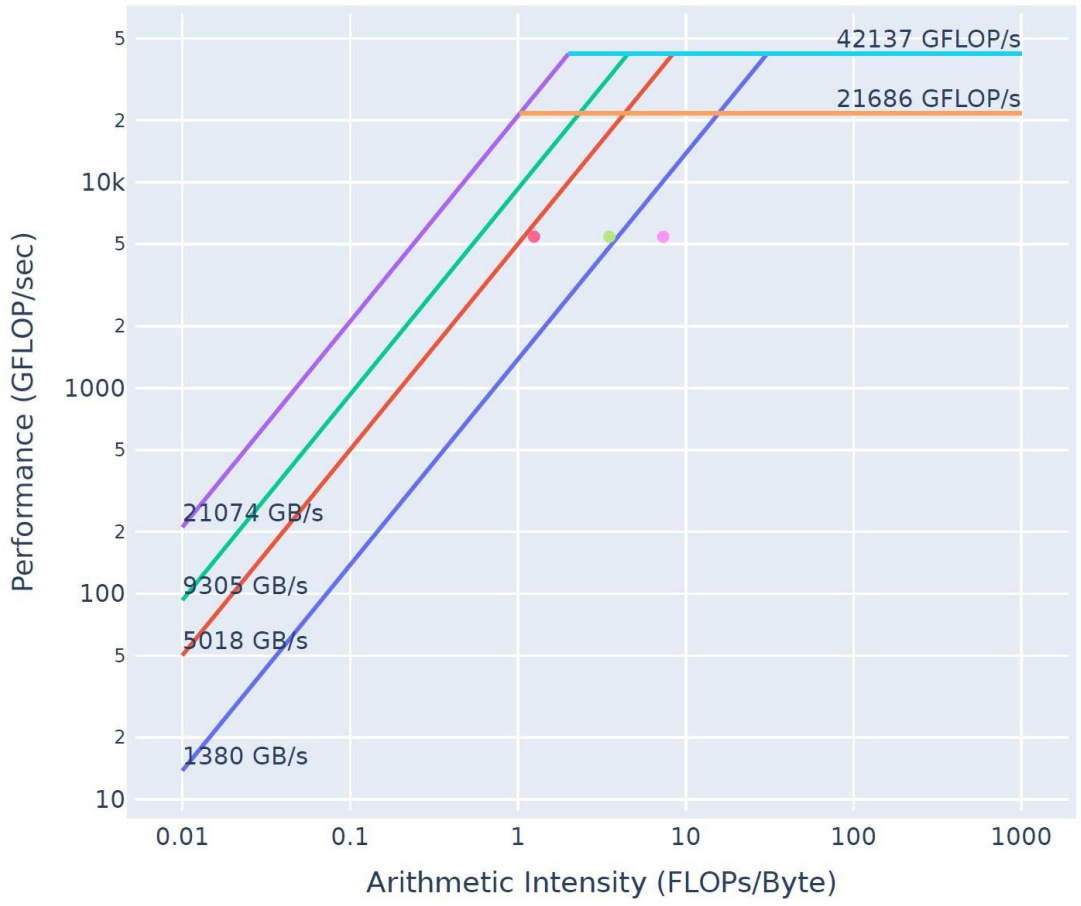
From 9452 to 11417 MCUPS – 1.21x faster

Check roofline block size from 16x16 to 32x32

Before



After



Check VALUBusy after optimization

1,1,1,1,820965,820965,4194304,5,"init_gaussian",1024,0,0,12,4,32,"VALUBusy",51.151937,1615275137641414,1615275137692294

2,2,1,1,820965,820965,2048,4,"apply_reflect_bc",256,0,0,32,0,32,"VALUBusy",6.51420313e-02,1615275137760134,1615275137771174

3,3,1,1,820965,820965,2048,4,"apply_reflect_bc",256,0,0,32,0,32,"VALUBusy",4.95911257e-02,1615275165596957,1615275165611677

4,4,1,1,820965,820965,4194304,3,"compute_rhs",1024,0,0,32,0,32,"VALUBusy",70.718763,1615275165640957,1615275165775997

5,5,1,1,820965,820965,4194304,1,"update_stage",1024,0,0,16,0,32,"VALUBusy",8.445584,1615275165810397,1615275165975997

VALUBusy increased to 71% from 50% by just setting a larger tile size

Can we do better? Reading long, consecutive, memory segments has the potential to bring the most benefits.

L1 cache hit is particularly important, using a rectangular grid instead of square improves the overall performance

Using a block size of 64x4:

Domain: 2048x2048, steps=500, dt=0.0728643

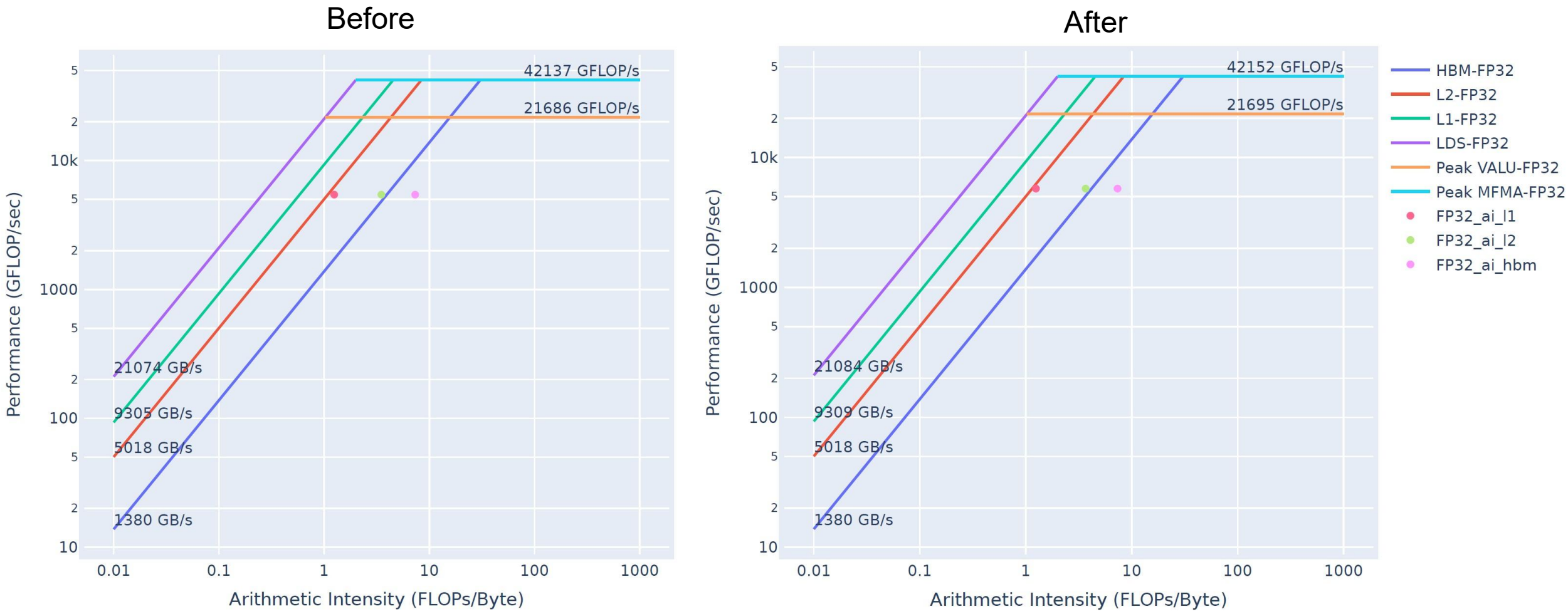
Elapsed: 0.654 s | Throughput (including RK4 stages): **12829.58** MCUPS

Mass: initial=4.194806655e+06, final=4.194806458e+06, rel.err=4.678e-08

Min(h) after run: 0.981776

From 11417 to 12830 MCUPS – 1.12x faster

Check roofline block size from 32x32 to 64x4



Conclusions

- Final speedup: 3.76x
- Profiling is an iterative process
- Profiling tools, no matter how powerful, do not tell the whole story
- A combination of tools and intuition is needed to build solid understanding
- How to optimize a HIP kernel to achieve a certain goal (e.g., improve occupancy) remains complicated
- Knowing the GPU architecture helps a lot in understanding the most successful optimizations
- AI can help with intelligent suggestions and code changes to try out

DISCLAIMERS AND ATTRIBUTIONS

The information contained herein is for informational purposes only and is subject to change without notice. While every precaution has been taken in the preparation of this document, it may contain technical inaccuracies, omissions and typographical errors, and AMD is under no obligation to update or otherwise correct this information. Advanced Micro Devices, Inc. makes no representations or warranties with respect to the accuracy or completeness of the contents of this document, and assumes no liability of any kind, including the implied warranties of noninfringement, merchantability or fitness for particular purposes, with respect to the operation or use of AMD hardware, software or other products described herein. No license, including implied or arising by estoppel, to any intellectual property rights is granted by this document. Terms and limitations applicable to the purchase or use of AMD's products are as set forth in a signed agreement between the parties or in AMD's Standard Terms and Conditions of Sale. GD-18

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.

AMD, the AMD Arrow logo, Radeon, Instinct, ROCm, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.


AMD