Porting QUDA from CUDA to other Backends

Xiao-Yong Jin (ANL)

P3HPC, September 1, 2020

Allstar Team
supported in part by lattice QCD ECP

Kate Clark (NVIDIA)	refactoring, pSTL
Dean Howarth (LLNL)	HIP
Xiao-Yong Jin (ALCF)	OpenMP
Balint Joo (OLCF)	HIP
Damon McDougall (AMD)	
James Osborn (ALCF)	DPC++
Patrick Steinbrecher (Intel)	
Alexei Strelchenko (FNAL)	DPC++ compatibility tool

This is an incomplete list
Unable to present all the work
Blame me if your name is missing

QUDA

- "QCD on CUDA" - https://github.com/lattice/quda
- Kate (lead developer) started it at Boston University in 2008; Now 31 authors listed in README
- Applications (BQCD, Chroma, CPS, MILC, TIFR, ...) use it as a GPU backend
- Maximize performance on Nvidia GPUs
  - Runtime autotuning (cooperative thread array size, shared memory size, grid size, ...)
  - `<jitify.hpp>` CUDA runtime compilation (optional)
  - Multi-GPUs
  - Tensor Cores; Multi-precision methods
  - Domain-decomposed preconditioners; Eigen solvers; Multi-grid solvers

Overall Porting Efforts

- Targets:
  - parallel STL
  - HIP
  - DPC++
  - OpenMP
  - Test conversion tools

- Strategy:
  1. Build System portability
     - `nvcc` to other SDK
  2. Refactor code, isolate CUDA
  3. Evolve generic "Backend API"
     - Portability of launching kernel
     - transform_reduce kernel
  4. Use conversion tools

Refactoring in Progress

- Organic process, instead of pre-designed
  - Look at the needs of backends
  - Find the right level of abstraction
- Started `include/quda_cuda_api.h`
- Want a unified kernel launching interface to avoid having backend specific code for every single instance of calling GPU kernels

HIP-ify

- hipify/hipexamine-perl.sh
  - Fails at complicated kernel launches
  - QUDA only cuda* functions
  - Feedback to refactoring process
- Can reuse most of the CUDA API skeleton files to HIP
- Build system
  - Using HIP-nvcc mode, "nvcc" as a C++ compiler
  - CMake ignore some flags to "nvcc" if not in CUDA mode
- Incremental hipifying the rest
- Refactoring the code first helps a lot

DPC++

- Compatibility tool: refactoring process is essential
- Rewriting kernel launching for more arguments
  - Passing around gridDim, blockDim, blockIdx, threadIdx

OpenMP

- Lack solid compiler supports, few 5.0 features
- Restrictive API
  - Requires workaround for variables contained inside any class object
    - pointers, reduction variables
  - Reduction cannot be initiated inside a thread

A Story of Reduction

Main reduction API as of Aug 16

After a major refactoring of the reductions

```
template <typename reduce_t, typename T, typename I, typename transformer, typename reducer>
class TransformReduce : Tunable
{ ...;
  TransformReduce(QudaFieldLocation location, std::vector<reduce_t> &result, const std::vector<T *> &v,
    I n_items, transformer &h, reduce_t init, reducer &r) : ...
  { ...; apply(0);}
  void apply(const qudaStream_t &stream)
  { ...; arg.launch_error = qudaLaunchKernel(transform_reduce_kernel<Arg>, tp, stream, arg); ...;}
  template <typename Arg> __launch_bounds__(Arg::block_size)
  __global__ void transform_reduce_kernel(Arg arg)
  { ...; arg.template reduce<Arg::block_size, false, decltype(arg.r)>(r_, j);}
};

template <typename T> struct ReduceArg { ...;
  template <int block_size, bool do_sum = true, typename Reducer = cub::Sum>
  __device__ inline void reduce(const T &in, const int idx = 0)
  { reduce2d<block_size, 1, do_sum, Reducer>(in, idx); }
  template <int block_size_x, int block_size_y, bool do_sum = true, typename Reducer = cub::Sum>
  __device__ inline void reduce2d(const T &in, const int idx = 0)
  { BlockLevelReduction; SaveBlockResult; CountBlocks; if(LastBlock){ ReduceAllBlockResults;}}
};
```

`BlockLevelReduction` uses cub

Porting Strategy

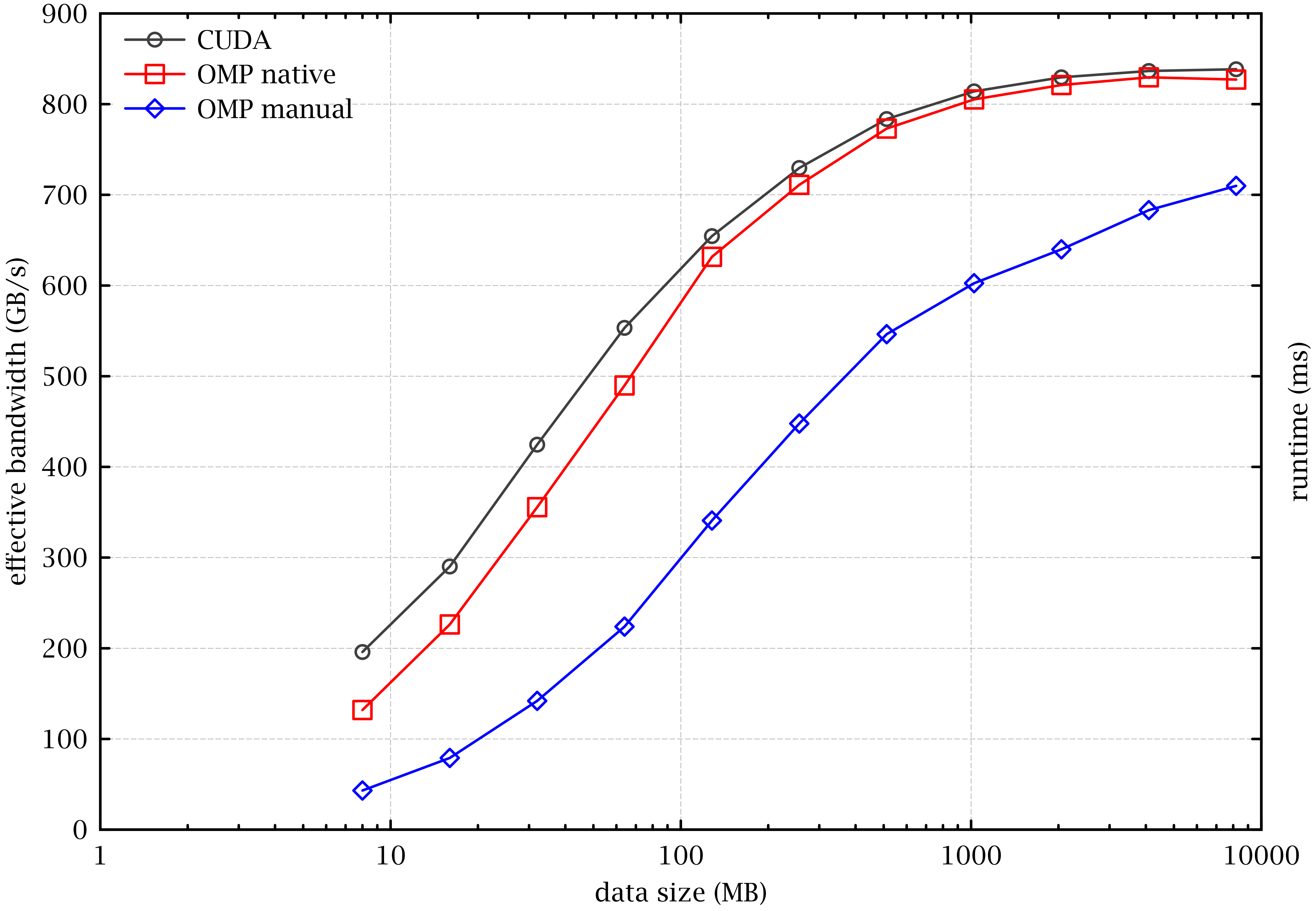
- HIP: straightforward
- DPC++:
  - `BlockLevelReduction`: Intel extension, `cl::sycl::intel::reduce`
    - Allows only Intel specific reducer, eg: `cl::sycl::intel::plus<T>`
  - `CountBlocks`: Intel extension, `cl::sycl::intel::broadcast`
- OpenMP: 🙀
  - Option A: rewrite kernel launching in loops and use reduction clause
  - Option B: implement `reduce2d`

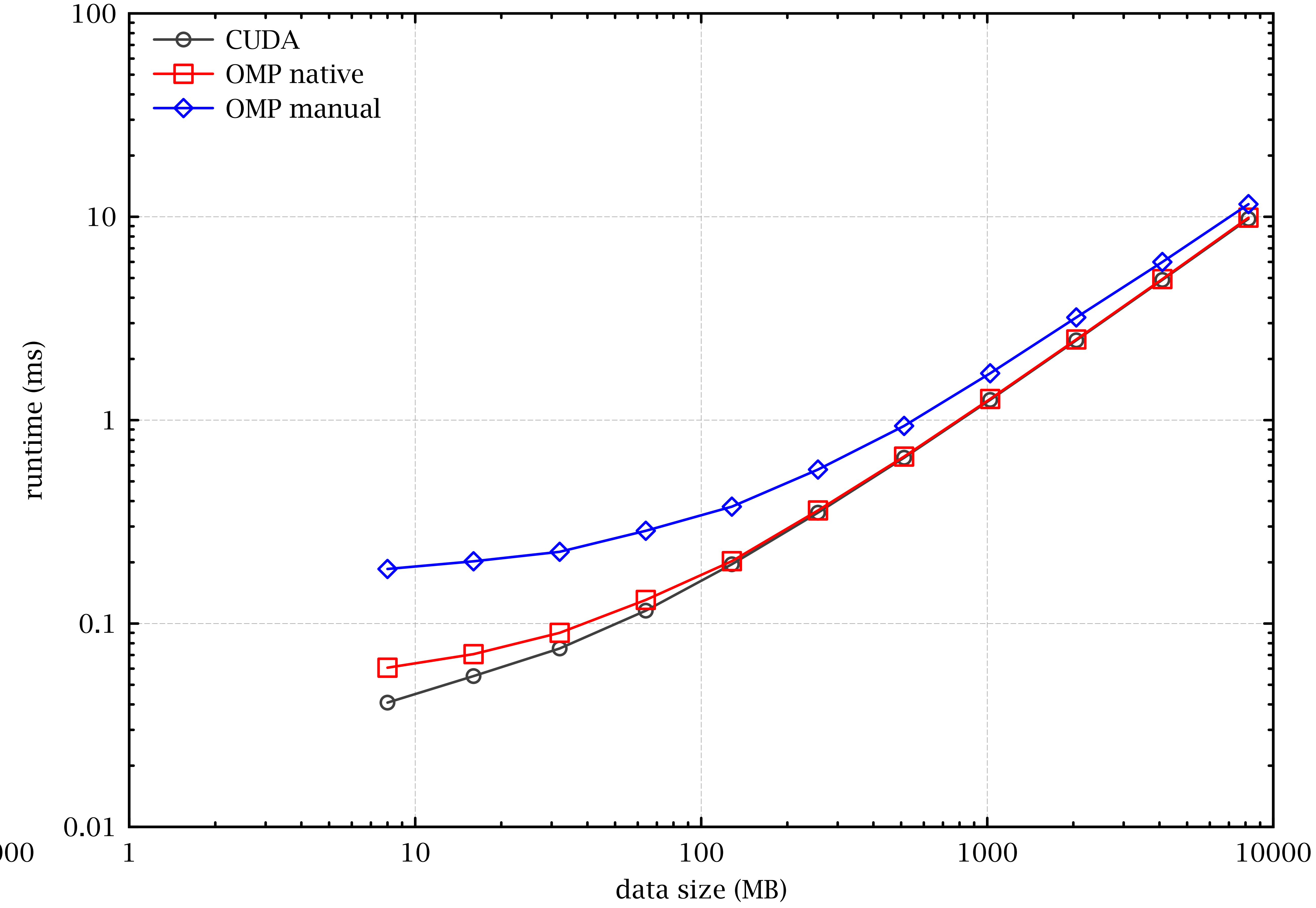
Implement `reduce2d` in OpenMP

- `BlockLevelReduction`: reduction per team
  - Part 1, synchronized (omp barrier) tree reduction
  - Part 2, reduce the partial tree in thread 0
- `SaveBlockResult`: access shared local memory
  - `omp allocate` has limited compiler support
  - modify kernel launch, init object inside `teams` before `parallel`
- `CountBlocks`: `omp atomic capture`

Effective Bandwidth & Latency

Minimum runtime of repeated runs with tuned parameters





Benchmarked on lassen (V100) at LLNL, xlC 16.01.0001.0008, CUDA 10.1.243

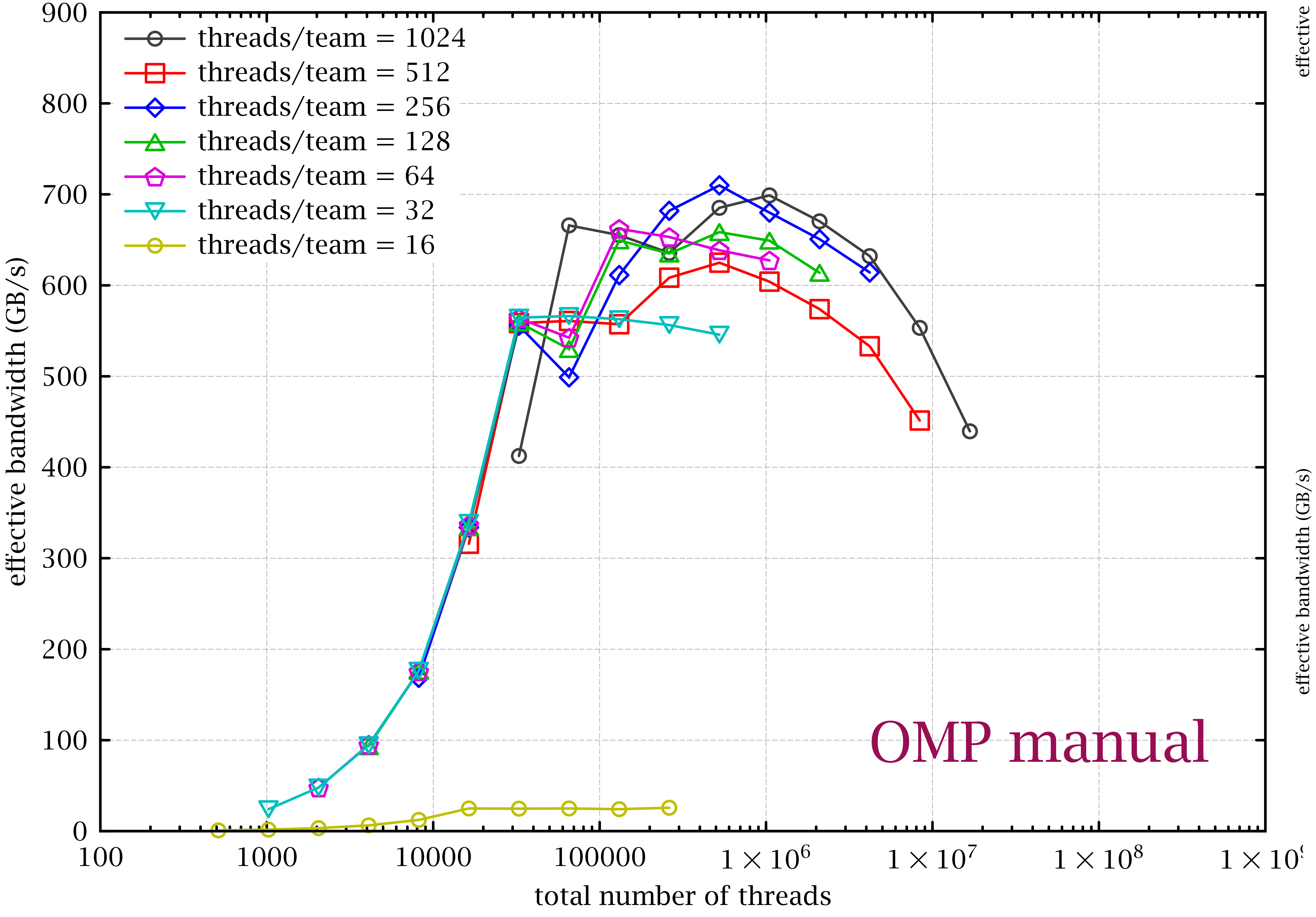
Measured Time of the First Run

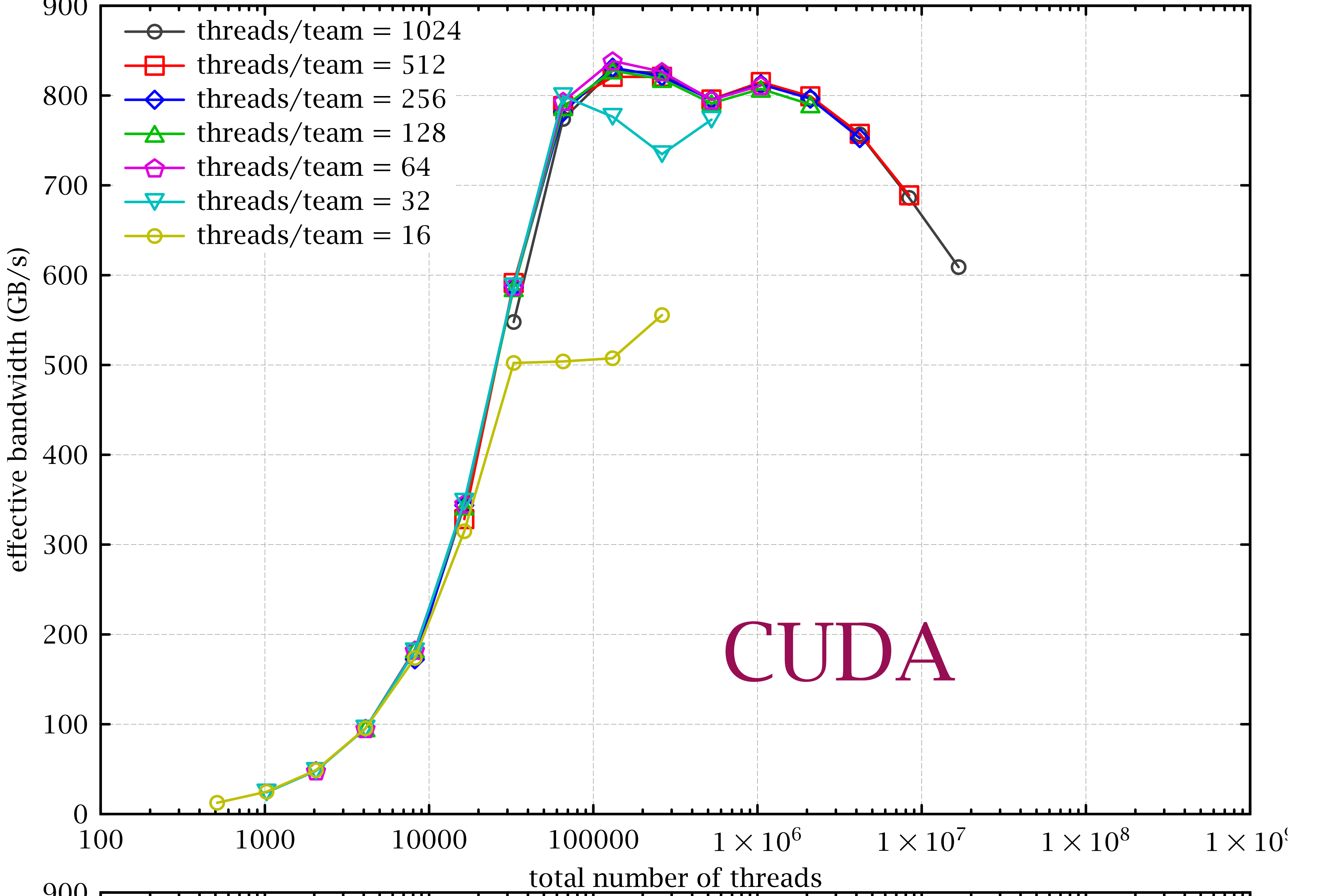
CUDA doesn't know the managed memory is not changed

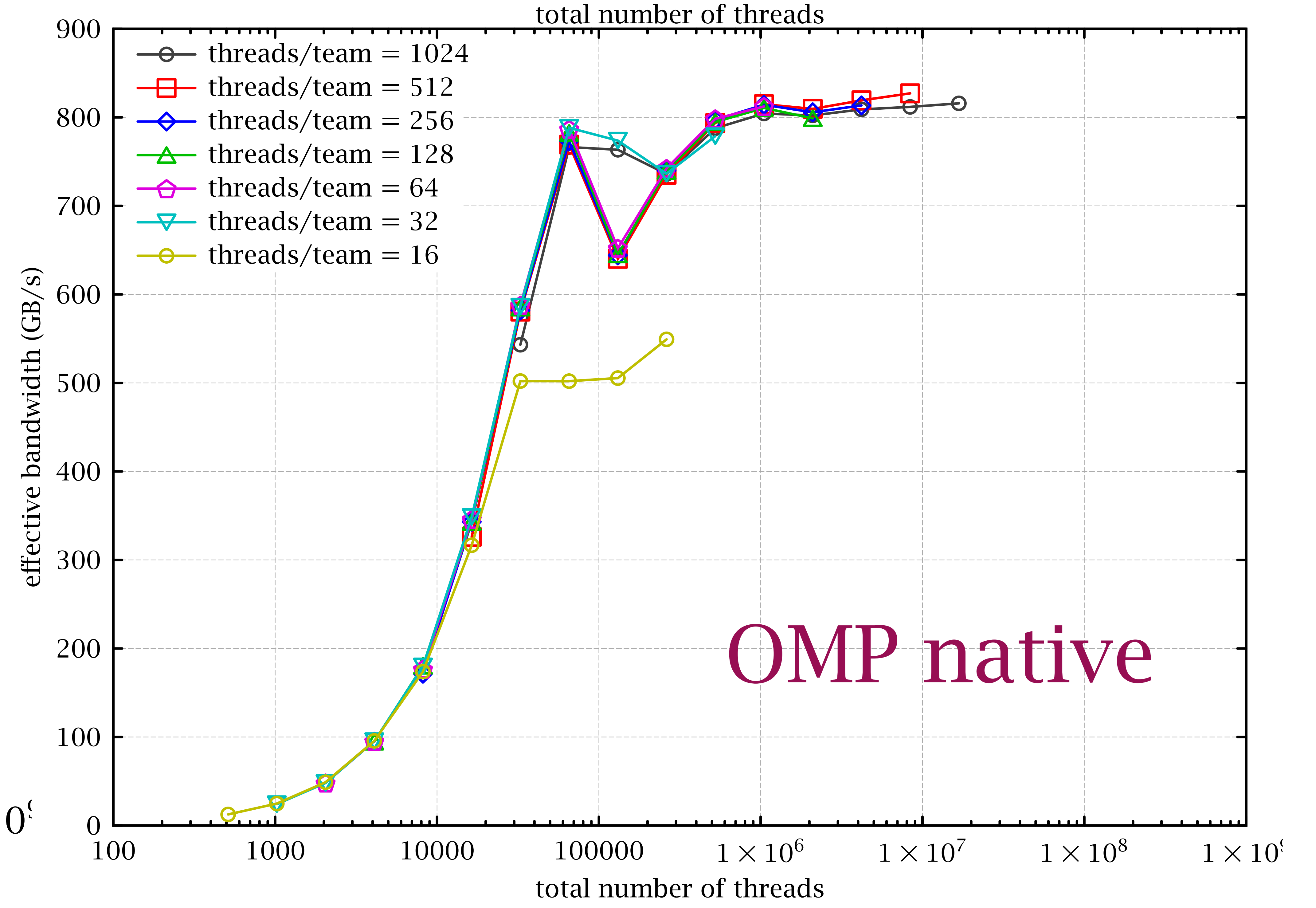
CUDA uses managed memory, while OpenMP copies data only once

Effective Bandwidth

Tuning threads/teams for 8 GB







OpenMP Plan for QUDA Reduction API

- Plan A: refactor the kernels so we can use OpenMP native reduction
  - Would we impose too much restrictions in the API?
- Plan B: continue with the current implementation of reduce2d
  - Optimize it more
  - Actually benchmark it in real kernels (we never do reductions only)

Summary

- We have a team of experts
  - Including the lead developer of QUDA, from Nvidia
  - Assistance from both Intel and AMD
  - Meet every week to discuss progress
- Refactoring code helps all backends
  - Currently try to deduplicate and simplify kernel launching code
  - Minimized exposure of CUDA API and abstract backends APIs
  - Still evolving following suggestions from code in other backends