# MocCUDA: Running CUDA Codes on Fugaku

**Slides contrib.: W. Moses (MIT), I. Ivanov (TokyoTech)**

Jens Domke, Dr. rer. nat. < jens.domke@riken.jp >
High Performance Big Data Research Team, RIKEN R-CCS, Kobe, Japan

# Initial DL4Fugaku team and Collaborators

Computer simulations create the future

## RIKEN R-CCS

### Operation team

#### Application tuning development unit

Unit Leader
Dr. Kazuo MINAMI

Researcher
Dr. Akiyoshi KURODA

Researcher
Dr. Kiyoshi KUMAHATA

Technical Staff
Mr. Kazuto ANDO

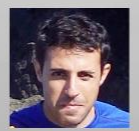
Researcher
Dr. Keigo NITADORI

### Research teams

#### High perf. AI system research team

Director
Dr. Satoshi MATSUOKA

Researcher
Dr. Aleksandr Drozd



Visiting researcher
Dr. Mohamed Wahib

#### High perf. BD research team

Team Leader
Dr. Kento SATO

Researcher
Dr. Jens Domke

#### Large-scale parallel numerical computing technology research team

Team Leader
Dr. Toshiyuki IMAMURA

Postdoctoral Researcher
Dr. Shuhei KUDO

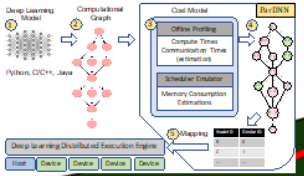
## Collaborators

(in alphabetical order)

- AIST
- ARM
- Cybozu
- Fujitsu Laboratories
- Fujitsu limited
- Linaro
- Tokyo Tech

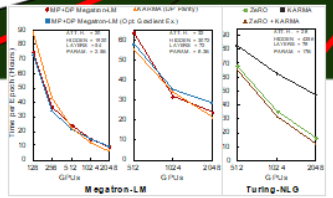
Most of the works on the slides are done by Akiyoshi Kuroda in application tuning development unit led by Kazuo Minami with significant assistance by Ikuo Miyoshi from Fujitsu limited

# Exploring and Merging Different Routes to O(100,000s) Nodes Deep Learning



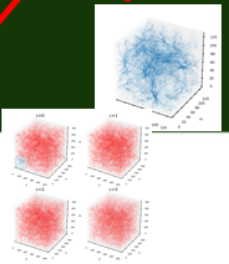

Non-intrusive graph-based partitioning strategy for large DNN models achieving superlinear scaling [1]

AIST, Koc U.



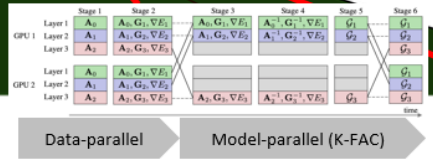

Out-of-core distributed training (pure data-parallel) outperforming SoTA NLP models on 2K GPUs [2]

AIST, Matsuoka-lab, RIKEN



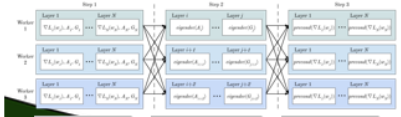
**Model-parallelism** enables 3D CNN training on **2K GPUs** with 64x larger spatial size and better convergence [3]

Matsuoka-lab, LLNL, LBL, RIKEN




A model-parallel **2nd-order method (K-FAC)** trains ResNet-50 on **1K GPUs** in 10 minutes [4]

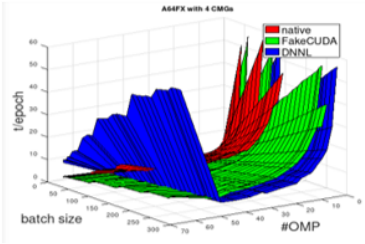
TokyoTech, NVIDIA, RIKEN, AIST




Layer-wise distribution and inverse-free design further accelerate K-FAC [5]

UT Austin, UChicago, ANL

## Merging Theory and Practice



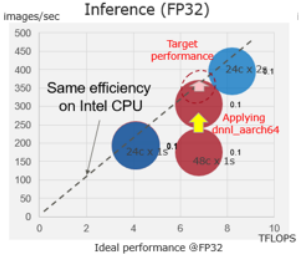

MocCUDA: Porting CUDA-based Deep Neural Network Library to A64FX and (other CPU arch.)

RIKEN, Matsuoka-lab, AIST

## Engineering for Performance Foundation

Porting CPU-based Deep Neural Network Library to A64FX chip

Fujitsu, RIKEN, ARM

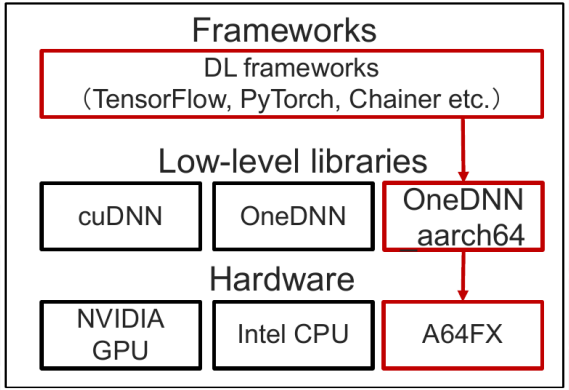




[1] M. Fareed et al., "A Computational-Graph Partitioning Method for Training Memory-Constrained DNNs", Submitted to PPoPP21
[2] M. Wahib et al., "Scaling Distributed Deep Learning Workloads beyond the Memory Capacity with KARMA", ACM/IEEE SC20 (Supercomputing 2020)
[3] Y. Oyama et al., "The Case for Strong Scaling in Deep Learning: Training Large 3D CNNs with Hybrid Parallelism," arXiv e-prints, pp. 1–12, 2020.
[4] K. Osawa, et al., "Large-scale distributed second-order optimization using kronecker-factored approximate curvature for deep convolutional neural networks," Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit., vol. 2019-June, pp. 12351–12359, 2019.
[5] J. G. Pauloski, Z. Zhang, L. Huang, W. Xu, and I. T. Foster, "Convolutional Neural Network Training with Distributed K-FAC," arXiv e-prints, pp. 1-11, 2020.

# Internal discussions (early spring ‘20)

- Prof. Matsuoka eye-opening remarks a long time ago:
  **“A64FX is more like a GPU than a CPU”**




- Brainstorming Alex & Wahib & myself
  - CUDA does gemm-based conv… why? -> Memory BW
  - oneDNN focuses on direct conv.  (gemm-based only “for debug”)
  - NNPACK / native CPU backends for “normal” CPUs all slow
  - If A64FX *“is a GPU”*, then why don’t we mimic its computation?

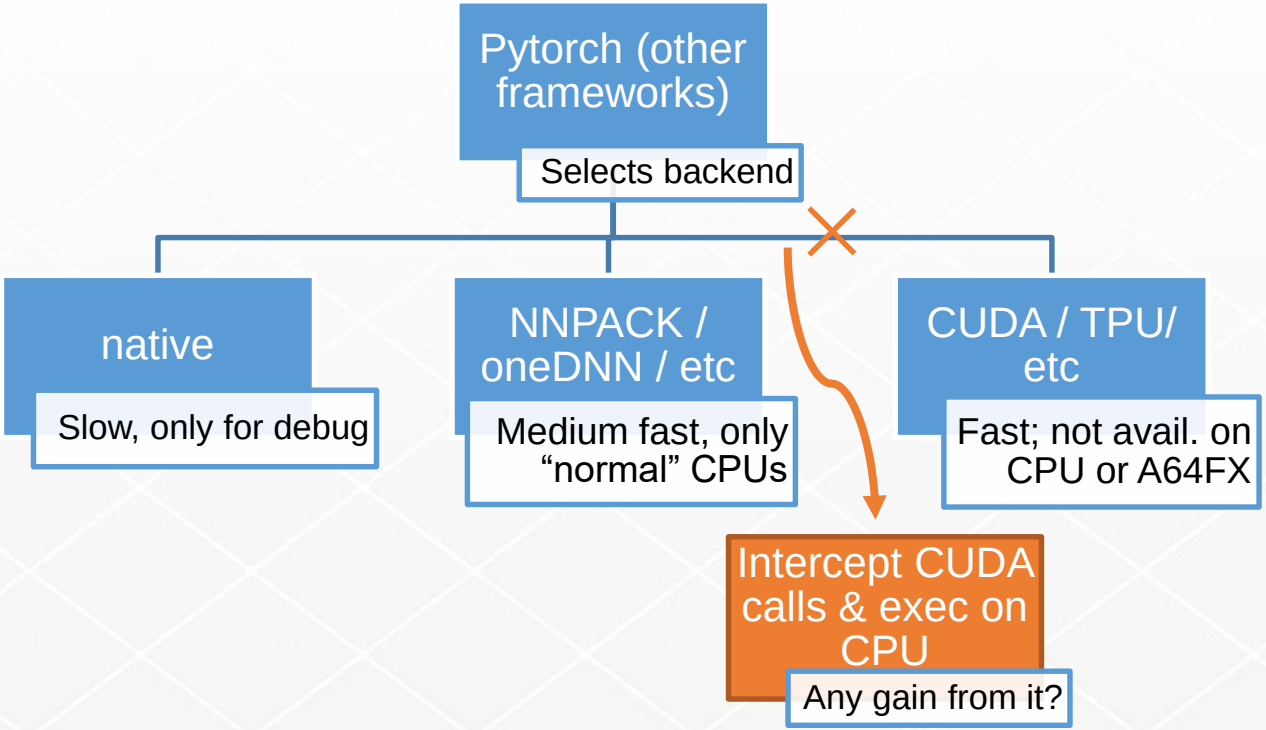
- **Option A**: use oneDNN’s internal gemm-based conv
  - need rewrite of interface to pytorch, or within high-level API of oneDNN ☹
  - refactoring torch scripts still necessary; Amdahl’s law issues still present ☹

# oneDNN for pytorch (and TF, +others…)

- Disadvantages of Intel's oneDNN approach
  - Tedious to port to A64FX (**years of engineering** by Fujitsu)
  - GEMM-based convolution not exposed
  - **Tuned for "normal" CPUs** with assumption: Memory is slow
  - Pytorch **scripts need to be modified** (convert model/tensor with X.to_mkldnn())
  - **Amdahl's law** problem (maybe too much sequential pytorch stuff in betw. parallel DNNL sections)

# Option B: can we replace CUDA RT & cuDNN?





# MocCUDA for x86 and A64FX (cuda-”native”)



- Architecture (only functions implemented if called by pytorch):
  - Wrapper library for CUDA runtime ➔ Easy ☺
  - Wrapper libs for cudnn (& cublas) ➔ medium hard (& trivial), but **no reference code** available (hence, took more time)
  - Wrapper libs for native cuda kernels (<<<…>>> in torch’s .cu files) ➔ annoying, **non-trivial coding** / reverse engineering, but doable ☹
  - Async work dispatch queues (cf. cuda streams) ➔ use Apple’s GCD
  - Finally: use **SSL2** for BLAS ops; use **Horovod** for MPI/multi-node
- Time: only 1-2 months of R&D without prior knowledge of CUDA programming or how to write DL kernels (bnorm, maxpool, conv, etc.)!
- Can run Resnet50 (batchsize >= 2) with **LD_PRELOAD=moccuda.so**

# Experimental MocCUDA Benchmarking

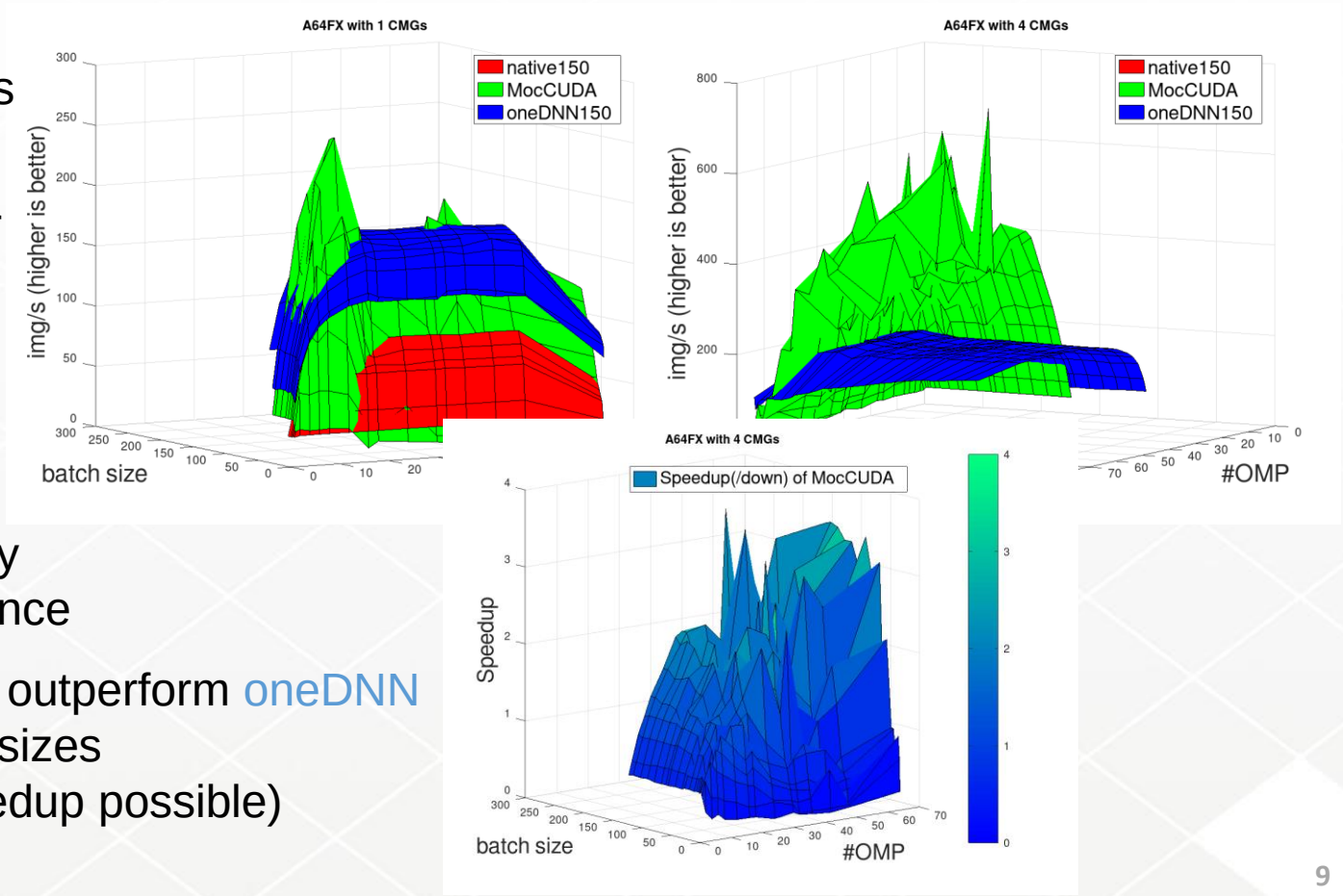
- **Tests on Fugaku**:
  - Alex' github.com/undertherain/benchmarker/ (Infer.: conv2d layer; Train.: Resnet50 w/ synthetic img)
  - Fujitsu's Resnet50 test_train.py script
  - Horovod's synthetic Resnet50 benchmark
- **Fujitsu's** official **pytorch v1.5** on Fugaku (FJ's fcc + oneDNN + SSL2)
- Self-compiled **Pytorch (v1.4) with CUDA** support (nvcc + clang13 + SSL2)
  - CUDA(4Arm)/cudnn "installed" in $HOME from RPMs
  - +3 changes to prevent inline of some functions (➔ not possible in v1.5 anymore)
- Execute test_train.py (10 epochs), and benchmarker (6 epoch) on A64FX, eg:

*for CMGs in (1, 2, 3, 4)*
*for OMP in (1, 2, …, 64)*
*for batch_size in (1, 2, …, 288)*
*test_train.py (--type **cpu_nomkl | cpu_mkltensor | gpu**) & eq. for benchmarker*

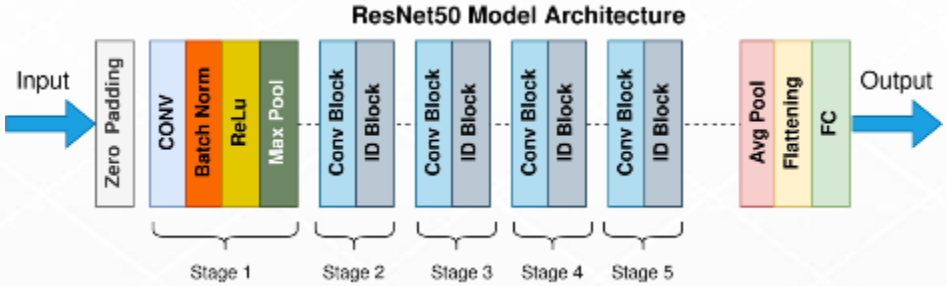
# Conv2d layer: Img/s (top) & Speedup (bottom)

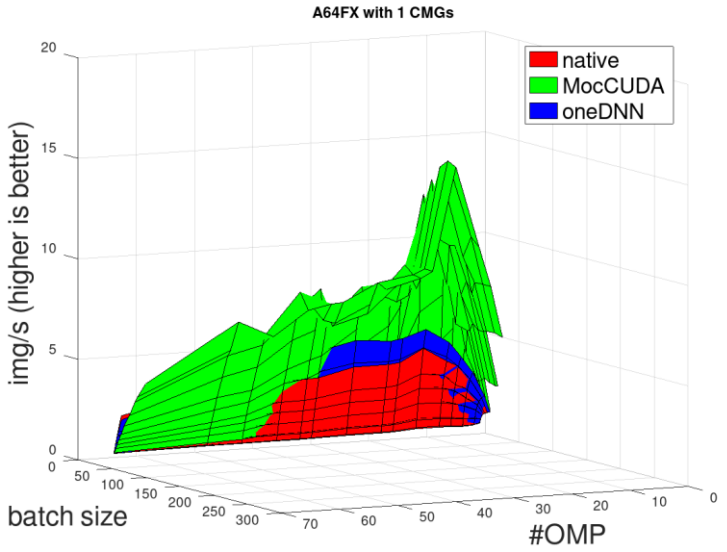


- Conv2d can be implemented as im2col+gemm
- Higher is better
- Possible to run on 1-to-4 A64FX CMGs
- Native CPU backend usually worst performance
- MocCUDA can outperform oneDNN for large batch sizes (up to ~4x speedup possible)

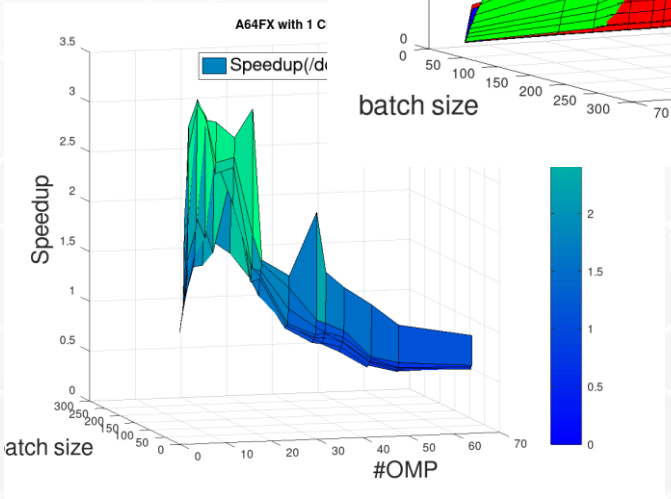


# Benchmarker: Resnet50 training



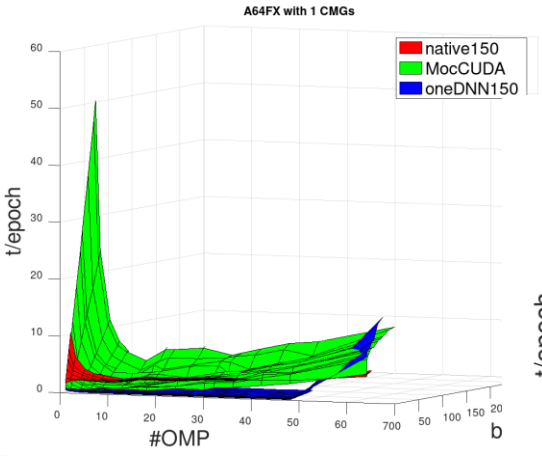



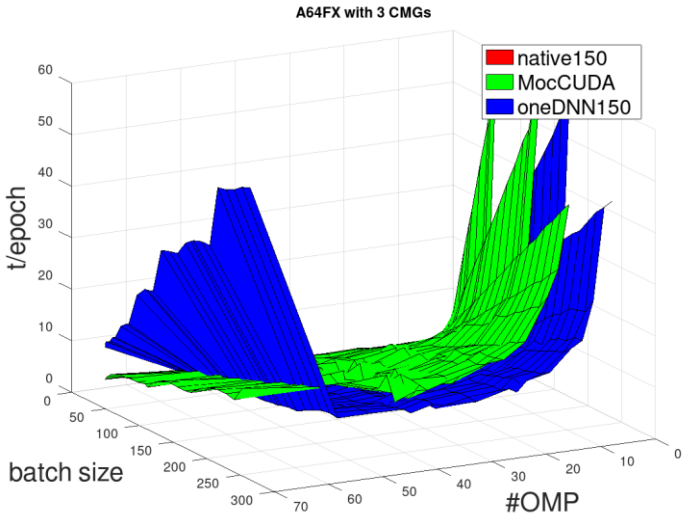

- Full multi-layer resnet50 training run (forward and back propagation) possible
- Higher is better
- MocCUDA outperforms oneDNN by up to 3x

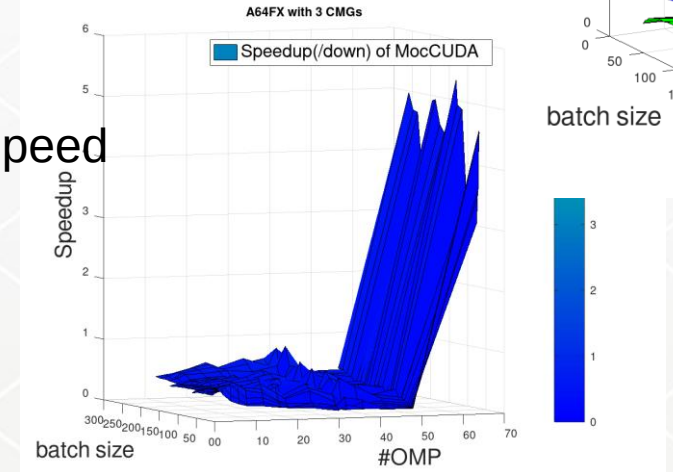


# Fujitsu's Resnet50 test_train.py script

- Lower is better
- Native slow; not scaling with batch size (OOM issues)
- oneDNN big problem with #OMP > #cores
- MocCUDA almost competitive
  - small #OMP: ~1/2 speed
  - #OMP ~= #cores: 5%-20% slower
  - >5x speedup for #OMP > #core



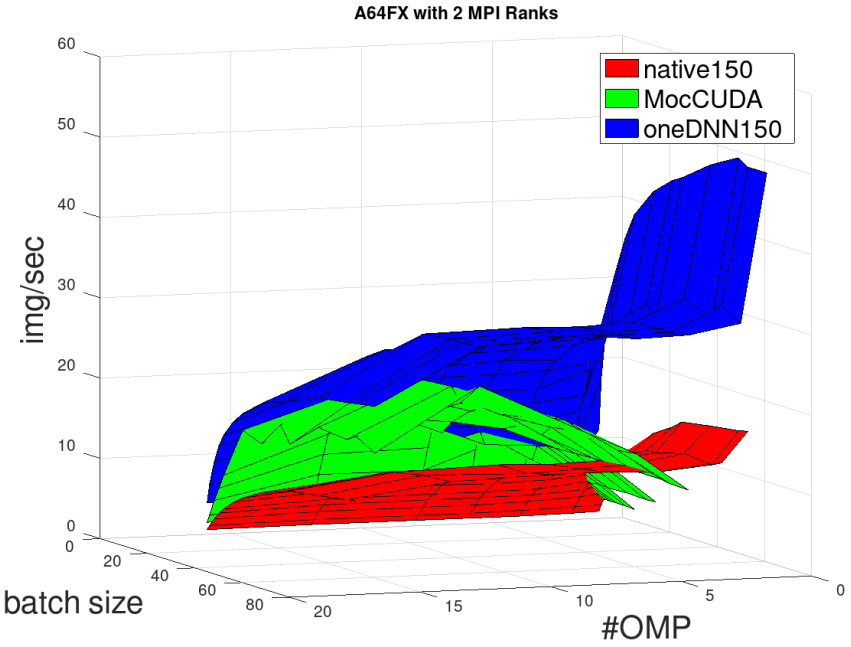



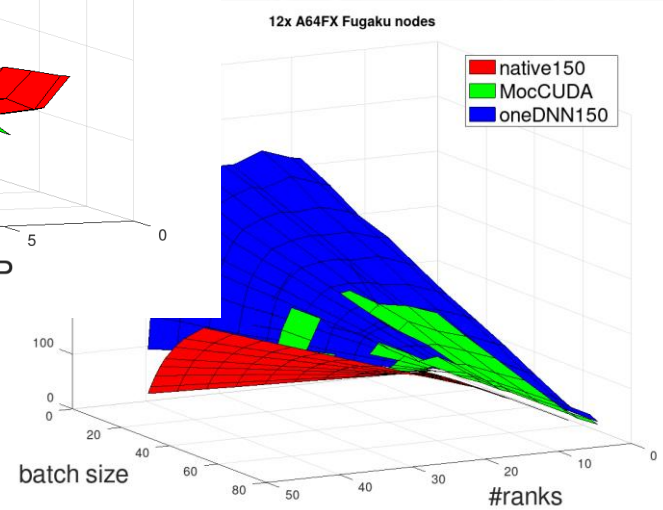



# Horovod's synthetic Resnet50 benchmark



- Higher is better
- Left: 2 ranks on 1 node; Right: scaling #MPI ranks with fixed #OMP=12
- Native slow
- oneDNN best, but with **odd performance behavior** (#OMP=1 best; >1 decreasing)
- MocCUDA close to oneDNN with 12 cores per MPI rank / CMG



# Primary remaining issue: native CUDA kernels



- Pytorch has various tensor operations implemented in native cuda (add, mul, threshold, softmax, … *more complex ops*)
- Number is reducing over time
  ➔ more and more move to libcudnn, libcublas, libcufft, ….
- Number will likely _not_ decrease to 0
  ➔ some fn not performance-relevant enough to migrate

➔ Collaboration on **automatic Cuda2OpenMP translation/compilation**

- Prior art exists (eg. GPU Ocelot, …) but is outdated
- **Approach A:** LLVM-IR (collab. w/ Ivan R. Ivanov @TokyoTech)
- **Approach B:** LLVM-MLIR (collab. w/ William S. Moses @MIT)

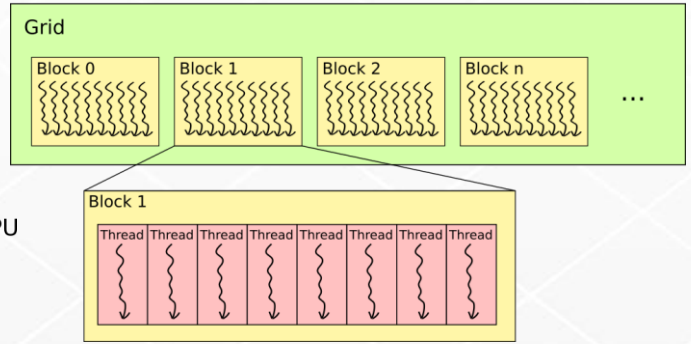
# Option A: LLVM-IR (I. Ivanov @TokyoTech)



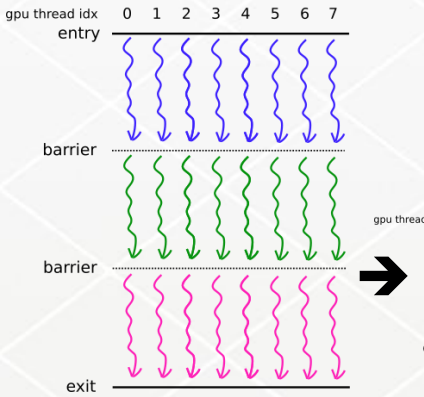
- CUDA execution model:
  - Kernels exec. in blocks in grids
  - Threads in blocks run in parallel
  - No guarantee about the order of blocks or parallelism of blocks
- In theory easily mapped to **6-way nested loop + OMP**
  - GPU thread ?= CPU thread
    ➔ oversubscribes CPU
    ➔ bad cache access ☹
  - Parallelize over blocks?
    ➔ GPU barriers… ☹

```
1 __global__ void VecAdd(float* A, float* B, float* C)
2 {
3   int i = threadIdx.x;
4   C[i] = A[i] + B[i];
5 }
```

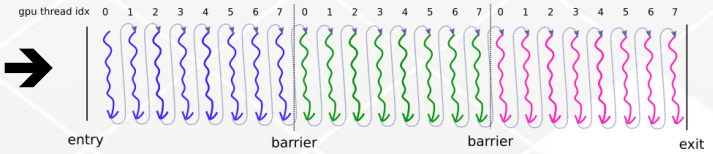
Kernel execution structure




Execution of a block on the GPU

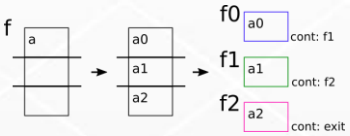


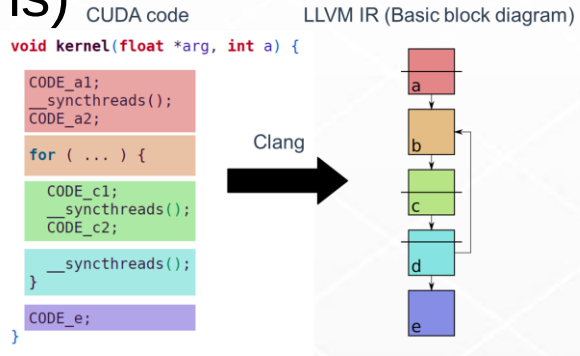

Execution of a block on a single CPU thread



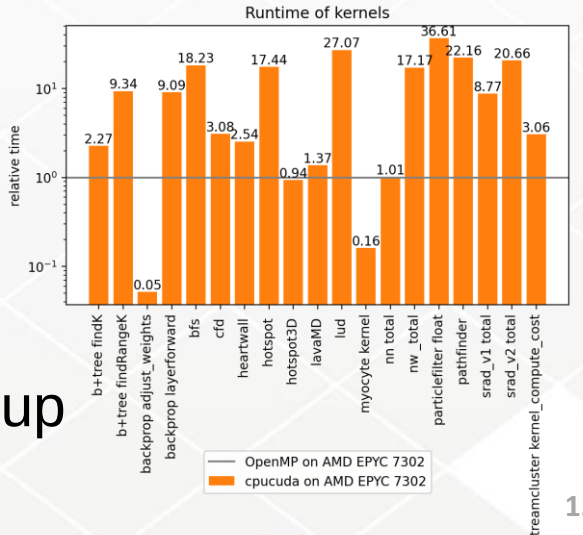
# Code transformation: continuation kernels

- Generate **continuation kernels** (functions)
- **Live variable analysis** to find state (variables) necessary to preserve
- Done on **LLVM IR** level



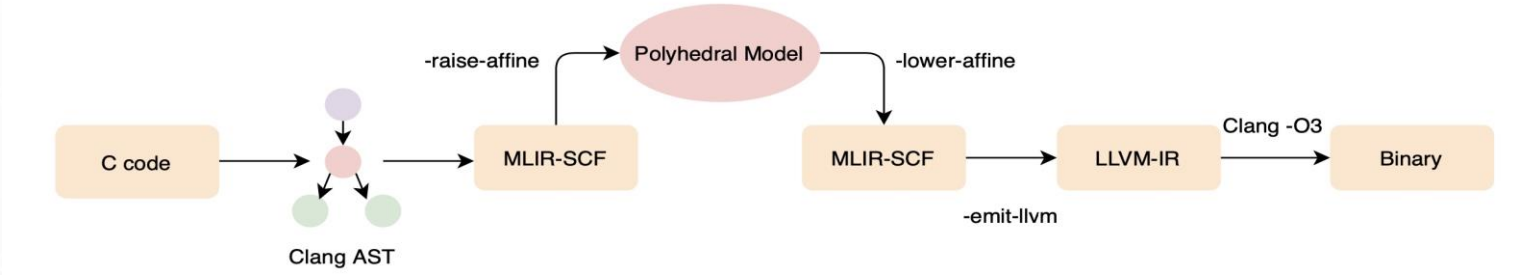




- Evaluation
  - **Rodinia** (has OpenMP & CUDA implementations of the same problem
  - Avg. ***cpucuda*** runtime 2.5x slower than native OpenMP (ranging from 20x speedup to 37x slowdown



# Option B: LLVM-MLIR (W. Moses @MIT)

- **Polygeist** compilation flow (github.com/wsmoses/Polygeist )




- Generic C/C++ frontend generates “standard” MLIR (multi-level interm. Rep.)
- Raising transformations for transforming “standard”→polyhedral MLIR (affine)
- Embedding of existing polyhedral tools (Pluto, CLooG) into MLIR
- Novel transformations and optimizations (statement splitting, reduction detection, etc) that rely on high-level compiler representation
- End-to-end evaluation of standard polyhedral benchmarks (Polybench)

# Use Polygeist to translate CUDA→OpenMP



- **Parallelize within blocks** and first-class representation of parallelism
  - Maintain GPU parallelism in a form understandable to the compiler
  - Enables optimization between caller and kernel

```
__device__ int sum(int* in, int n);
__global__ void normalize(int *out, int *in, int n) {
  int tid = threadIdx.x;
  if (tid < n)
    out[tid] = in[tid] / sum(in, n);
}

void launch(int* out, int* in, int n) {
  normalize<<<nblocks, nthreads>>>(out, in, n);
}
```

→

```
func private @_Z3sumPii(memref<?xi32>, i32) -> i32

func @_Z6launchPiS_i(%arg0: memref<?xi32>, %arg1: memref<?xi32>,
                     %arg2: i32) {
  %c1 = arith.constant 1 : index
  %c0 = arith.constant 0 : index
  scf.parallel (%arg3) = (%c0) to (%arg2) step (%c1) {
    %2 = memref.load %arg1[%arg3]
    %sum = call @_Z3sumPii(%arg1, %arg2)
    %4 = arith.divsi %2, %sum : i32
    memref.store %4, %arg0[%arg3]
    scf.yield
  }
  return
}
```

```
func private @_Z3sumPii(memref<?xi32>, i32) -> i32

func @_Z6launchPiS_i(%arg0: memref<?xi32>, %arg1: memref<?xi32>,
                     %arg2: i32) {
  %c1 = arith.constant 1 : index
  %c0 = arith.constant 0 : index
  %sum = call @_Z3sumPii(%arg1, %arg2)
  scf.parallel (%arg3) = (%c0) to (%arg2) step (%c1) {
    %2 = memref.load %arg1[%arg3]
    %4 = arith.divsi %2, %sum : i32
    memref.store %4, %arg0[%arg3]
    scf.yield
  }
  return
}
```

- Remaining issue again: **Synchronization Lowering**
  - Efficiently **lower a top-level sync by distributing** the *parallel_for* loop around the sync
  - Store registers or **re-compute values** which are required in 2nd loop
  - **Sync. within control flow** (for, if, while, etc.) can be lowered by splitting and and interchanging loops

```
parallel_for %i = 0 to N {
  codeA(%i);
  sync_threads;
  codeB(%i);
}
```

→

```
parallel_for %i = 0 to N {
  codeA(%i);
}
parallel_for %i = 0 to N {
  codeB(%i);
}
```

```
parallel_for %i = 0 to N {
  codeA(%i);
  for %j = … {
    codeB1(%i, %j);
    sync_threads;
    codeB2(%i, %j);
  }
  codeC(%i);
}
```

→

```
parallel_for %i = 0 to N {
  codeA(%i);
}
parallel_for %i = 0 to N {
  for %j = … {
    codeB1(%i, %j);
    sync_threads;
    codeB2(%i, %j);
  }
}
parallel_for %i = 0 to N {
  codeC(%i);
}
```

→

```
parallel_for %i = 0 to N {
  codeA(%i);
}
for %j = … {
  parallel_for %i = 0 to N {
    codeB1(%i, %j);
    sync_threads;
    codeB2(%i, %j);
  }
}
parallel_for %i = 0 to N {
  codeC(%i);
}
```

→

```
parallel_for %i = 0 to N {
  codeA(%i);
}
for %j = … {
  parallel_for %i = 0 to N {
    codeB1(%i, %j);
  }
  parallel_for %i = 0 to N {
    codeB2(%i, %j);
  }
}
parallel_for %i = 0 to N {
  codeC(%i);
}
```

# Summary and Job/Collaboration Opportunities

Computer simulations create the future

- Advantages of MocCUDA & LLVM-IR:
  - **Full control over SW stack**; tune as we like (algos/code) w/o Intel
  - CUDA impl. (torch/etc.) implicitly supports async dispatch -> **no Amdahl's law issues**
  - **Implicit support for other DL frameworks** (incl. those without oneDNN support)
  - **Easily integrate diff. precisions** / kernel fusion (analyze GCD queues) / SVE / etc.
  - Usage **potential far beyond just DL** framework ➔ backporting GPU codes to HBM-based x86/Arm CPUs
- Collaborations and **Job opportunities**:
  - Our research teams and open positions: https://www.riken.jp/en/research/labs/r-ccs/ and https://bit.ly/3faax8v
- **Internship/fellowship** (Bachelor➔PhD):
  - www.riken.jp/en/careers/programs/index.html
  - www.r-ccs.riken.jp/en/about/careers/internship/
- **Supercomputer Fugaku**:
  - Apply for node-hours: www.r-ccs.riken.jp/en/fugaku/user-guide/
  - Interactive, virtual tour: www.r-ccs.riken.jp/en/fugaku/3d-models/ and www.youtube.com/watch?v=f3cx4PGDGmg

< jens.domke@riken.jp >