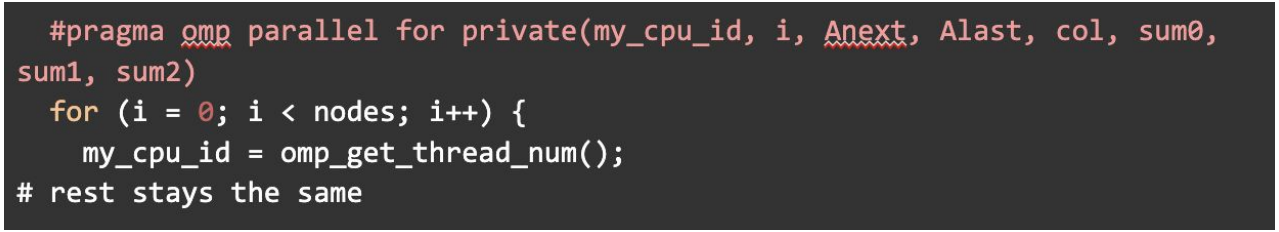
Evaluating OpenMP 4.5 Support on Compilers

Rakrish Dhakal, Pranav Sivaraman, Sathwik Yanamaddi

OpenMP Background

- OpenMP is a commonly used API for parallel programming (shared memory model)
- We already have used the API on CPUs:



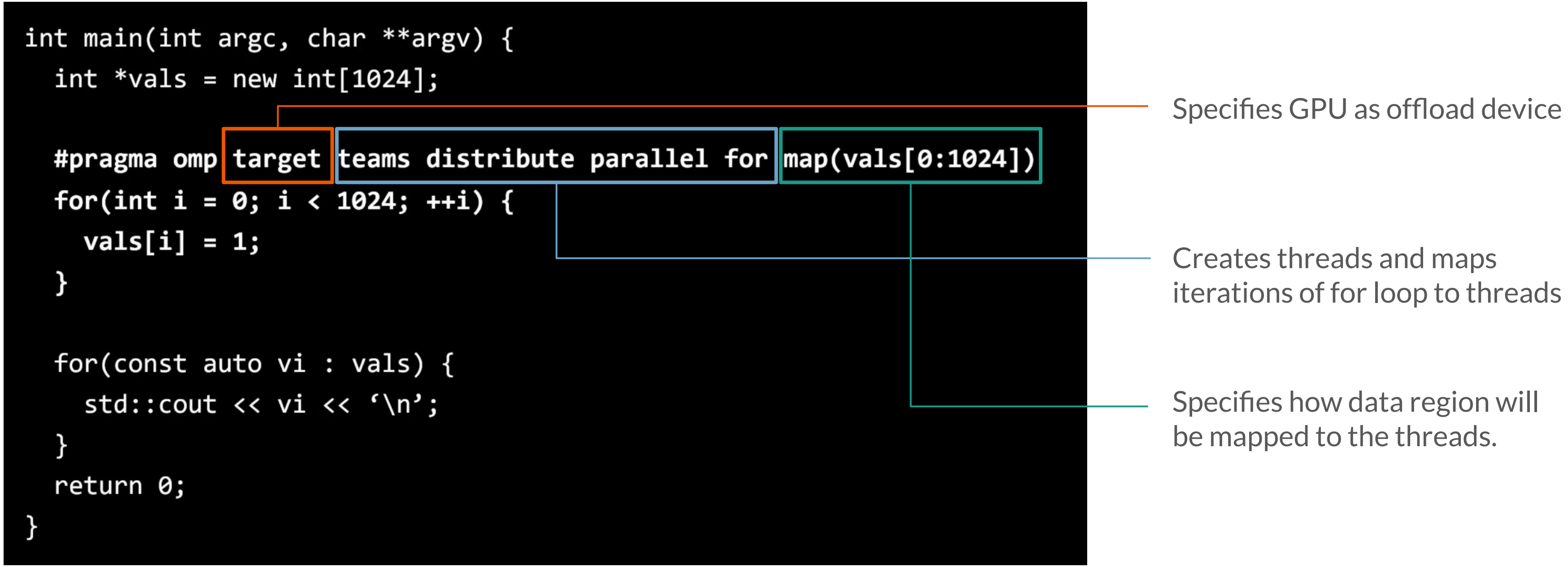
```
  #pragma omp parallel for private(my_cpu_id, i, Anext, Alast, col, sum0,
sum1, sum2)
  for (i = 0; i < nodes; i++) {
    my_cpu_id = omp_get_thread_num();
# rest stays the same
```

- Standard updated in 2013 to improve GPU offloading support

OpenMP 4.5 for GPUs

- GPUs have become significantly better at certain compute-heavy tasks than CPUs
- How can programmers offload these tasks to GPUs without writing different Instruction Set Architectures (ISAs) for existing apps?
- In 2013, OpenMP committee started extending the API with directives to offload code blocks to accelerators (starting from version 4.0)
- These API specifications have to be implemented by specific compilers

GPU Code Snippet



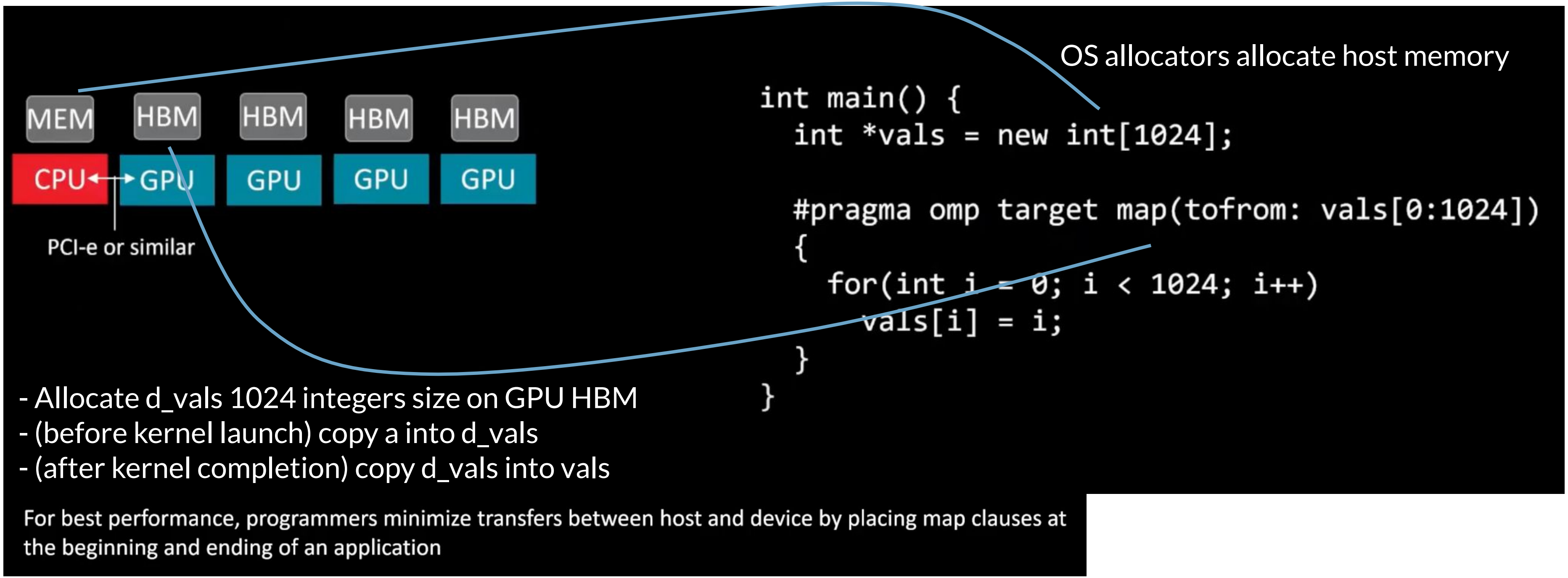
```
int main(int argc, char **argv) {
  int *vals = new int[1024];

  #pragma omp target teams distribute parallel for map(vals[0:1024])
  for(int i = 0; i < 1024; ++i) {
    vals[i] = 1;
  }

  for(const auto vi : vals) {
    std::cout << vi << ‘\n’;
  }
  return 0;
}
```

Specifies GPU as offload device

Creates threads and maps iterations of for loop to threads

Specifies how data region will be mapped to the threads.

Detailed look at GPU procedure



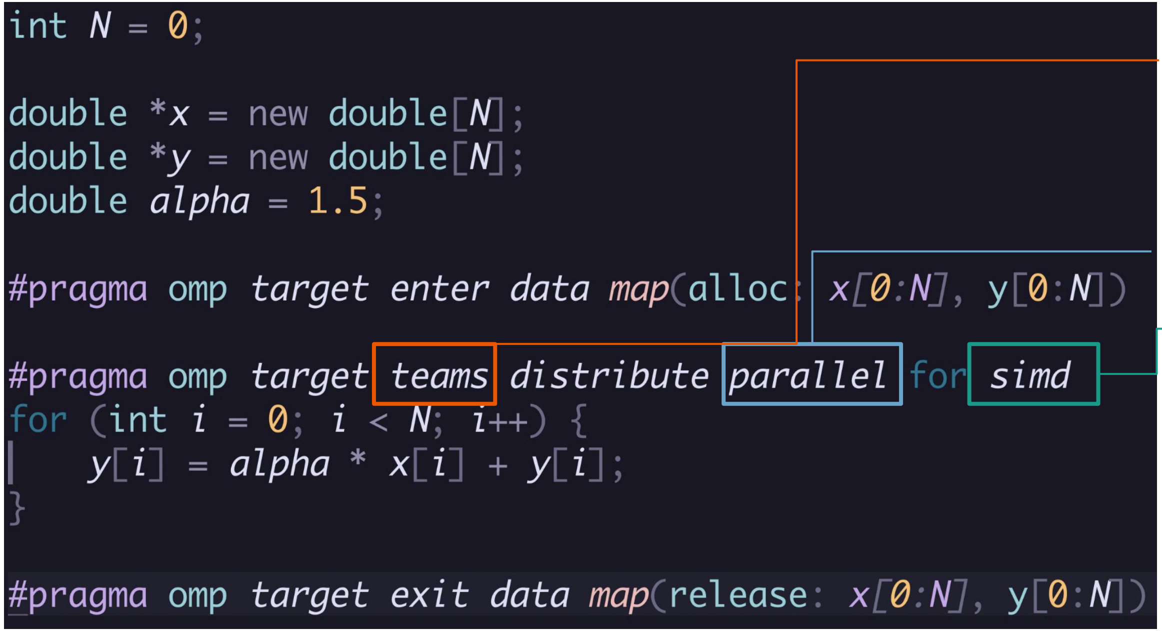
OS allocators allocate host memory

```
int main() {
  int *vals = new int[1024];

  #pragma omp target map(tofrom: vals[0:1024])
  {
    for(int i = 0; i < 1024; i++)
      vals[i] = i;
  }
}
```

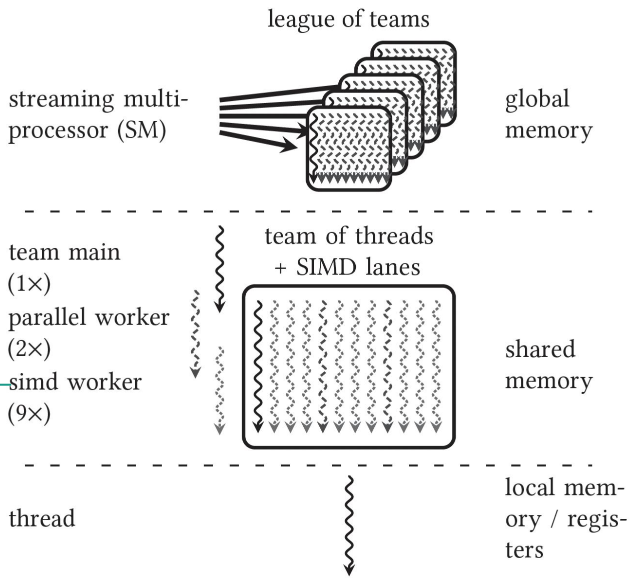
- Allocate d_vals 1024 integers size on GPU HBM
- (before kernel launch) copy a into d_vals
- (after kernel completion) copy d_vals into vals

For best performance, programmers minimize transfers between host and device by placing map clauses at the beginning and ending of an application

3 GPU Levels of Parallelism



```
int N = 0;

double *x = new double[N];
double *y = new double[N];
double alpha = 1.5;

#pragma omp target enter data map(alloc: x[0:N], y[0:N])

#pragma omp target teams distribute parallel for simd
for (int i = 0; i < N; i++) {
    y[i] = alpha * x[i] + y[i];
}

#pragma omp target exit data map(release: x[0:N], y[0:N])
```



Why Not CUDA/HIP/SYCL?

- Language extensions:
  - **CUDA**: API specific to Nvidia GPUs developed by Nvidia
  - **HIP**: CUDA-like API developed by AMD (can also run on Nvidia)
  - **SYCL**: higher-level, more portable, can run on GPUs, FPGAs, etc. (mostly used for Intel)
- Directive-based:
  - **OpenMP**: easily parallelize for loops and other sections of code by adding directives
  - **Benefits:** familiarity, ease of integration, code portability, simplicity
    - *Portability*: write code once, achieve good performance everywhere (ideally)

Same API, multiple implementations

- While OpenMP API provides specifications for GPU-based directives, it's up to each compiler how they want to implement these
- GCC will produce different machine code than Clang for the same source code

- **Goal of our project:** benchmark the performance of GPU offloading for various implementations of OpenMP 4.5 (i.e. various compilers)
  - We benchmark 2 applications across 4 compilers and compare the results

Experimental Setup - Applications

1. **BabelStream**
   - *Memory-bound* application
   - Designed to “measure memory transfer rates to/from global device memory on GPUs”
   - GPU equivalent of the STREAM kernels for CPUs, but allocates arrays on the heap (which is the parallel programming best practice)

2. **miniBUDE**
   - *Compute-bound* application
   - “Runs the energy evaluation for a single generation of poses repeatedly, for a configurable number of iterations”
   - Mini-app that implements Bristol University Docking Engine (BUDE) for HPC prog. models

Both of them created/maintained by University of Bristol HPC group, both have been used in publications to compare performance of programming models and runtimes

BabelStream Logo

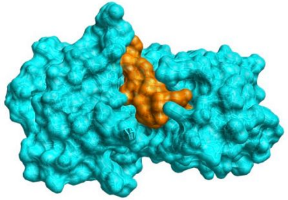
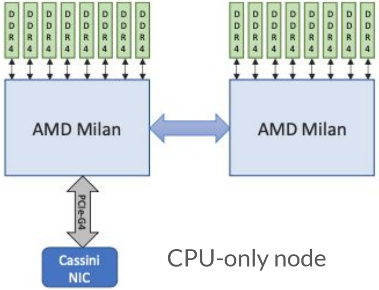


Diagram of protein simulation in BUDE

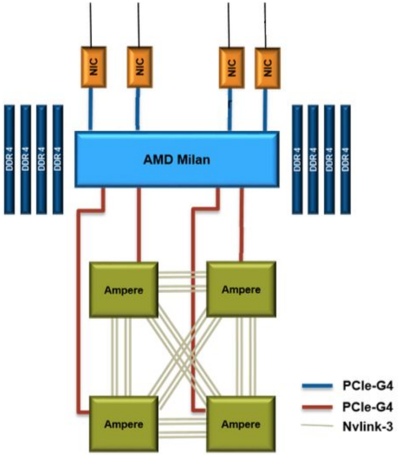
Image courtesies of UoB

Experimental Setup - Machine

- We ran our experiments on **Perlmutter**, an HPE Cray EX supercomputer at the **National Energy Research Scientific Computing Center (NERSC)** of the US Department of Energy
- System Specs:
  - 3,072 CPU-only nodes (2x AMD EPYC 7763)
  - 1,536 GPU-accelerated nodes (4x NVIDIA A100 + 1x AMD EPYC 7763)
- Network:
  - 3-hop dragonfly (SOTA)
- Storage:
  - 35 PB of flash disk space
  - 16 metadata servers, 274 I/O servers



CPU-only node



GPU-accelerated node

Image courtesies of NERSC

Experimental Setup - Machine

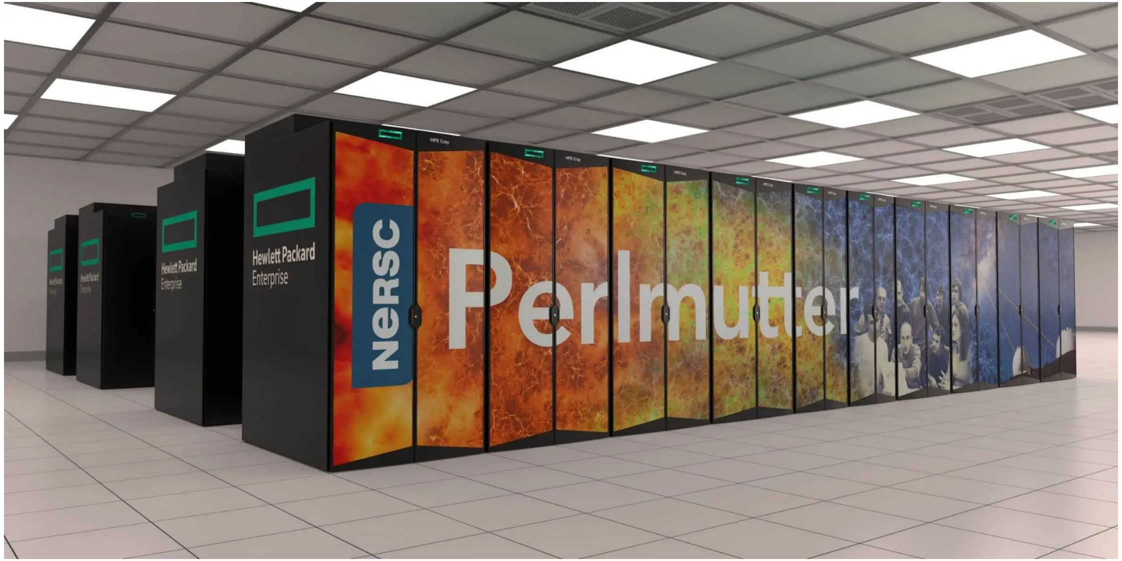


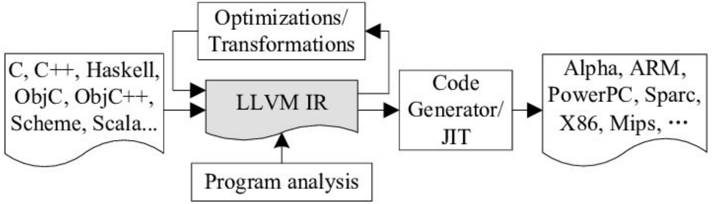
Image courtesy of Nvidia

Perlmutter (NERSC-9) – “World’s fastest AI supercomputer” built by HPE in partnership with NVIDIA and AMD

Experimental Setup - Compilers

We chose these compilers because they allow us to specify the GPU architecture, with support for NVIDIA & AMD GPUs.

1. **GCC (13.2.0)**
   - Open source, supports multiple languages, general-purpose
2. **Clang (17.0.4)***
   - Open-source, emphasizes compliance with standards, modular and extensible
3. **NVHPC (23.9)***
   - NVIDIA's suite of compilers optimized specifically for HPC on NVIDIA devices
4. **CCE (Cray) (16.0.1)***
   - Developed by Cray, optimized for performance on Cray hardware/architecture

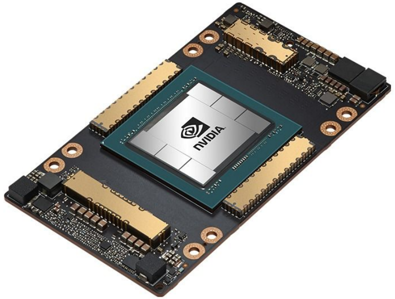

C, C++, Haskell, ObjC, ObjC++, Scheme, Scala...
Optimizations/ Transformations
LLVM IR
Program analysis
Code Generator/ JIT
Alpha, ARM, PowerPC, Sparc, X86, Mips, …

*Built off of LLVM

Image courtesy of Santosh Nagarakatte

Experimental Setup - Procedure

- We measured the **single-node, single-GPU performance** of each application across the compilers
- We chose the **default parameters** where appropriate (i.e. BabelStream)
  - miniBUDE is already somewhat tuned

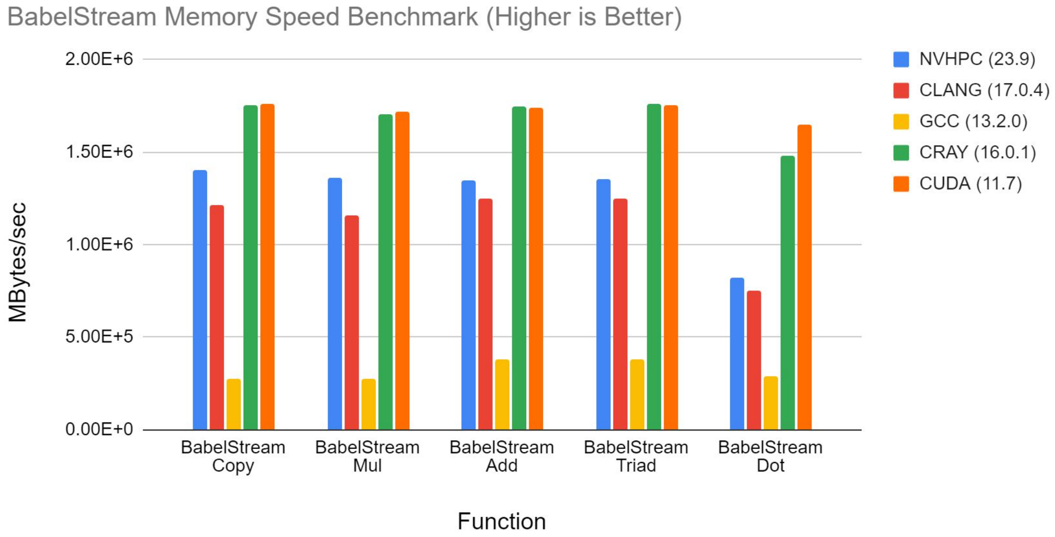


NVIDIA A100 GPU

Image courtesy of Nvidia

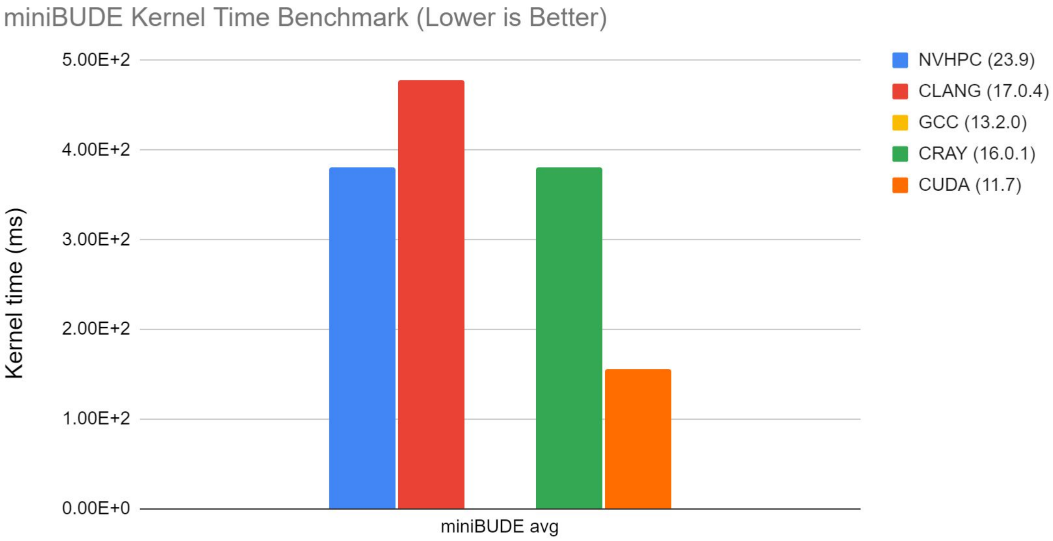
Results - BabelStream

- Comparison baseline: CUDA (native)
- Observations:
  1. CRAY outperforms other compilers, almost as good as CUDA
  2. CLANG performs slightly worse than NVHPC
  3. GCC performs the worst, by far, out of all compilers



Results - miniBUDE

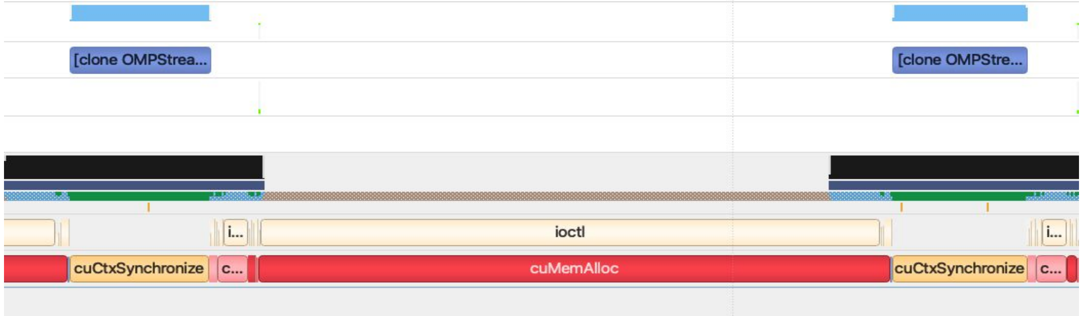
- Comparison baseline: CUDA (native)
- Observations:
  1. CUDA significantly outperforms all OpenMP implementations
  2. CLANG performs the worst, NVHPC and CRAY perform about the same
  3. GCC does not produce correct results



Why is GCC performing poorly?

Looking at the Trace

- GCC OpenMP Implementation for NVIDIA GPUs performs memory allocations and frees before and after each kernel launch. This behavior is not present in other implementations.
- The issue affects the results in miniBUDE as the main kernel (fasten) is executed multiple times before the final result is sent back to the CPU. Frees and allocations of data between kernel launches lead to the loss of previously computed results.

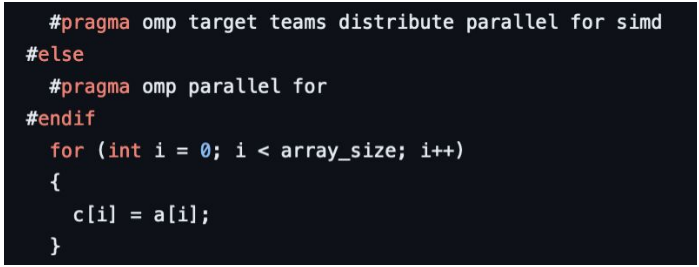


Trace of BabelStream compiled with GCC

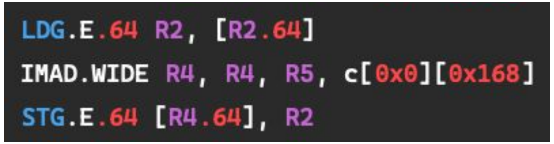
Why is GCC performing poorly? (Continued)

Looking at the Profiles/SASS

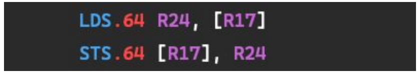
- GCC implementation introduces numerous barriers and synchronization in the actual assembly (SASS).
- For a simple copy kernel it loads into shared memory first and then writes it out to global memory.
- The actual assembly code for each kernel is significantly larger compared to alternative implementations.



```
  #pragma omp target teams distribute parallel for simd
#else
  #pragma omp parallel for
#endif
  for (int i = 0; i < array_size; i++)
  {
    c[i] = a[i];
  }
```



```
LDG.E.64 R2, [R2.64]
IMAD.WIDE R4, R4, R5, c[0x0][0x168]
STG.E.64 [R4.64], R2
```



```
LDS.64 R24, [R17]
STS.64 [R17], R24
```

Why is Cray doing so well?

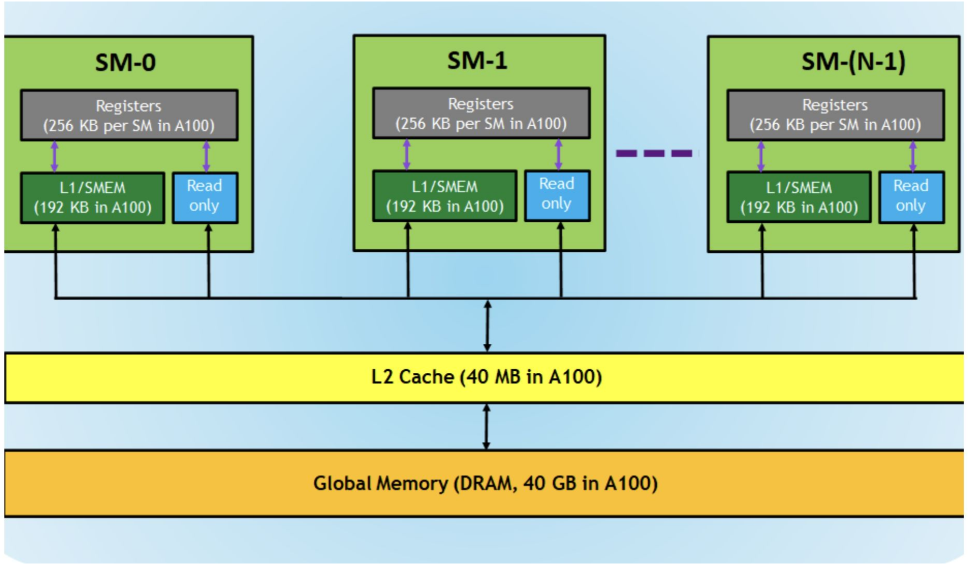
Looking at the Profiles

- No Barriers Present in the Cray version (Threads are Not Waiting)
- Cray is the manufacturer of the supercomputer so it make sense to expect good performance

NVIDIA wants to incentivize the use of CUDA and OpenACC not much progress has been made on its implementation

Getting closer to CUDA Baseline?

- The CUDA version of miniBUDE's main kernel and the utilizes dynamic shared memory, a feature absent in all OpenMP
- BabelStream also makes use of shared memory for it's dot kernel (reduction).
- In LLVM, the Third Level of Parallelism (SIMD) is not implemented, potentially contributing to a performance gap.



Conclusions

- Some implementations are farther ahead than others. These implementations may be tuned to the specific hardware

- Others need improving, but they also have to be more general for other GPU implementations (AMD, Intel)

- Tuning of the baseline CUDA kernels launch parameters could affect the performance