13th International Workshop on OpenCL and SYCL

IWOCL 2025

Adaptivity in AdaptiveCpp: Optimizing Performance by Leveraging Runtime Information During JIT-Compilation

Aksel Alpay, Universität Heidelberg

Aksel Alpay, Vincent Heuveline, Universität Heidelberg

April 7-11, 2025 | Heidelberg, Germany | iwocl.org

Often, performance can be increased by generating kernels that make more assumptions about the actual execution context in which they are invoked – more **specialized** kernels.

- E.g. a kernel that assumes a certain alignment of input pointers may perform better – at the expense of generality.
- The loss of generality can be solved by automatically generating such kernels at runtime in a JIT-compilation scenario.
- In this case, the tradeoff is potentially more performance for potentially more JIT-overhead.
- In this study, we refer to such automatic specializations at JIT-time as **JIT-time optimizations**.

JIT-time optimizations

- JIT-time optimizations are not a new idea and have been explored in the context of other programming models (e.g. OpenCL[1], OpenMP[2])
- Code patterns between programming models may diverge; some JIT-time optimizations may address aspects specific to one programming model.
  - Results for other programming models cannot always be transferred to SYCL in a straight-forward way.
- Implementing such a functionality in SYCL is challenging:
  - Multi-backend architecture of SYCL, no well-defined intermediate representation across all backends
  - Unlike e.g. OpenCL, SYCL is not inherently designed around JIT
- A SYCL compiler that wishes to support JIT-time optimizations must be deliberately designed for that purpose.

[1] Jääskeläinen et al. 2015. pocl: A Performance-Portable OpenCL Implementation.

[2] Tian et al. 2022. Just-in-Time Compilation and Link-Time Optimization for OpenMP Target Offloading.

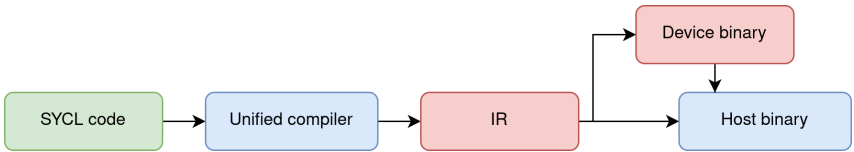
Our contributions

Adaptivity in AdaptiveCpp: Framework for automatic JIT-time optimization/kernel specialization in AdaptiveCpp

- First framework for automatic JIT-time optimization in SYCL;
- A production implementation in AdaptiveCpp, supporting CPUs, NVIDIA GPUs, AMD GPUs, Intel GPUs in a backend-independent manner;
- A performance evaluation of our solution, demonstrating substantial performance gains, often outperforming vendor compilers.
- A framework that addresses gaps in standard SYCL without needing changes in user code
  - Many of the optimizations that we propose cannot easily be replicated in standard SYCL
  - Attempting to do so may lead to non-idiomatic, overly complex, less general or unportable code

Implemented in AdaptiveCpp

- Open-source compiler stack supporting multiple programming models, including SYCL, C++ standard parallelism offloading and a CUDA dialect: `https://github.com/adaptivecpp/adaptivecpp`
- Only SYCL compiler with a unified host-device compiler an a unified code representation (based on LLVM IR) across backends:



- Unified JIT-compiler allows implementation of JIT-time optimizations in a backend-independent manner
- Supports any LLVM-supported CPU, NVIDIA GPUs, AMD GPUs, Intel GPUs

A framework for JIT-time optimizations

How to avoid excessive JIT compilation overhead?

Design the framework such that after a finite amount of kernel invocations or application runs, no JIT is needed anymore:

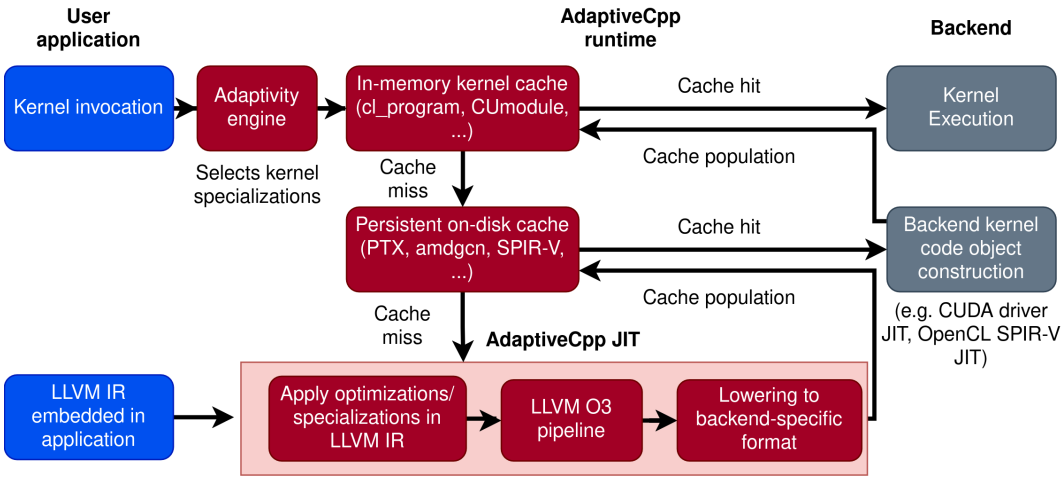
1. **Optimization phase** $\rightarrow$ new specialized kernels are being generated
2. **Stable phase** $\rightarrow$ no kernels generated anymore, peak performance reached.

Application will **eventually** be free of JIT overheads $\rightarrow$ overheads can be amortized

In practice: We need to be able to return to the optimization phase from the stable phase!

- Application kernel usage patterns may change, e.g. due to user input!

A framework for JIT-time optimizations: Architecture



A framework for JIT-time optimizations: Architecture

- Persistent on-disk cache is crucial!
  - Allows amortizing JIT-overheads across multiple application runs
- Certain optimizations need persistent statistics → **appdb**
  - per-application database for metadata on kernels and their usage patterns
- Mechanism for users to control aggressiveness of JIT-time optimizations:
  - `ACPP_ADAPTIVITY_LEVEL=0` (AL0): All JIT-time optimizations disabled. Number of JIT-compilations is minimized (JIT-compiles entire device translation units at once).
  - `ACPP_ADAPTIVITY_LEVEL=1` (AL1, default): All JIT-time optimizations are enabled that are expected to typically reach stable phase by second kernel invocation. JIT-compiles every kernel separately.
  - `ACPP_ADAPTIVITY_LEVEL=2` (AL2): Additionally enable JIT-time optimizations that may need multiple kernel launches to reach stable phase.

Implemented optimizations

Known Group Size (AL1)

Observation: Typical applications often invoke a given kernel with the same work group size.
Idea: Specialize kernel for the given group size as part of AL1.
Implementation: Replace all uses of group size with a constant in IR, assert to optimizer that all local ids are smaller than given group size, inform compiler backend of group size.
Potential benefit: Simplified calculations involving group size, perhaps free registers, better register scheduling.
Equivalent in standard SYCL? Effect similar to `sycl::reqd_work_group_size` attribute.

- The attribute however requires knowledge of group size at compile time
- less flexible and may not always be viable (e.g. imagine a SYCL library where the group size decision is controlled by user code)

Global Size Fits In Int32 (AL1)

Observation: SYCL APIs for global ids/ranges rely on 64-bit ints. But in many cases, the global range fits into 32-bit integers.
Idea: Specialize kernels based on whether the global range fits in 32-bit integers.
Implementation: If global size fits in 32-bit, internally implement `get_global_id()`, `get_global_linear_id()` etc builtins using 32-bit arithmetic, add `llvm.assume(x < UINT_MAX)`
Potential benefit: Simplified calculation of global ids/ranges, potentially using instructions for smaller data types.
Equivalent in standard SYCL? Requires manually reimplementing `get_global_id()` and similar builtins, nonidiomatic code. `__builtin_assume()` not guaranteed to exist by the SYCL specification.

No, this optimization does not impact correctness!

Local memory specialization (AL1)

Observation: Local memory usage typically depends directly on group size. (“X bytes of memory per work-item”)
Idea: If we can specialize the group size, we can also specialize the internal configuration (e.g. range) of `local_accessor` objects.
Implementation: Hardwire internals of `local_accessor` for the actual requested local accessor size.
Potential benefit: Simplified calculation of indices in local accessors, save registers
Equivalent in standard SYCL? None; requires modifying internals of local accessor objects.

Alignment specialization (AL1)

Observation: For a given pointer kernel parameter, the alignment of the pointer data typically does not change – especially since commonly, pointers to start of allocations are passed in

Idea: Specialize kernels for the alignment of pointer kernel arguments

Implementation: Attach information about actual alignment of pointer data to LLVM IR

Potential benefit: Perhaps vector load/stores, better vectorization etc

Equivalent in standard SYCL? `std::assume_aligned()` only available in C++ 20, requires providing alignment at compile-time, which may not always be viable (e.g. API accepting pointers from user code)

Automatic noalias (AL1)

Observation: Whether two pointer kernel arguments alias typically does not change
Idea: Detect pointers that don't alias other pointer arguments, and specialize kernels based on that.
Implementation: Attach LLVM IR `noalias` attribute to pointers qualifying for this property
Potential benefit: Reordering of load/stores, better memory access patterns, caching loads, better vectorization
Equivalent in standard SYCL? Not really - `restrict` attribute not guaranteed to be available in SYCL, and even if it is, the restrict information may be dropped when arguments are stored in the kernel lambda object.

Automatic noalias (AL1)

How to detect pointer arguments that qualify for `noalias`?

In general difficult, but one special case can be detected:

- If there is no indirect access in the kernel (needs compiler check) and
- the allocation pointed to by a pointer kernel argument is not also pointed to by another pointer argument.

Implementation:

- `ACPP_ALLOCATION_TRACKING=1` enables tracking of allocations in a radix tree → efficient finding of allocation base for any pointer within allocation
- Compiler check not yet implemented → **This optimization is currently not enabled by default and should be considered a preview!**
- (the applications tested in this study do not use indirect access in a problematic way, so the optimization is safe here)

Automatic argument specialization (AL2)

Observation: For some applications, certain runtime kernel argument values do not change (e.g. problem size)

Idea: Detect argument values that remain mostly invariant, and specialize the kernel for those

Implementation:

- Non-trivial heuristics (see paper for details), here only rough idea
- Statistical data stored in appdb about frequency of individual kernel argument values
- Specialization occurs if frequency (including previous application runs) exceeds relative threshold (default: >= 0.8)

Potential benefit: Depending on how kernel argument is used, e.g. computation simplification, control flow simplification, saving registers, …

Equivalent in standard SYCL? This corresponds to utilizing SYCL 2020 specialization constants automatically where it makes sense.

Summary: Optimizations

- All AL1 optimizations are carefully chosen and designed such that typically, they do not increase the number of needed JIT compilations!
  - If the backend already does JIT (e.g. OpenCL/SPIR-V), no fundamental change in JIT behavior for AL1.
- Only AL2 optimization is automatic argument specialization – higher risk, higher reward.
- Many of these JIT-time optimizations cannot be replicated easily (or at all) in standard SYCL.

Evaluation

Experimental setup

Test hardware from 3 vendors, of different ages $\rightarrow$ sample as broad set of hardware as possible.

- NVIDIA RTX A5000
- AMD Radeon Pro VII
- Intel UHD 630 iGPU

Various benchmarks from different scientific disciplines:

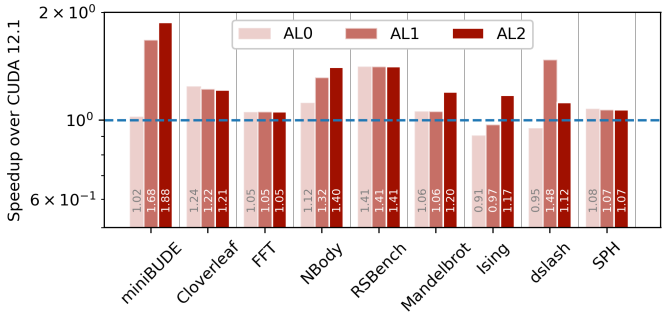
- mini-apps CloverLeaf, miniBUDE
- FFT, NBody, RSBench, Mandelbrot, Ising, dslash, SPH taken from HeCBench
- Measured performance data as reported by benchmarks - includes host-side overheads (e.g. JIT).

See paper for details on software stack.

Overall performance: Performance relative to native vendor models

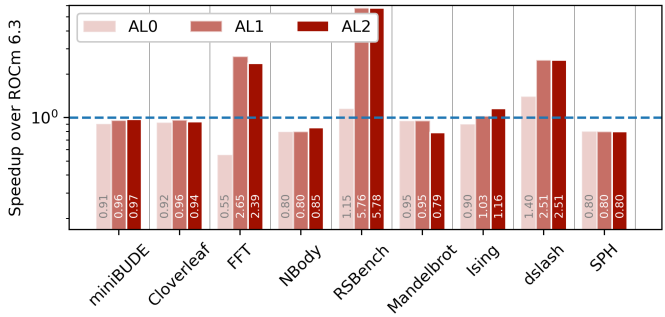
- Compare to nvcc-compiled CUDA on NVIDIA
- Compare to hipcc-compiled HIP on AMD
- Compare to oneAPI icpx-compiled SYCL on Intel
- Ran benchmarks until stable phase is reached, i.e. no JIT overheads anymore

Performance on NVIDIA



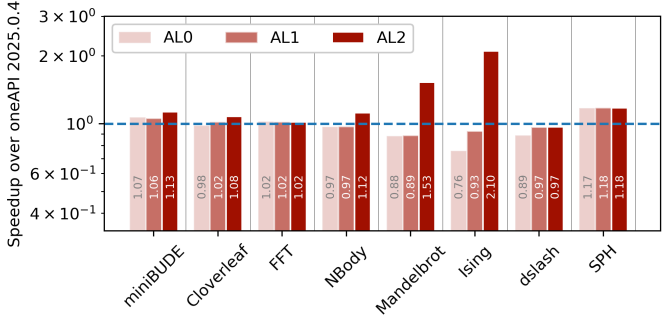
- Higher adaptivity levels generally deliver more performance
- Geometric mean of best AdaptiveCpp configuration: AdaptiveCpp outperforms CUDA by 30%

Performance on AMD



- Massive performance gain in some cases (up to 5.8×!)
- Geometric mean of best AdaptiveCpp configuration: AdaptiveCpp outperforms HIP by 44%

Performance on Intel



- RSBench failed to validate
- Mostly AL2 has an effect
- Geometric mean of best AdaptiveCpp configuration: AdaptiveCpp outperforms oneAPI by 23%

Impact of individual optimizations

Enable optimizations progressively

Number of enabled optimizations	Enabled JIT-time optimizations	Notes
0	–	=AL0
1	known group size	
2	All above + global size fits in int32	
3	All above + local memory specialization	
4	All above + alignment specialization	
5	All above + automatic noalias	=AL1
6	All above + automatic argument specialization	=AL2

Impact of individual optimizations

Change in # optimizations	RTX A5000	Radeon Pro VII	UHD 630
$0 \rightarrow 1$	1.00	1.39	1.02
$1 \rightarrow 2$	1.01	1.02	1.00
$2 \rightarrow 3$	1.04	1.00	1.00
$3 \rightarrow 4$	1.05	1.01	1.00
$4 \rightarrow 5$	1.04	1.10	1.02
$5 \rightarrow 6$	1.02	0.99	1.23

Figure: Geomean of speedup across all applications when enabling an additional optimization

- On NVIDIA, (amost) all optimizations contribute modestly
- On AMD, known-group-size is primarily responsible for the speedup
- On Intel, speedup is mostly due to automatic argument specialization
- $\rightarrow$ All optimizations are beneficial in at least **some** circumstance!

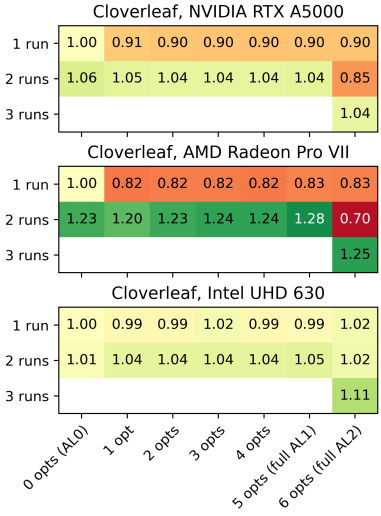
Performance behavior in detail

How quickly does performance converge?
What about kernel submission latency due to additional analysis?
What about JIT overhead?
What about individual applications?

CloverLeaf as proxy for worst-case:

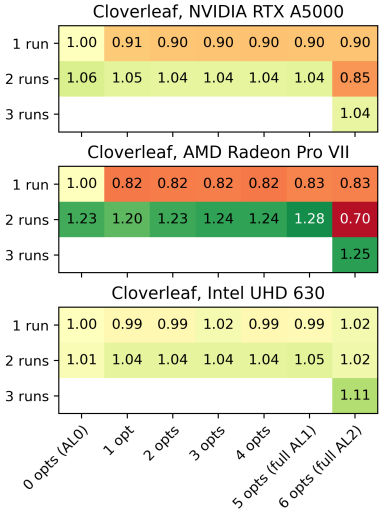
- Many short-running kernels
- Kernels are simple and (mostly) insensitive to our optimizations

How quickly does performance converge?



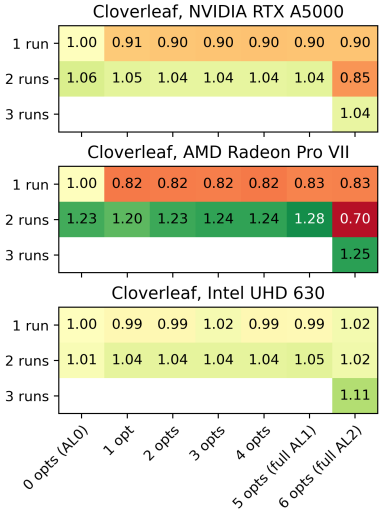
- Rerun applications until no JIT occurs anymore
- Shown: Speedup over first AL0 run (cell in top-left corner)
- AL1: Across all applications, no JIT anymore by second application run
  - AL1 optimizations, as expected, converge by the second kernel launch
- AL2: Across all applications, no JIT anymore by third application run
  - AL2 typically needs another application run

What about kernel launch latency?



- The additional analysis for each kernel launch could introduce additional latency.
- But: No noticable slowdown once JIT-cache is populated $\rightarrow$ runtime overheads are expected to not be signicant for typical apps.

What about JIT overhead?



- Compare first row to last row
- Performance drop in very first run is **not** due to more kernels being generated!
  - If so, performance would regress further as more optimizations are enabled
  - Instead: Runtime no longer compiles whole TU at a time
- $\rightarrow$ AL1 optimizations are sufficiently carefully selected such that additional JIT is generally not an issue!
- For AL2, more difficult to interpret - will the user always run the app with the same arguments?

Interesting behavior of individual applications

Summary of individual application behaviors

Here only summary, see paper for full discussion of individual application results!

- For some apps, e.g. miniBUDE on NVIDIA, we see ideal behavior:
  - Performance increases for more enabled optimizations and more application runs
- High speedup on AMD for FFT is due to known-group-size:
  - Compiler replaces group barrier with subgroup barrier for group size of 64.
- High speedup on AMD for RSBench is due to known-group-size:
  - Improved register allocation, substantially reducing register spilling (reduces data read/written to/from global memory by 11x!)
- For dslash, signficant speedup due to automatic-noalias on AMD/NVIDIA:
  - Instead of regular loads/stores compiler emits texture loads/stores which utilize the texture cache

Conclusion

- The new JIT-time optimizations deliver substantial performance improvements
  - 30% geometric mean speedup over CUDA, 44% over HIP, 23% over oneAPI
- For some applications, speedup can be much higher (e.g. >5x for RSBench)
- Especially at AL1, there are few downsides: Typically, the number of generated kernels does not increase → no noticable JIT overhead
- Also addresses some gaps/issues in the current SYCL standard (e.g. APIs returning 64-bit `size_t`, lack of noalias/restrict semantics, ...)
- **Already in production and stable!** If you've used any recent AdaptiveCpp release, you've used this framework and AL1.
- In the future, framework could be extended with autotuning, integration with LLVM PGO, ...

This project has received funding from the European Union's HE research and innovation programme under grant agreement No 101092877 (SYCLops project).

Known-group-size + Global-Size-Fits-In-Int32 Example

Let's consider the following, simple SYCL kernel:

```
1  const int wg_size = 128;
2  const int global_size = 16 * wg_size;
3  int *ptr = sycl::malloc_device<int>(global_size, queue);
4  sycl::nd_range<3> range{
5    {1, 1, global_size}, {1, 1, wg_size}};
6  sycl::queue{}.parallel_for(range,
7  [=](sycl::nd_item<3> id) {
8    int gid = id.get_global_linear_id();
9    ptr[gid] = gid;
10 }).wait();
```

Results when JIT-compiling for NVIDIA:

- At AL0: 34 instructions, lots of `cvt` instructions to widen special registers to 64 bit
- With known-group-size and global-size-fits-in-int32:
  - Reduced to 14 instructions, no `cvt` anymore, all calculations in 32 bit.

Without JIT-time optimizations (NVIDIA PTX)


```
ld.param.u64 %rd1, [__param_0];
mov.u32 %r1, %ctaid.y;
mov.u32 %r2, %ctaid.x;
mov.u32 %r3, %ntid.y;
cvt.u64.u32 %rd2, %r3;
mov.u32 %r4, %ntid.x;
cvt.u64.u32 %rd3, %r4;
cvt.u64.u32 %rd4, %r1;
cvt.u64.u32 %rd5, %r2;
mov.u32 %r5, %tid.x;
cvt.u64.u32 %rd6, %r5;
mov.u32 %r6, %tid.y;
cvt.u64.u32 %rd7, %r6;
mov.u32 %r7, %nctaid.x;
cvt.u64.u32 %rd8, %r7;
mov.u32 %r8, %nctaid.y;
cvt.u64.u32 %rd9, %r8;
mov.u32 %r9, %ntid.z;
mov.u32 %r10, %tid.z;
cvt.u64.u32 %rd10, %r10;
mov.u32 %r11, %ctaid.z;
mul.wide.u32 %rd11, %r11, %r9;
add.s64 %rd12, %rd11, %rd10;
mul.lo.s64 %rd13, %rd12, %rd9;
add.s64 %rd14, %rd13, %rd4;
mul.lo.s64 %rd15, %rd14, %rd2;
add.s64 %rd16, %rd15, %rd7;
mul.lo.s64 %rd17, %rd16, %rd8;
```

...

```
add.s64 %rd18, %rd17, %rd5;
mul.lo.s64 %rd19, %rd18, %rd3;
add.s64 %rd20, %rd19, %rd6;
cvt.s64.s32 %rd21, %rd20;
shl.b64 %rd22, %rd21, 2;
add.s64 %rd23, %rd1, %rd22;
st.global.u32 [%rd23], %rd20;
```

- Quite a lot of code for such a simple kernel!
- Notice the many `cvt` instructions to widen the work item indices, group indices,...to 64-bit!

Correctness of global-size-fits-in-int32

UNIVERSITÄT HEIDELBERG ZUKUNFT SEIT 1386

Global-size-fits-in-int32 does **not** negatively affect correctness!
Given the following C++ code:

```
1 uint64_t gid = idx.get_global_linear_id();
2 gid *= 1024;
```

the optimization does **NOT** generate an equivalent of the following, which would create an overflow hazard!

```
1 /*uint64_t*/ uint32_t gid = idx.get_global_linear_id<uint32_t>();
2 gid *= 1024;
```

but rather:

```
1 uint64_t ret_gid = idx.get_global_linear_id<uint32_t>();
2 __builtin_assume(ret_gid < UINT_MAX);
3 uint64_t gid = 1024 * ret_gid;
```

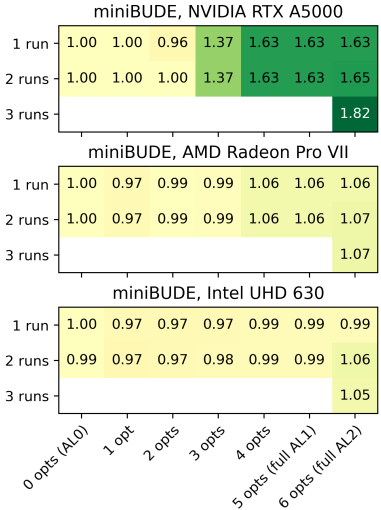
The data type formally remains 64-bit – but the internal global id calculations happen in 32-bit, and LLVM is made aware that the value is smaller than `UINT_MAX`.

With known-group-size and global-size-fits-in-int32

```
ld.param.u64 %rd1, [__param_0];
mov.u32 %r1, %ctaid.z;
mov.u32 %r2, %ctaid.y;
mov.u32 %r3, %ctaid.x;
mov.u32 %r4, %tid.x;
mov.u32 %r5, %nctaid.y;
mov.u32 %r6, %nctaid.x;
shl.b32 %r7, %r3, 7;
or.b32  %r8, %r7, %r4;
shl.b32 %r9, %r6, 7;
mad.lo.s32 %r10, %r1, %r5, %r2;
mad.lo.s32 %r11, %r9, %r10, %r8;
mul.wide.u32 %rd2, %r11, 4;
add.s64 %rd3, %rd1, %rd2;
st.global.u32 [%rd3], %r11;
```

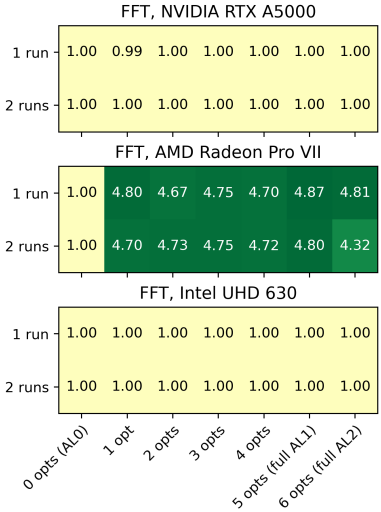
- 34 instructions reduced to 14
- No `cvt` anymore!
- Known group size: Note `shl %rX, 7` instead of multiplication by group size

miniBUDE



On NVIDIA: Aligns with theoretically expected ideal behavior: Faster for more optimizations and more runs

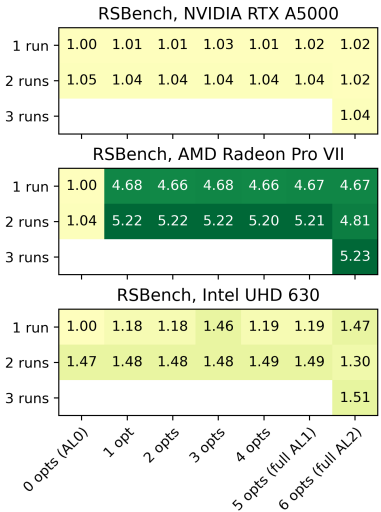
FFT



What happened on AMD?

- Massive speedup due to known-group-size
- Kernel relies on reordering data in local memory $\rightarrow$ lots of group barriers
- This application happens to use use group size 64, which matches subgroup size on this hardware
- Subgroup barriers are no-ops on this hardware $\rightarrow$ compiler omits all synchronization (`s_barrier` instruction)

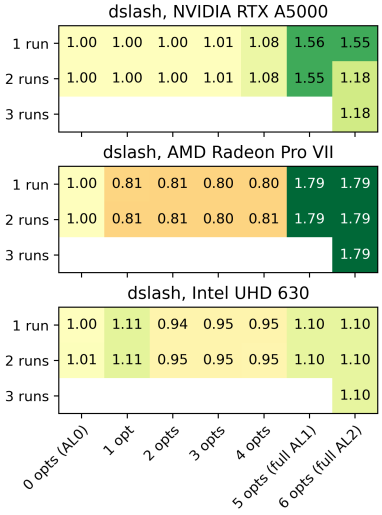
RSBench



What happened on AMD?

- Again 5x speedup due to known-group-size
- Kernel is large, register pressure and spilling play a significant role
- Knowing group size allows compiler to optimize register allocation
  - Does not need to pessimistically assume maximum group size
  - Allocate 128 instead of 64 vector registers!
- $\rightarrow$ Less register spilling - total global memory data read/write reduced by 11x (!)

dslash



- Reacts strongly to automatic-noalias
- Instead of regular global memory read, compiler uses texture memory (`ld.global.nc` on NVIDIA) which enables additional caching
- Why does perf decrease again on NVIDIA at AL2?
  - register usage drops from 84 to 80, occupancy increases from 30% to 44%
  - But: Texture cache hit rate drops from 56% to 47%
  - $\rightarrow$ More pressure on the cache due to more simultaneously active warps
  - AL2 still does its job