# Implementing and Evaluating OpenCL on ARMv8 Multi-Cores and Many-Cores

Jianbin Fang

Software Institute, College of Computer
National University of Defense Technology

*j.fang@nudt.edu.cn*

September 24, 2017

# Overview

1. Background and Motivation
   - Introducing Feiteng Processors
   - What is OpenCL?
   - Why Bother?
2. Design and Implementation
   - Our OpenCL Design
   - Kernel Compiler
   - Runtime System
3. Experimental Results
4. Take-Home Message

# A Bird-View of FT-1500A

- FT-1500A is an ARMv8-based 16-core CPU from Phytium
- Each SIMD core has a private 32KB data cache
- Very four cores share a 2MB L2 data cache
- All the 16 cores share a 8MB L3 data cache

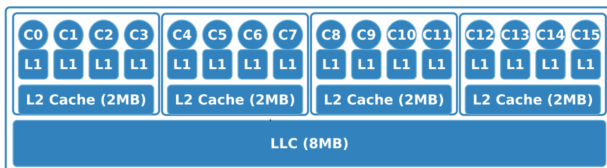
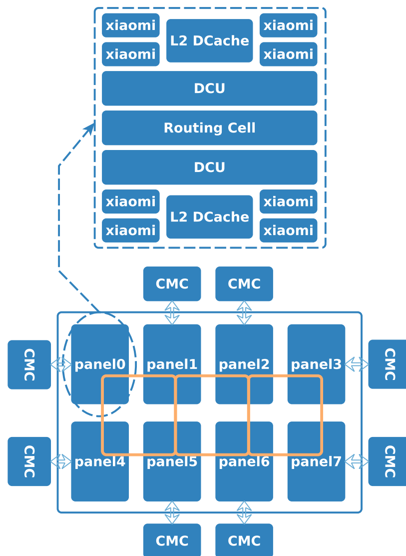



Figure: The FT-1500A CPU

# A Bird-View of FT-2000

- FT-2000 has 8 panels, each having 8 xiaomi cores (64 cores in total)
- Again every four xiaomi cores share a 2MB L2 data cache (32MB in total)
- Maintaining directory-based cache coherence with DCU and routing cells
- w/ a mesh topology on chip network
- 8 Cache & Memory Chips (CMC), containing 128MB L3 data cache and 16 DDR3 channels
- Panel-based data affinity architecture (i.e., 8 NUMA nodes)
- Two 16-lane PCIE3.0
- ECC and parity protection



# What is OpenCL?

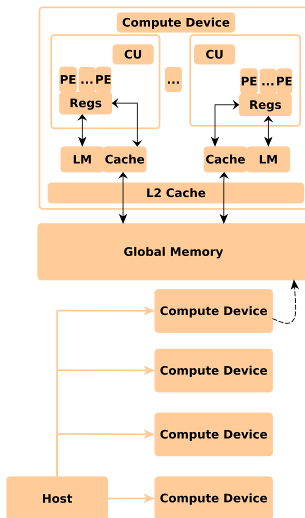
- **Standard** for the development of data parallel applications
- Supports CPUs, GPUs, and other accelerators such as Phis, DSPs, FPGAs, Cell/B.E.
- From cell phones to supercomputers
- Supported by a large range of vendors: Apple, AMD, ARM, Intel, NVIDIA, IBM, Imagination, Huawei, TI, etc.
- OpenCL is inspired by CUDA, but HW & SW vendor neutral

**OpenCL is cross-platform!**

# What is OpenCL?

The BIG idea for cross-platform is that OpenCL defines a unified **platform model**, which is further mapped to the underlying **physical machines**.

- Platform model
  - An OpenCL platform has one host and one or more compute devices
  - Memory divided into host memory and device memory
- Device model & memory model
  - Each compute unit is divided into one or more processing elements
  - A hirarchical memory system
- Execution model
  - A host program defines the contexts for the kernels and manages their execution, and it run on the host
  - The computational task is coded into *kernel* functions and runs on devices

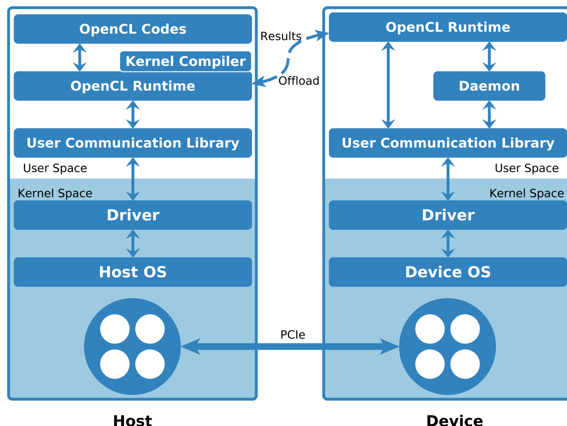


# Why Bother?

- OpenCL for the Feiteng CPUs
  - The host CPU features more computational and memory resources
  - Blurred boundaries between host and accelerating devices
  - Have to care about a large amount of legacy codes in OpenCL
- Engineering requirements
  - Heterogeneous nodes in Tianhe-2A
  - Replacing Xeon Phi with Matrix2000
  - Building block: Xeon CPUs + Matrix2000

# Our OpenCL Design

- The software stack and the OpenCL module
  - The host side and the device side
  - The FT CPU-only mode (✓)
  - The Xeon-Matrix2000 mode (✓)



# Kernel Compiler

- An OpenCL program has two parts: a host program and kernels
  - e.g., vector addition ($C = A + B$)

The host-side code

```
1  void main(int argc, char** argv){
2    int n = 1024;
3    int size = n * sizeof(float)
4    float * A = (float *)malloc(size);
5    float * B = (float *)malloc(size);
6    float * C = (float *)malloc(size);
7    cl_mem dA = clMalloc(size);
8    cl_mem dB = clMalloc(size);
9    cl_mem dC = clMalloc(size);
10   clH2D(dA, A, size);
11   clH2D(dB, B, size);
12   clExKernel("vector_add",dA,dB,dC,n);
13   clD2H(C, dC, size);
14   clFree(dA); clFree(dB); clFree(dC);
15   free(A); free(B); free(C);
16 }
```

(a) Host

An OpenCL kernel

```
1  /*
2    vector addition: C = A + B
3  */
4  __kernel
5  void vector_add(__global float* A,
6                  __global float* B,
7                  __global float* C,
8                  int n){
9    int gid = get_global_id(0);
10   if(gid < n){
11     float a = A[gid];
12     float b = B[gid];
13     float c = a + b;
14     C[gid] = c;
15   }
16 }
```

(b) Device

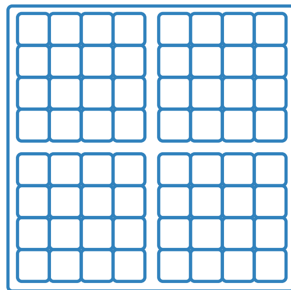
# Kernel Compiler

- OpenCL execution model (continued)
  - Work-item: a kernel instance
  - Work-group: a group of work-items
  - NDRange: index space

An OpenCL kernel

```
1  /*
2    vector addition: C = A + B
3  */
4  __kernel
5  void vector_add(__global float* A,
6                  __global float* B,
7                  __global float* C,
8                  int n){
9    int gid = get_global_id(0);
10   if(gid < n){
11     float a = A[gid];
12     float b = B[gid];
13     float c = a + b;
14     C[gid] = c;
15   }
16 }
```

(c) Device



(d) Index space

# Kernel Compiler

- Our kernel compiler
  - Aims to compile kernels into device-specific binaries
  - The basic idea is to transform work-item-functions (WIF) into work-item-loops (WIL), which are then distributed to threads

**An OpenCL kernel**

```
1  /*
2    vector addition: C = A + B
3  */
4  __kernel
5  void vector_add(__global float* A,
6                  __global float* B,
7                  __global float* C,
8                  int n){
9    int gid = get_global_id(0);
10   if(gid < n){
11     float a = A[gid];
12     float b = B[gid];
13     float c = a + b;
14     C[gid] = c;
15   }
16 }
```

(e) WIF

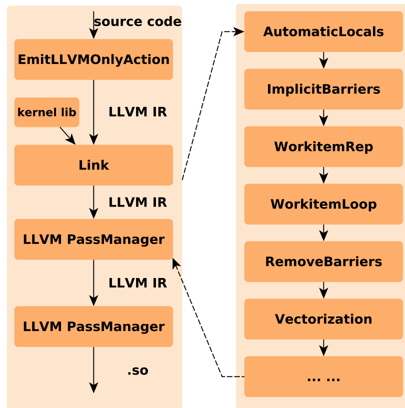
**An OpenCL kernel (WIL)**

```
1  __kernel
2  void vector_add(__global float* A,
3                  __global float* B,
4                  __global float* C
5          int n, int start, int end){
6    int gid = 0 ; //= get_global_id(0);
7    for(int gid=start; gid<end; gid++){
8      if(gid < n){
9        float a = A[gid];
10       float b = B[gid];
11       float c = a + b;
12       C[gid] = c;
13     }
14   }
15 }
```

(f) WIL

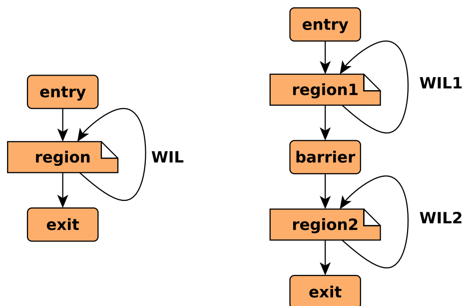
# Kernel Compiler

- Our kernel compiler
  - Aims to compile kernels into device-specific binaries
  - The basic idea is to transform work-item-functions (WIF) into work-item-loops (WIL), which are then distributed to threads
  - Relies on the LLVM compilation infratructure



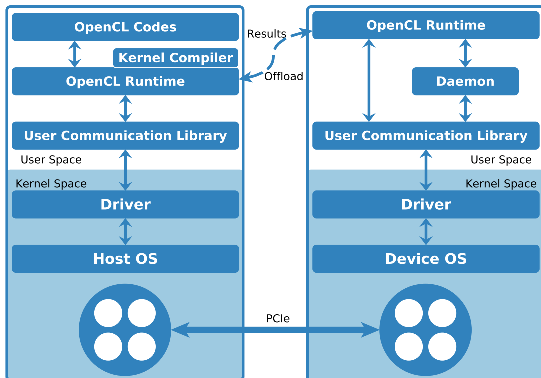
# Kernel Compiler

- Dealing with barriers
  - Barrier is used to synchronize the work-items within a work-group
  - We have to respect the synchronization sematics of OpenCL kernels
  - Our solution: partition code into regions at the barriering locations
  - Handling conditional barriers is more complicated



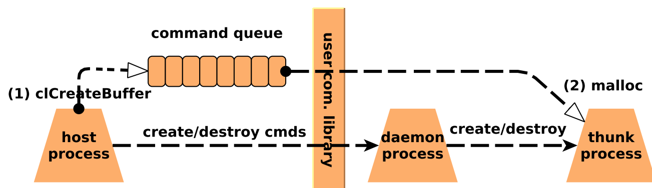
# Runtime System

- Our runtime system
  - Implements the OpenCL APIs and generates `libOpenCL.so`
    - e.g., `gcc vector_add.c -o vector_add -lOpenCL`
    - invokes the kernel compiler in `clBuildProgram`
  - Uses a daemon process on the device side for the first-time communication and device-side process management purpose



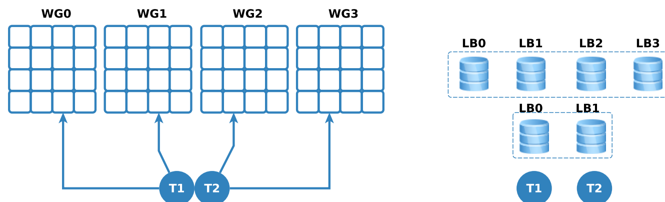
# Runtime System

- The host-side runtime
  - The host-device interaction is performed in terms of **commands**
    - Buffer allocation/deallocation
    - Data movement from host to device and vice versa
    - Kernel compilation/execution
    - Synchronization
  - We use a thunk process on the device side
    - To manage the device-side runtime
    - Created during the initialization stage
    - Destroyed when finializing the OpenCL program
  - The interaction relies on the **user communication library**



# Runtime System

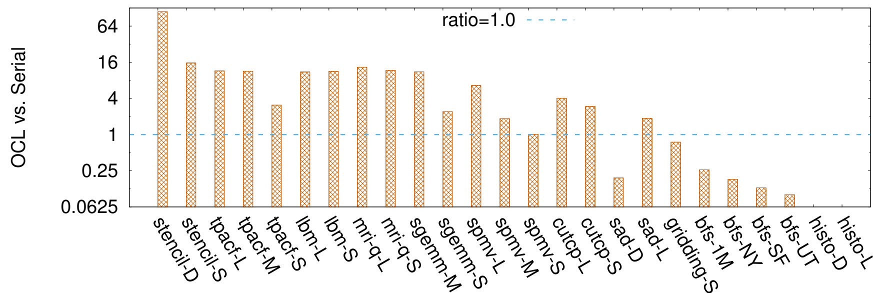
- The device-side runtime
  - Dispatching tasks
    - The computation from a work-group is referred to be as a **task**
    - The tasks are distributed evenly to hardware threads on the device
    - The thunk process manages a thread-pool (16 or 64) with `pthread`
  - Dealing with local memory
    - Local buffers are allocated at the start of kernel execution and freed when finishing kernel execution
    - They are allocated physically in the global memory space
    - Using a copy per work-group vs. using a copy per thread



# Experimental Setup

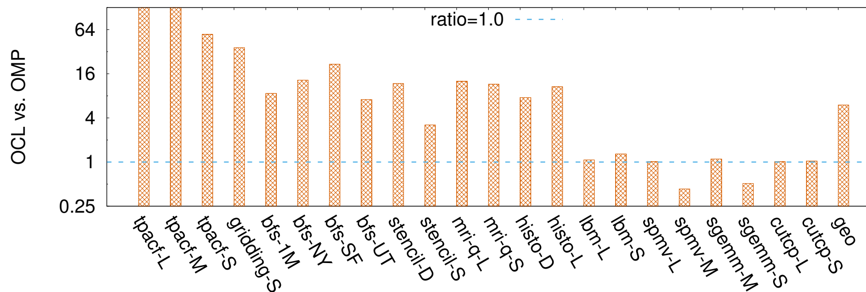
- Our OpenCL is installed on FT-1500A with Linux Kylin `v3.14.0`
- The OpenCL kernel code is compiled with LLVM/Clang `v3.9.0`
- We selected 11 benchmarks from the `Parboil` benchmark suite
  - bfs, stencil, lbm, histo, sgemm, mri-q, cutcp, tpacf, mri-gridding, spmv, and sad
  - Each benchmark has 3 versions: serial, OpenMP, and OpenCL
  - The OpenMP version is compiled in GCC `v6.2.0`
- Each experiment is run 10 times and the medium results are recorded
- We aim to answer
  - How well our OpenCL performs compared to the serial code?
  - How well our OpenCL performs compared to the OpenMP code?

# Experimental Results: OpenCL VS. Serial




- For stentil, tpacf, lbm, mri-q, sgemm, spmv, cutcp, sad, OpenCL performs better than Serial
  - The speedup for most apps in OpenCL is less than 16
  - Stentil in OCL runs particular fast due to efficient memory accesses
- For mri-gridding, bfs, histo, OpenCL performs worse
  - Coalesced memory accesses is perferred by GPUs, but not FT CPUs
  - Staging data in local memory is not preferred on FT CPUs

# Experimental Results: OpenCL VS. OpenMP




- The OCL code performs, on average, 6X as fast as the OMP code
- For bfs, tpacf, mri-gridding, and histo, OCL runs faster than OMP
  - OMP uses *critical* which is avoided by OCL
- For mri-q, OCL uses customized built-in functions, e.g., *sin*, *cos*
- for stencil, OCL can gaurantee contiguous memory access per thread

# Take-Home Message

- OpenCL is an open programming interface for modern many-cores, and its implementations on domestic processors are desired
- Our OpenCL implementation includes a **kernel compiler** and a **runtime**. The kernel compile resists on LLVM/Clang while the runtime relies on the user communication library
- Although very straightforward techniques are used our OpenCL implementation yields a speedup of 6×, compared to that of OpenMP
- The kernels optimized for GPUs are often unsuitable for FT-like CPUs
- We will further explore the design space
    - Fine-grained vectorization for Xiaomi SIMD cores
    - Using more complex task dispatching techniques
    - In support of more data-moving modes

# The End