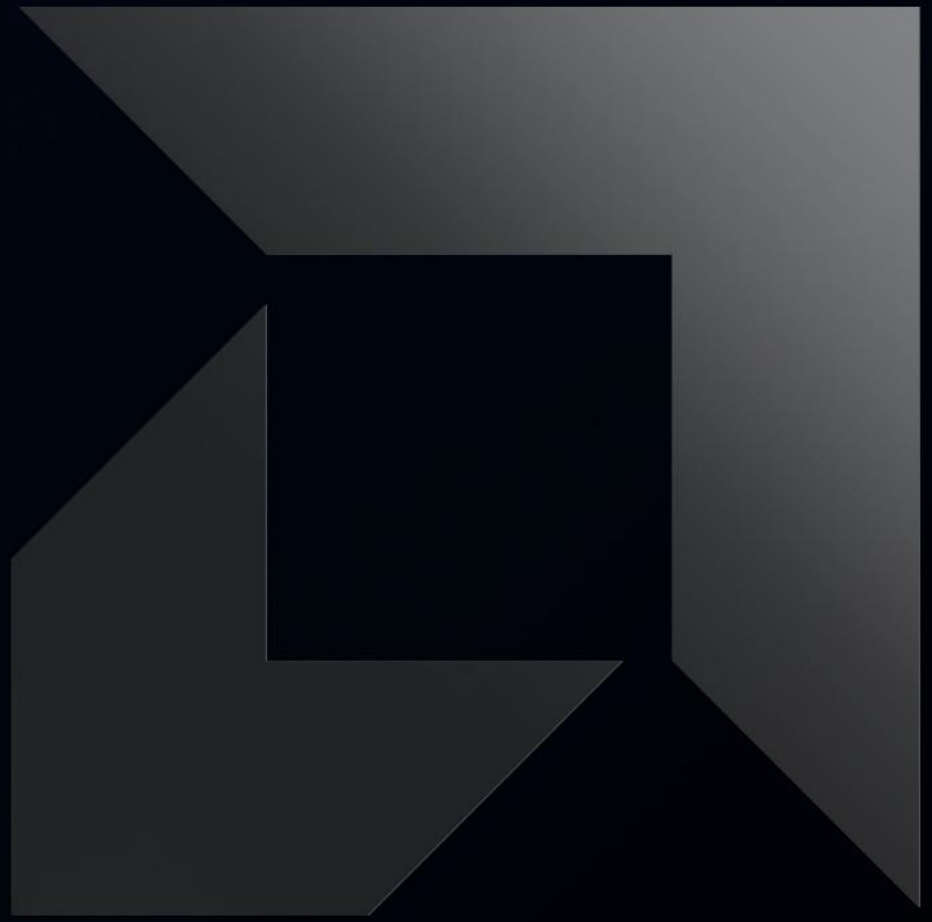


Accelerating PyTorch models with LLM augmented HIP kernels

Presenter: Giacomo Capodaglio
October 15th, 2025
AMD @ CASTIEL

AMD
together we advance_

Agenda

1. Background Information
2. Introduction
3. Motivation
4. Recipe for accelerating kernels
   - General LLM workflow approach
5. BitNet inference
6. Neck-and-neck performance with CUDA PTX BitNet kernels
7. Key takeaways

Generated by ChatGPT

Intro

- Main goal: accelerating PyTorch models with LLM-augmented HIP kernels

- We focus on accelerating Microsoft BitNet, a novel 1.58-bit quantized LLM
  - 20k GitHub stars and no current AMD GPU support

- Subjects
  - adding AMD support to PyTorch models
  - greatly enhancing developer productivity with state-of-the-art LLMs
  - setting up a workflow for easy kernel iteration with LLMs
  - leveraging PyTorch compile for optimized model execution
  - calling HIP kernels in PyTorch with custom C++ operator registration

- Our LLM-augmented HIP kernel and optimizations demonstrate significant performance gains
  - Over a 2x speedup over baseline PyTorch on RDNA-based GPUs
  - We also highlight the same kernel on an MI300x outperforming H100's using Microsoft's official CUDA+PTX inference kernels in peak tokens-per-second

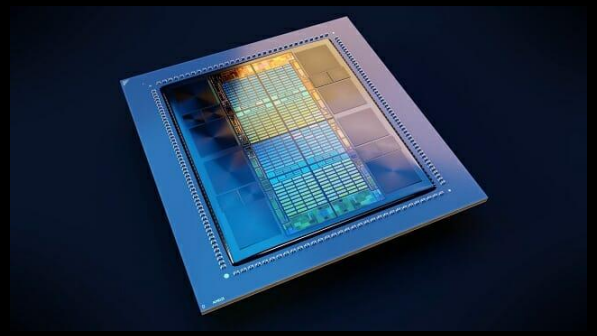
Software

Python 3.12.11
PyTorch 2.7.1
- Wheel respective to ROCm or CUDA

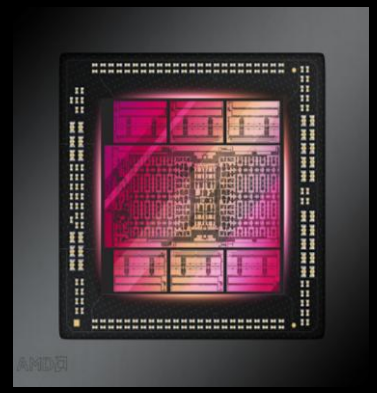
Pip versions
- Documented in /bitnet/pip_freeze.txt

Hardware

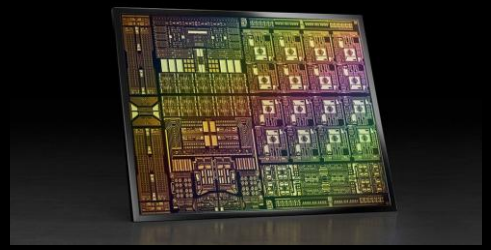
CDNA
- MI200, MI300



RDNA
- 6900XT, Navi3



CUDA
- A100, H100



AMD @ CASTIEL
Note: Images from AMD and NVIDIA

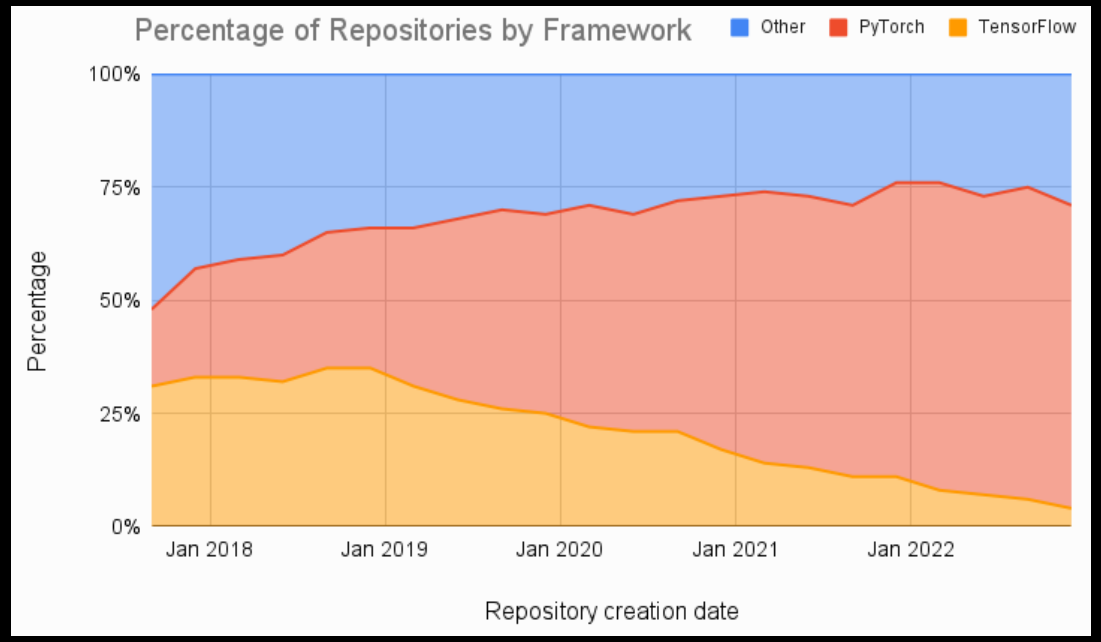
Background Information

PyTorch
- The most popular deep learning framework

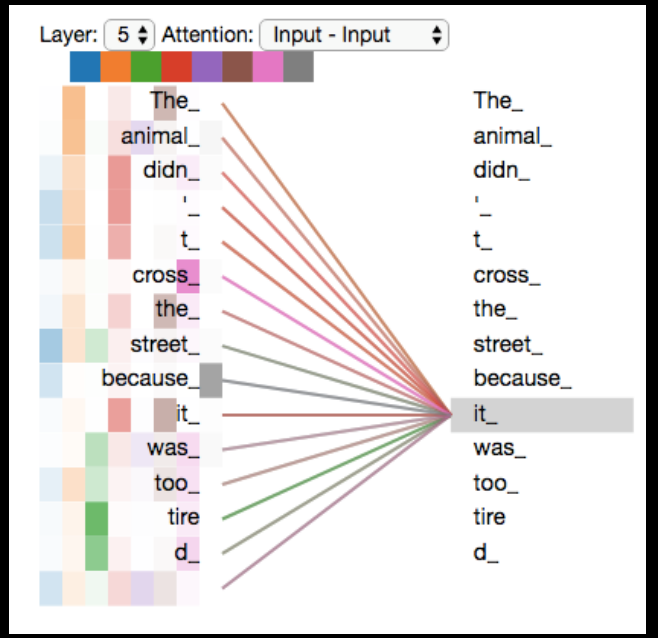
PyTorch 2.x
- Introduced torch compile, a JIT Compiler to optimized kernels
- https://pytorch.org/get-started/pytorch-2-x/

Self-Attention
- Allows each token to determine how much to weight other tokens

FlashAttention
- Fast and Memory-Efficient Exact Attention with IO-Awareness



PyTorch usage over time



Self-Attention

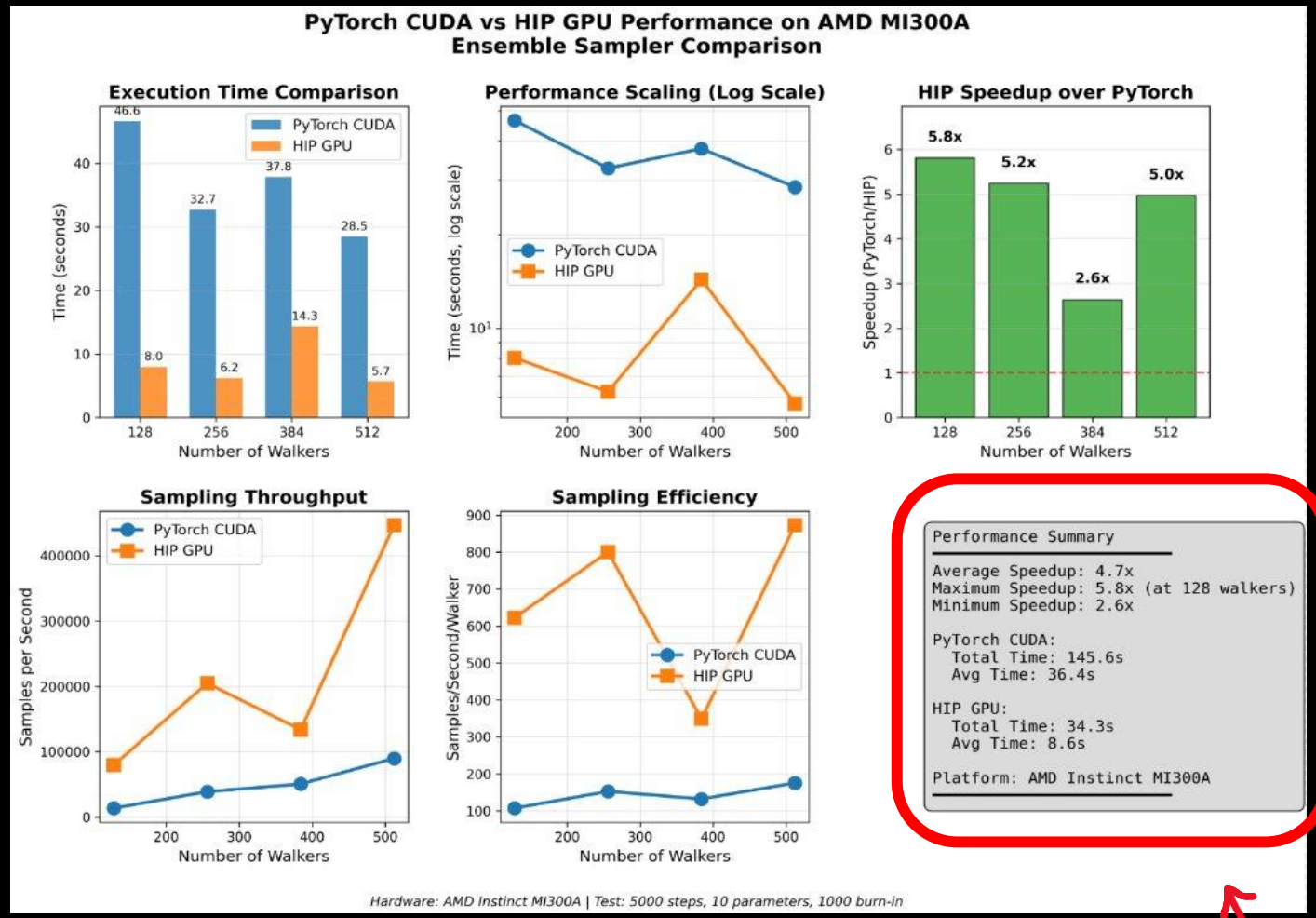
AMD @ CASTIEL

Note:Images from Medium and Jay Alamar
FlashAttention: https://arxiv.org/abs/2205.14135

Inspiration

- CambridgeIOA @ AMD Hackathon at EPCC in Edinburgh UK
  - "Parallelized affine invariant MCMC sampling"
- Converted PyTorch code to HIP using Claude LLM
- Resulted in a 5.8x speedup
- Speedup comes from HIP generating fewer kernels than PyTorch
  - PyTorch: Total kernels: 29779
  - HIP: Total kernels: 8978

Does this extend to deep learning models?



Cambridge's Results
NOTE: PyTorch CUDA means pure PyTorch in the plots

How we used LLMs

Which LLMs have been used

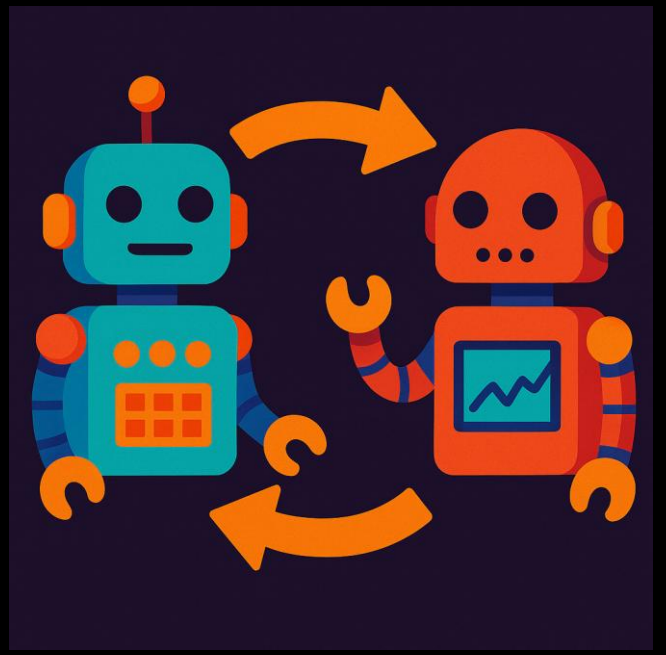
- Proprietary
  - ChatGPT
    - 03, o4-mini, o4-mini-high
  - Gemini
    - 2.5 Flash, 2.5 Pro
- Open Source
  - DeepSeek-R1 (671B params)
  - Qwen3-Coder (405B params)
- We found best results from reasoning models Gemini 2.5 Pro and ChatGPT o3
  - Qwen3-Coder is a great free alternative
- Models with more parameters are, in general, more capable
  - Due to scaling laws: https://arxiv.org/abs/2001.08361
- Use whichever model is the most powerful
  - Find the best LLMs at https://www.vellum.ai/llm-leaderboard/

Differences between LLMs

- ChatGPT
  - Great for generating ideas for optimization or areas to debug
  - Doesn't like to paste entire output
  - In our experience, Canvas mode is worse than just outputting in the chat
  - Will completely change the style of your code
- Gemini
  - Very consistent, strong coder
  - Great for implementing the code you tell it to write
  - Will not diverge from your coding style
  - Not lazy, will paste entire files
- Open-source models
  - In our experience they are less capable at coding
  - Very hard to out compete large (1T+ param) proprietary models
  - Extremely cheap and fast for simple tasks
  - Qwen3-Coder can compete with proprietary models

Switching between models

- Sometimes we found that switching between ChatGPT and Gemini chats could unlock the benefits of both models
- Models can get stuck in a repetitive loop
  - Having fresh context can help
- Example
  - Generate optimization ideas with ChatGPT
  - Decide which one to implement
  - Prompt Gemini to start that optimization



Generated by ChatGPT

How to use and find different models

- Where to find models
  - Gemini: gemini.google.com
  - ChatGPT: chatgpt.com
  - Qwen: chat.qwen.ai
- Open Source
  - HuggingFace
  - Ollama
  - LM Studio

Hugging Face

Gemini

Qwen

ChatGPT

LM Studio

Ollama

Note: logos from company front pages

Prompting

- We have used very simple prompting techniques
  - "optimize this kernel for ___ "
  - "please fix this error"
- However, you must be very specific in detailing the changes you want made
  - Iterative, small changes work better than large refactors
- Context lengths are becoming quite large, take advantage of them
  - Paste entire documentation and source code files into your chat
  - `rocminfo` output, console error logs
  - The more the better
- For fast iteration time, tell it to print the entire function or source file
  - Make sure to verify changes
  - Side-by-side git diffs are extremely useful for this type of work
- One-shot solutions rarely work. Usually within 10 prompts you can resolve an error
  - If taking longer than this, retry in a new tab

LLMs + GPU kernels

Why LLM inference (and not training)

- Low(er) hardware requirements
  - Don’t need to store optimizers or the backward pass in RAM, resulting in a much smaller memory footprint
- Easily ensure accuracy with LLMs – does the output make sense?
- With proper optimization, we can run LLMs on a Radeon GPU, and possibly on a mini-pc or consumer laptop
- Kernels are usually short (1000 lines of code)
  - Perfect for pasting into your LLM
- End result: chat with LLM on your local machine!

Note: image from Reddit

Recipe for accelerating GPU kernels

Ideally this process can be largely automated

General LLM workflow

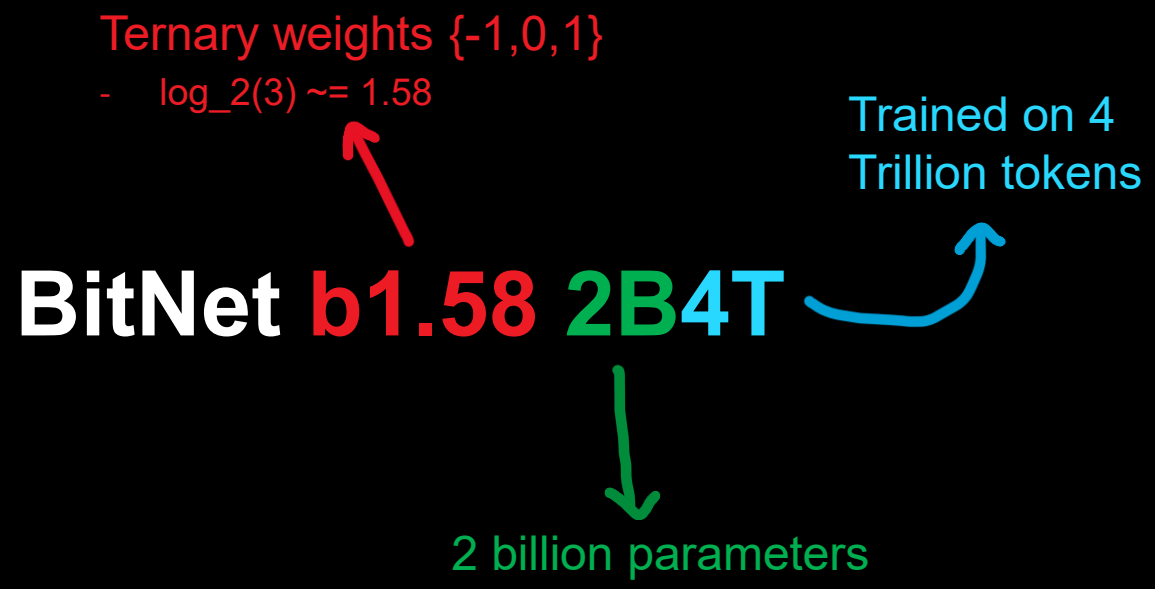
1. Set up boilerplate and minimum-viable working solution
2. Ask LLM to optimize
3. Compile and run file
4. Ensure correct output.
    - Does this match the correct, unoptimized output?
5. If error, feed back to LLM
6. Save working kernel
7. Generate `rocprof` stats
8. Feed LLM context the `rocprof` stats and working kernel
9. Iterate!

The key is properly guiding the LLM to success

- Useful context
- Expertise in knowing what to optimize

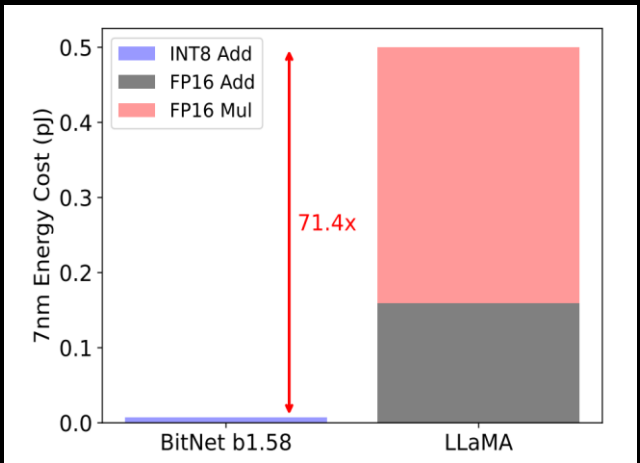
Use case: BitNet Inference

BitNet b1.58 2B4T



What is BitNet?

- TLDR; "a super efficient LLaMa-style transformer"
  - Replaces LLaMa linear layer
  - 2B parameters, trained on 4T tokens
- Very efficient due to "1.58 bit" weights
  - {-1,0,1} encoded as INT2 instead of FP16/FP32
  - log_2(3) ~= 1.58
- Optimized inference code only has CUDA support
  - CUDA/PTX kernel replaces torch.nn.linear
- Open-source LLM developed by Microsoft
  - 20k stars on GitHub
  - No AMD support!
  - Paper: https://arxiv.org/abs/2402.17764



BitNet energy efficiency compared to LLaMa

```cpp
template <typename T1, typename T2>
__device__ void decode_i2s_to_i8s(T1 *_i2s, T2 *_i8s, const int N = 16)
{
  // convert 8 int2b_t to 8 int8b_t -> 2 int32
  uint *i8s = reinterpret_cast<uint *>(_i8s);

  // i2s = {e0, e4, e8, e12, e1, e5, e9, e13, e2, e6, e10, e14, e3, e7, e11, e15}
  uint const i2s = *_i2s;

  static constexpr uint immLut = (0xf0 & 0xcc) | 0xaa;     // 0b11101010
  static constexpr uint BOTTOM_MASK = 0x03030303;          // 0xf -> 0b11 select 0,3
  static constexpr uint I4s_TO_I8s_MAGIC_NUM = 0x00000000;

#pragma unroll
  for (int i = 0; i < (N / 4); i++)
  {
    asm volatile("lop3.b32 %0, %1, %2, %3, %4;\n"
                 : "=r"(i8s[i])
                 : "r"(i2s >> (2 * i)), "n"(BOTTOM_MASK), "n"(I4s_TO_I8s_MAGIC_NUM), "n"(immLut));
    i8s[i] = __vsubss4(i8s[i], 0x02020202);
  }
}
```

BitNet kernel containing PTX

Adding AMD support

- Enhanced official code
  - Fixed eot_id bug, CUDA compile bug, added `--pytorch` flag
- Set up HIP kernel bindings
- Resolved Meta Xformers library incompatibility
  - No support for flash attention (flash.FwOp) backend on ROCm
  - ROCm installation provides composable kernel operator at the moment

```
output = fmha.memory_efficient_attention_forward(
    xq, cache_k, cache_v, attn_bias, op = fmha.flash.FwOp
)
```

NVIDIA version, fast

```
output = fmha.memory_efficient_attention_forward(
    xq, cache_k, cache_v, attn_bias, op = fmha.ck.FwOp
)
```

ROCm version, slower

BitNet Linear Kernel

- w2a8 GEMV
  - 2-bit weights × 8-bit activations
- Super optimized linear transformation
  - Sixteen packed INT2 weights into a single INT32
  - INT2 weights in 16×32 blocks with interleaved swizzle pattern for coalesced memory accesses
- Replaces torch.nn.Linear

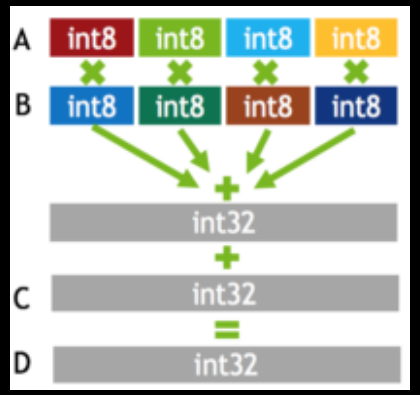
How the w2a8 GEMV works:

1. Decode swizzled data to int8

```
[0, 4, 8, 12, 1, 5, 9, 13, 2, 6, 10, 14, 3, 7, 11, 15]
```

Special interleaving pattern for packed INT2 weights

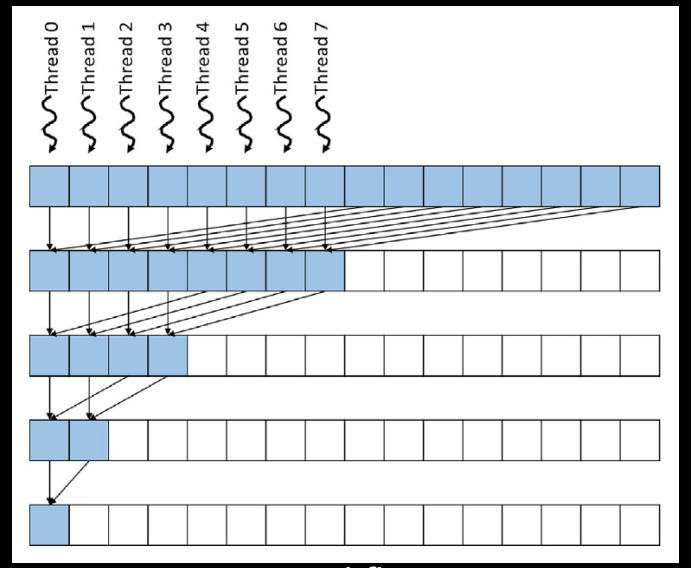
2. Multiple sdot4



dp4a/sdot4

Note: diagram from https://developer.nvidia.com/blog/mixed-precision-programming-cuda-8/

3. Shuffle down accumulate



__shfl

Note: diagram from Science Direct

PyTorch bindings to HIP kernel

1. Compile kernel as shared library and call with ctypes
   - Little to no set up
   - Quick for experimentation
2. Custom C++ PyTorch operator
   - Fast, allows for greater fusion
     - Dynamo/AOTAutograd can reason about shapes/dtypes for the custom HIP operator without tracing into it
   - We saw 5%-7% end-to-end speedups over method #1
   - Takes time to set up

#1. Compile kernel as shared library and call with ctypes

1. Create a dispatch function in .hip

```
extern "C" void bitlinear_int8xint2(int8_t* input0, int8_t* input1, hip_bfloat16* output0, hip_bfloat16* s, hip_bfloat16* ws, int M, int N, int K, hipStream_t stream){
```

2. Define your kernel in .h

```
template <int M, int N, int K, int ws_num, int K_block_size, int N_block_size>
__global__ void __launch_bounds__(128) ladder_int8xint2_kernel(int8_t* __restrict__ A,
```

3. Compile as a shared library

```
bitnet > bitnet_kernels_hip > $ compile.sh
1  hipcc -std=c++17 -fPIC --shared bitnet_kernels.cpp -o libbitnet.so
```

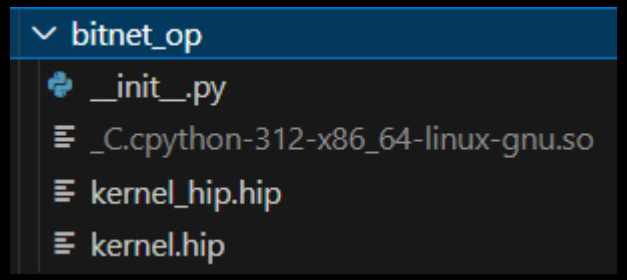
4. Link shared library to PyTorch with ctypes

```
import ctypes
bitnet_lib = ctypes.CDLL('bitnet_kernels_hip/libbitnet.so')

def bitnet_int8xint2_linear(input0, input1, s, ws, amd=False):
    out_shape = list(input0.shape)
    out_shape[-1] = input1.shape[0]

    stream = torch.cuda.current_stream()

    M = input0.shape[0]
    if len(out_shape) == 3:
        M *= input0.shape[1]
    N = input1.shape[0]
    K = input1.shape[1] * 4

    ret = torch.zeros(*out_shape, dtype=torch.bfloat16, device=input0.device)

    bitnet_lib.bitlinear_int8xint2(*[ctypes.c_void_p(input0.data_ptr()),
                                     ctypes.c_void_p(input1.data_ptr()),
                                     ctypes.c_void_p(ret.data_ptr()),
                                     ctypes.c_void_p(s.data_ptr()),
                                     ctypes.c_void_p(ws.data_ptr()),
                                     ctypes.c_int(M),
                                     ctypes.c_int(N),
                                     ctypes.c_int(K),
                                     ctypes.c_void_p(stream.cuda_stream)])

    return ret
```

#2. Custom C++ PyTorch operator

1. Create directory to hold operator
   - /bitnet_op
2. Create .hip file with PyTorch bindings
3. Create setup.py
4. uv pip install -e . --no-build-isolation
   - export PYTORCH_ROCM_ARCH=gfx942
5. Move .so library from /build to /bitnet_op
6. Import custom op folder into PyTorch code

Custom op folder



setup.py

```python
from setuptools import setup
from torch.utils.cpp_extension import BuildExtension, CUDAExtension

# change bitnet_op to bitnet_v1 for rdna
setup(
    name='bitnet_op',
    version='1.0',
    packages=['bitnet_op'],
    ext_modules=[
        CUDAExtension('bitnet_op._C', ['bitnet_op/kernel.hip'],)
    ],
    cmdclass={
        'build_ext': BuildExtension
    }
)
```

PyTorch tutorial: https://docs.pytorch.org/tutorials/advanced/cpp_custom_ops.html

Register with PyTorch inside kernel.hip

```
TORCH_LIBRARY(bitnet_op, m) {
  m.def("bitlinear(Tensor input0, Tensor input1, Tensor s, Tensor ws) -> Tensor");
}

TORCH_LIBRARY_IMPL(bitnet_op, CUDA, m) {
  m.impl("bitlinear", &bitlinear_pytorch);
}
```

BitNet runs locally!

- Run the full 2B BitNet on your local ROCm machine
  - Need >= 8GB GPU ram
- Github Repository
  - Currently internal
  - github.com/AMD-HPC/bitnet/
- RDNA and CDNA support

github.com/AMD-HPC/bitnet/blob/main/setup.sh

Entire setup

```
uv venv --python 3.12 # uv venv because built-in venv runs out of memory when installing rocm torch with pip
source .venv/bin/activate

# install dependencies
uv pip install torch torchvision torchaudio xformers --index-url https://download.pytorch.org/whl/rocm6.3
uv pip install fire sentencepiece tiktoken blobfile flask einops transformers
uv pip install -U xformers --index-url https://download.pytorch.org/whl/rocm6.3

cd src/hip/
bash compile.sh # if on rdna, comment the first line and uncomment the second
cd ..

# download and convert the BitNet-b1.58-2B model
mkdir checkpoints
huggingface-cli download microsoft/bitnet-b1.58-2B-4T-bf16 --local-dir ./checkpoints/bitnet-b1.58-2B-4T-bf16
python3 ./convert_safetensors.py --safetensors_file ./checkpoints/bitnet-b1.58-2B-4T-bf16/model.safetensors --output checkpoints/model_state.pt --model_name 2B
python3 ./convert_checkpoint.py --input ./checkpoints/model_state.pt
rm ./checkpoints/model_state.pt
```

Chat with model!

```
(bitnet-gpu) [kebuhler@TheraC10 gpu]$ python3 generate.py ./checkpoints/ --interactive --chat_
format
loaded model in 2.97 seconds
compiled model in 2.61 seconds
enter prompt: what is AMD
[[128000, 1502, 25, 220, 12840, 374, 25300, 128009, 72803, 25, 220]]
11 32
> what is AMD
AMD stands for Advanced Micro Devices. It is a multinational technology company that designs,
manufactures, and markets microprocessors, graphics processing units (GPUs), and other
---------------
```

Flags

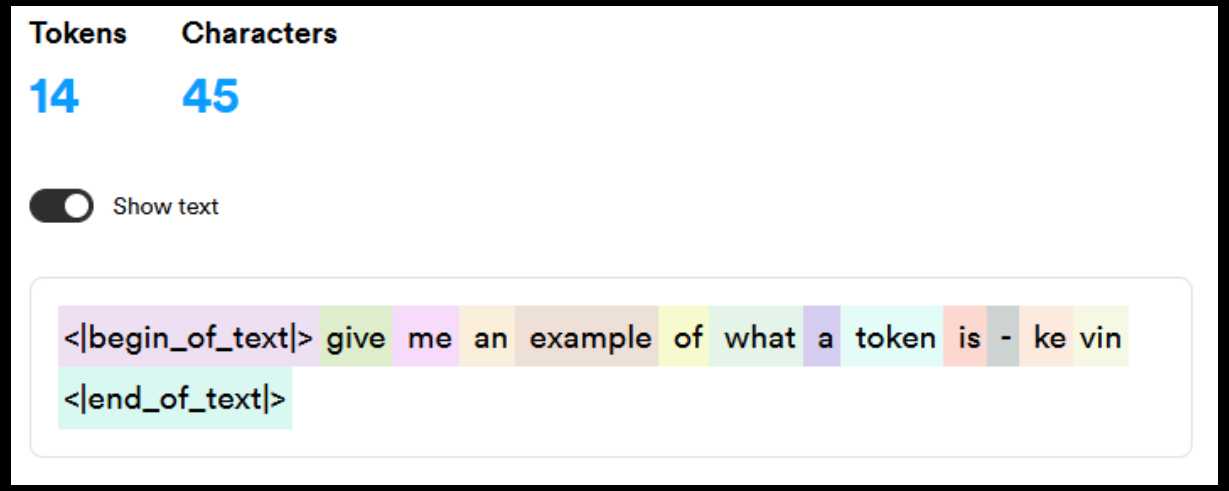
```
--interactive
--chat_format
--pytorch
```

Matching CUDA/PTX performance

BitNet inference performance

Measured in peak tokens per second (tok/s)

- Human reading speed: 5-7 tok/s
- Prompt: “what is an eigenvalue”
- 128 output tokens
- Measured on MI300x



Llama 3 tokenizer example

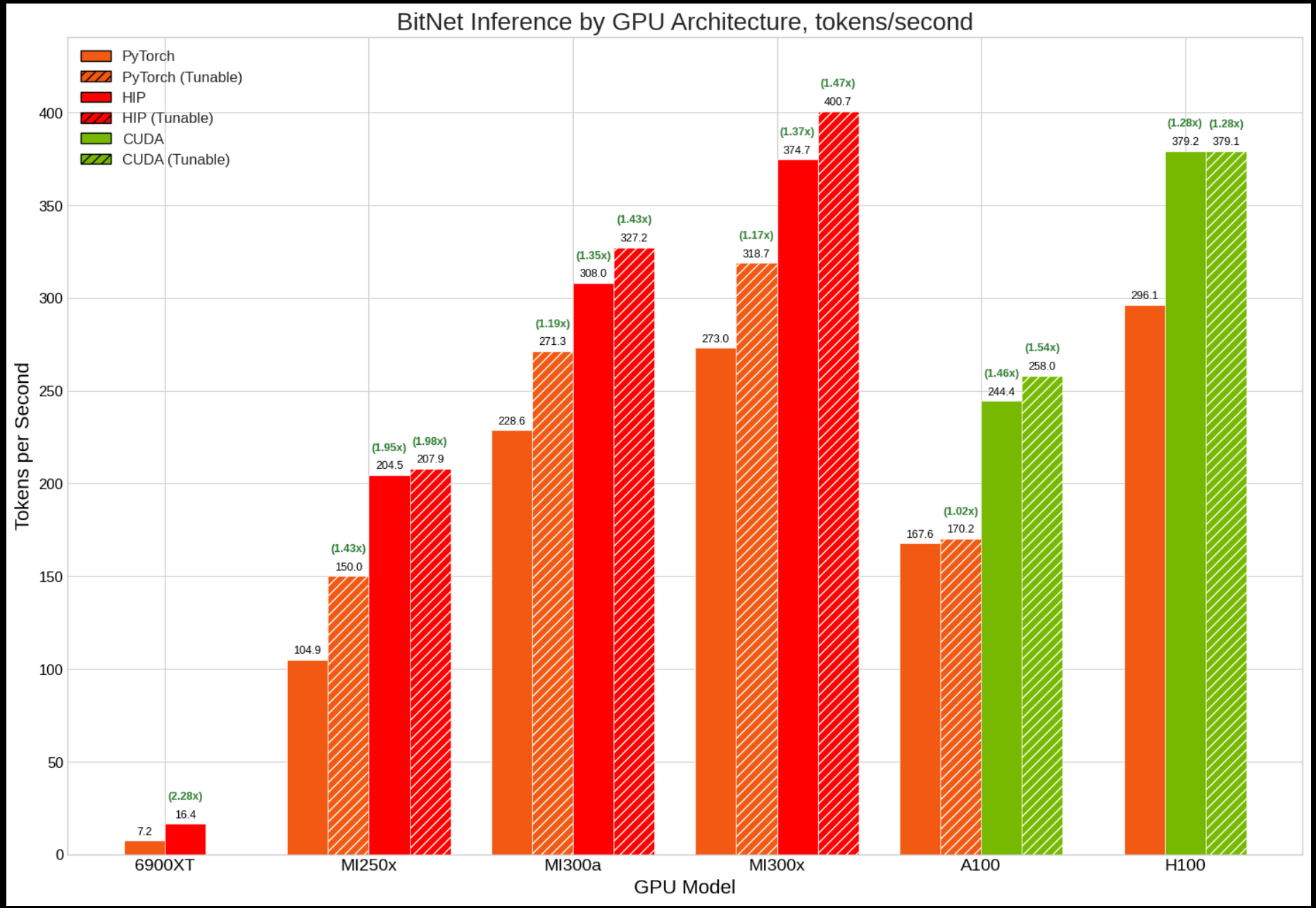
Three different types of models to test

- PyTorch
  - AMD and NVIDIA hardware
- PyTorch + CUDA/PTX w2a8 kernel
  - NVIDIA hardware
- PyTorch + HIP w2a8 kernel
  - AMD hardware

Note: logos from company front pages

Consistent speedups

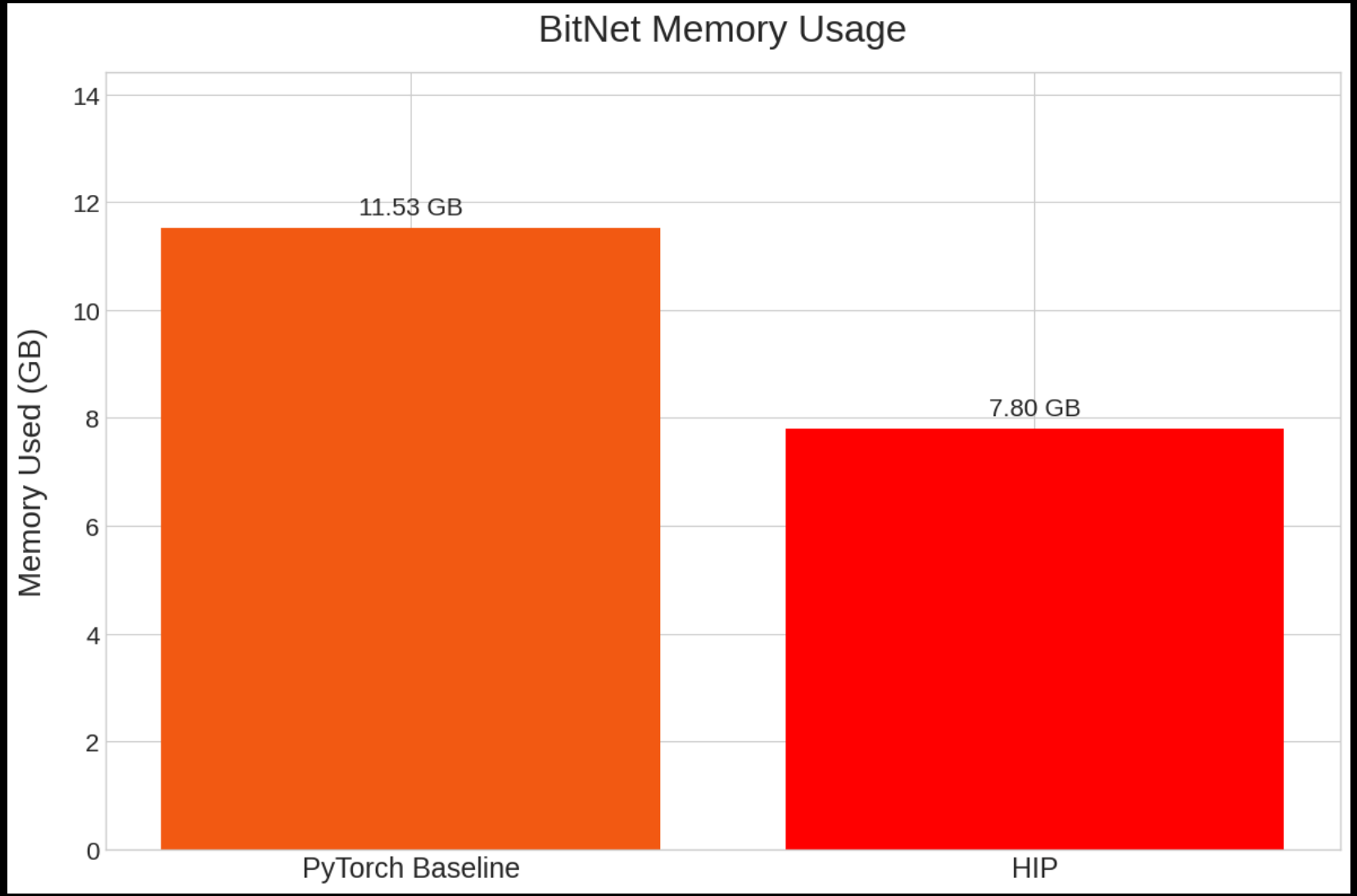
- End-to-end speedups over PyTorch
  - MI300x: ***1.47x***
  - MI300a: ***1.43x***
  - MI250x: ***1.98x***
  - 6900XT: ***2.28x***
- Trend
  - Larger speedups for smaller GPUs
- MI300x faster than H100!



Memory Differences

Memory dropped from 11.43 GB to 7.64GB

- ~33% decrease

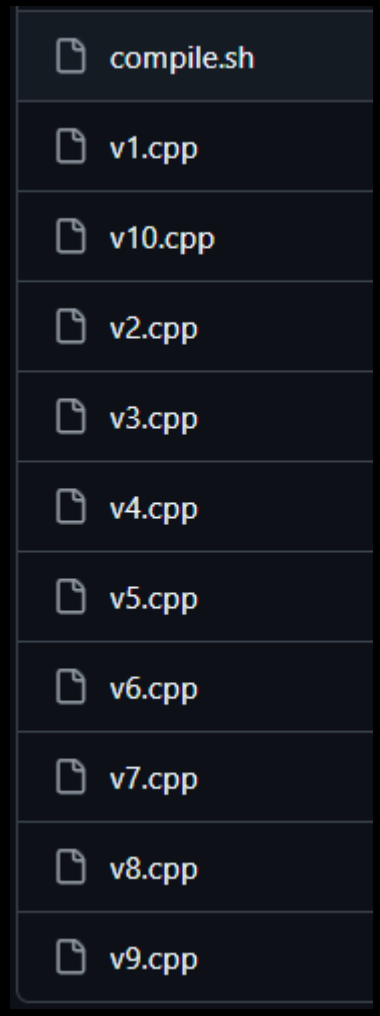


BitNet w2a8 Kernel

The main driver of our speedups

BitNet w2a8 Kernel

- In total, we created 10 different versions of this kernel
- We will be focusing on three kernels:
  - V1: Converted from CUDA to pure C++ HIP
  - V2: AMD intrinsic builtin
  - V10: Unified RDNA/CDNA kernel
- Code soon to make public (now in private repo)
  - https://github.com/AMD-HPC/bitnet/tree/main/src/hip



RDNA support for w2a8

- As a reminder, AMD provides two different ISAs
  - CDNA (Compute DNA), used in datacenter GPUs
  - RDNA (Radeon DNA), used in consumer GPUs
- Adding RDNA support requires two changes
- Change #1
  - Problem: Xformers CK backend for memory efficient attention is not supported on RDNA
  - Solution: Rewrite the Attention mechanism from scratch in pure PyTorch
    - Results in two files: rdna_model.py and rdna_generate.py
- Change #2
  - Problem: __builtin_amdgcn_sdot4 is only available on CDNA
  - Solution: A unified RDNA/CDNA kernel. Next slide...

Unified RDNA/CDNA kernel

- __builtin_amdgcn_sdot4 intrinsic isn't portable
  - LLVM target must be 'mai-insts'
- Assembly dump
  - Compiles to V_DOT4C_I32_I8 instructions
    - Not available in RDNA ISA
- However, RDNA has V_DOT4_I32_IU8
  - An alias for V_DOT4_I32_I8
  - Or, V_DOT4C_I32_I8 without an accumulator
- Result: same kernels works on both RDNA/CDNA!

CDNA

V_DOT4C_I32_I8 57

Compute the dot product of two packed 4-D signed 8-bit integer inputs in the signed 32-bit integer domain and accumulate with the signed 32-bit integer value in the destination register.

```
tmp = D0.i32;
tmp += i8_to_i32(S0[7 : 0].i8) * i8_to_i32(S1[7 : 0].i8);
tmp += i8_to_i32(S0[15 : 8].i8) * i8_to_i32(S1[15 : 8].i8);
tmp += i8_to_i32(S0[23 : 16].i8) * i8_to_i32(S1[23 : 16].i8);
tmp += i8_to_i32(S0[31 : 24].i8) * i8_to_i32(S1[31 : 24].i8);
D0.i32 = tmp
```

RDNA

V_DOT4_I32_IU8 22

Dot product of signed or unsigned bytes.

```
declare A : 32'I[4];
declare B : 32'I[4];
// Figure out whether inputs are signed/unsigned.
for i in 0 : 3 do
    A8 = S0[i * 8 + 7 : i * 8];
    B8 = S1[i * 8 + 7 : i * 8];
    A[i] = NEG[0].u1 ? 32'I(signext(A8.i8)) : 32'I(32'U(A8.u8));
    B[i] = NEG[1].u1 ? 32'I(signext(B8.i8)) : 32'I(32'U(B8.u8))
endfor;
C = S2.i;
// Signed multiplier/adder. Extend unsigned inputs with leading 0.
D0.i = A[0] * B[0];
D0.i += A[1] * B[1];
D0.i += A[2] * B[2];
D0.i += A[3] * B[3];
D0.i += C
```

```
asm volatile("V_DOT4_I32_I8 %0, %1, %2, %3"
             : "=v"(acc)        // %0: Dst
             : "v"(packed_A),   // %1: Src0
             "v"(packed_W),    // %2: Src1
             "v"(acc));        // %3: Src2 (accumulator)
```

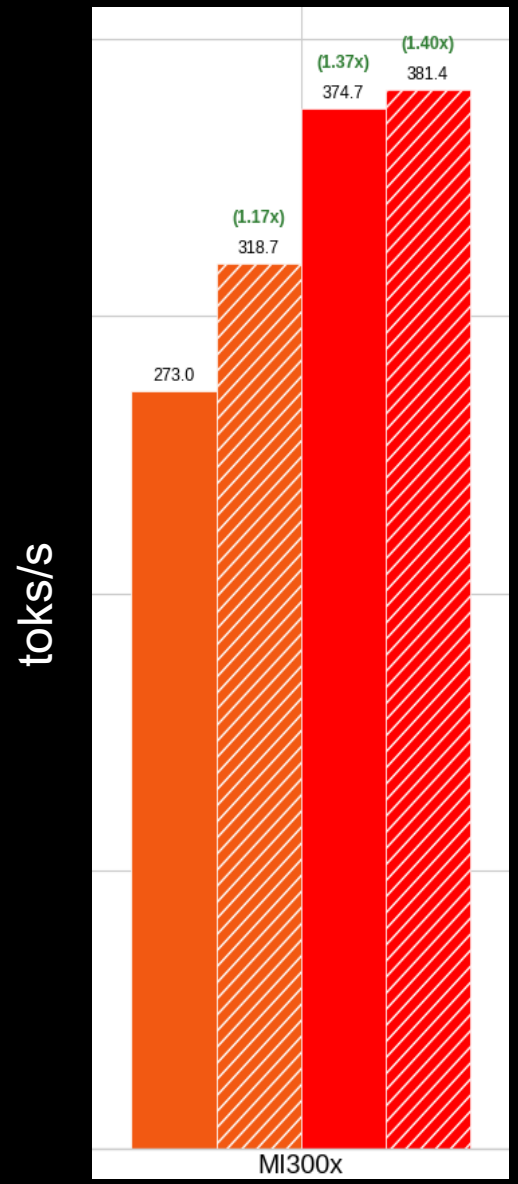
Unified kernel inline assembly

https://github.com/AMD-HPC/bitnet/blob/main/src/hip/v10.cpp

Extra Performance: PyTorch Tunable Op

- Environment flag to squeeze extra juice out of AMD GPUs
- Profiles different versions of our GEMMs and then runs the fastest
- We see noticeable boosts in runtime speedups when enabling
- Some errors when using on RDNA
- More information
  - https://docs.pytorch.org/docs/stable/cuda.tunable.html
  - https://rocm.blogs.amd.com/artificial-intelligence/pytorch-tunableop/README.html

Striped bars mean Tunable Op Enabled



v0
PyTorch

v1
PTX -> pure C++

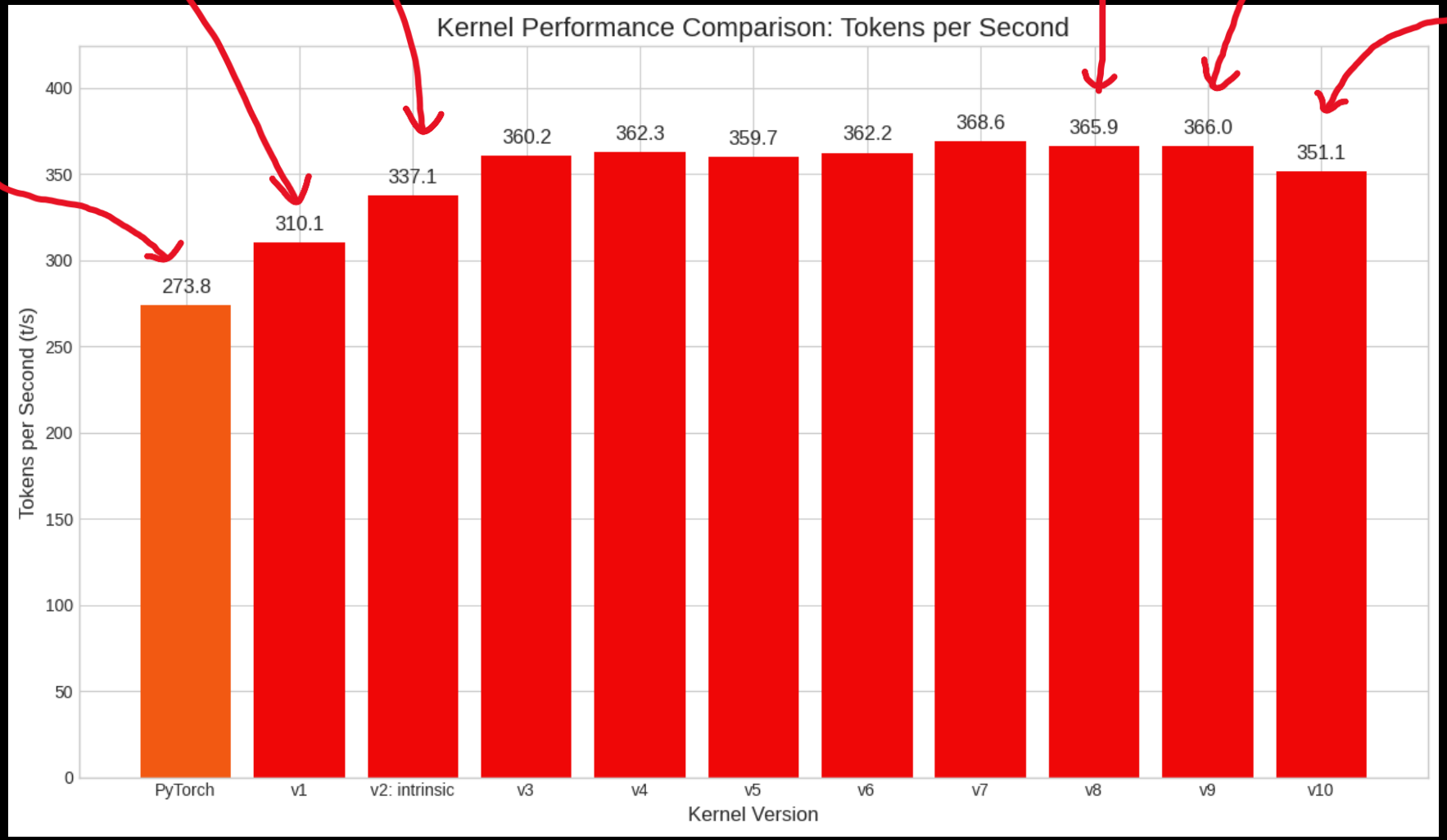
V2
__builtin_amdgcn_sdot4

v3
Removed
launch bounds

V8
Final sdot

V9
4x4x4 MFMA

V10
Unified Kernel

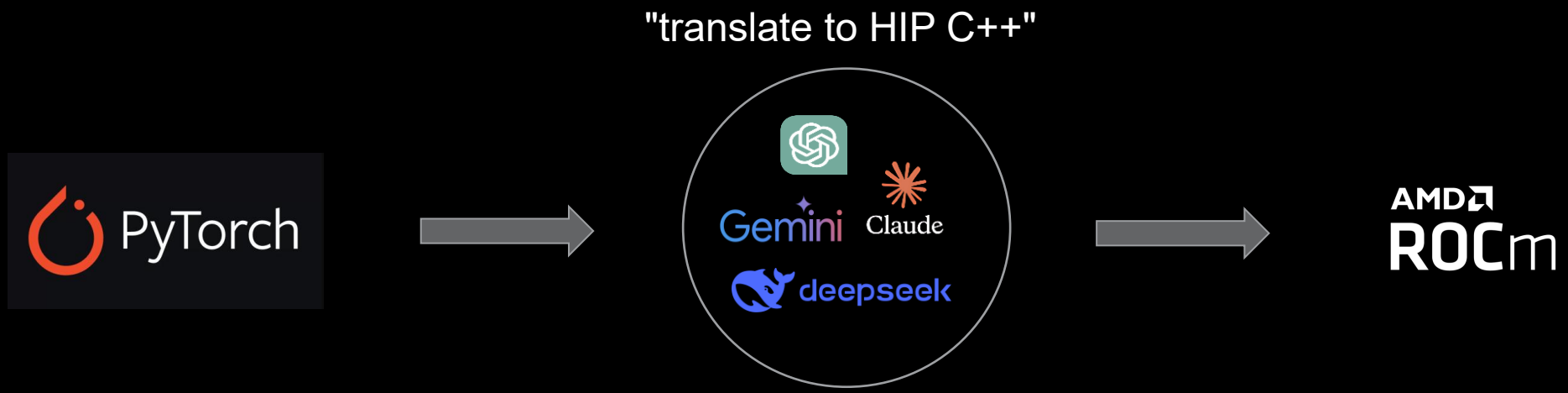


When controlling for just the kernel used, V10 Unified kernel performs worse than V8/V9?

- However, it empirically has the best performance with fusion and Tunable Op
- Works on both RDNA/CDNA

v1 - Porting CUDA to HIP C++

- LLMs are great at translating and optimizing pre-existing kernels
- LLM was able to easily translate CUDA kernel with PTX instructions to a naïve C++ HIP kernel
  - Worked better than HIPIFY-perl and HIPIFY-clang, which gave a myriad of errors
  - ~2x slower than CUDA version, similar performance to PyTorch



v2 - __builtin_amdgcn_sdot4

- Intrinsics and AMD ISA assembly are great places to find speedups
- This code block can be replaced by one intrinsic or assembly call
- Where to find intrinsics
  - https://github.com/llvm/llvm-project/blob/main/clang/include/clang/Basic/BuiltinsAMDGPU.def

Intrinsic replaces entire function

```
__device__ inline int dp4a_hip(int a, int b, int c) {
    int result = c;
    result += static_cast<int>(reinterpret_cast<const signed char*>(&a)[0]) * static_cast<int>(reinterpret_cast<const signed char*>(&b)[0]);
    result += static_cast<int>(reinterpret_cast<const signed char*>(&a)[1]) * static_cast<int>(reinterpret_cast<const signed char*>(&b)[1]);
    result += static_cast<int>(reinterpret_cast<const signed char*>(&a)[2]) * static_cast<int>(reinterpret_cast<const signed char*>(&b)[2]);
    result += static_cast<int>(reinterpret_cast<const signed char*>(&a)[3]) * static_cast<int>(reinterpret_cast<const signed char*>(&b)[3]);
    return result;
}
```

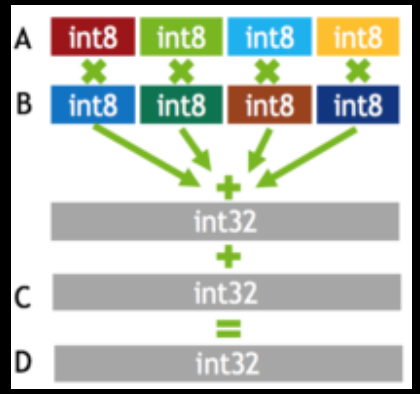
```
#pragma unroll
for (int k2 = 0; k2 < 4; ++k2) acc = __builtin_amdgcn_sdot4(*(int*)(&A_loc[k2 * 4]), *(int*)(&W_loc[k2 * 4]), acc, 0);
```

v2 - __builtin_amdgcn_sdot4

- We saw that we can replace a big chunk of code with one intrinsic, __builtin_amdgcn_sdot4
- What does this even do?
- Four INT8 dot products into a single INT32

Step 2 of our w2a8 GEMV kernel

2. Multiple sdot4



dp4a/sdot4

ic]

Kernel summary

- 10 different versions
  - v1: LLM naively translated PTX/CUDA to HIP C++
  - v2: sdot4 intrinsic
  - v10: Unified RDNA/CDNA kernel
- LLM
  - ChatGPT o3 to generate optimization ideas, Gemini 2.5 Pro to help implement them
- Changes from v1 to v10
  - Aggregate runtime decreased from 30.85% to ***22.44%***
  - Achieved Roofline utilization increased from 29.6% to ***74.5%***
  - Wall duration of w2a8 kernels: 784.505 / 204.235 ms = ***3.84x*** speedup
    - Generated with `rocprof --hip-trace python3 generate.py checkpoints` on an MI250x
- Final kernel
  - https://github.com/AMD-HPC/bitnet/blob/main/src/hip/v10.cpp

Summary: Matching CUDA performance

- LLM and I were able to get neck-and-neck CUDA performance with HIP and AMD ISA with ~2 weeks of pure work
  - Decode special swizzle pattern
  - Unpack INT2 weights into INT8 at runtime
  - Unrolling loops
  - Removing launch bounds
  - AMDGPU LLVM intrinsics
    - __builtin_amdgcn_sdot4 isn't officially documented but LLM found it?
    - __frcp_rn, __shfl_xor
  - AMD ISA instructions
- Use PYTORCH_TUNABLEOP_ENABLED=1 for max performance
- Compiler flags seem to have a small impact
  - compiled with `hipcc -O3 -fPIC --shared -o libbitnet.so v10.cpp`
- Register your kernel with PyTorch to allow more kernel fusion
  - 5-10% speedup
- Intrinsics/AMD ISA provide the majority of the speedup
  - Try to take advantage of them

Key Takeaways: PyTorch + HIP

- Operations that are not natively support by PyTorch are prime for converting to HIP kernels
  - For example, w2a8 is not natively supported
- First convert your code to pure C++ (no intrinsics) HIP kernels, then repeatedly optimize
- AMDGPU LLVM intrinsics and ISA give 80% of the speedup comes for 20% of the effort
- Create a custom PyTorch operator for your kernel
  - Faster runtime as it allows greater kernel fusion for torch.compile()
- Use PYTORCH_TUNABLEOP_ENABLED=1 environment flag for maximum performance
- Kernel optimization process could likely be automated
  - Start LLM on a certain task and repeatedly feed in errors until working solution
- Custom HIP kernels use significantly less memory than PyTorch

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.

LLVM is a trademark of LLVM Foundation
The OpenMP name and the OpenMP logo are registered trademarks of the OpenMP Architecture Review Board


AMD