基于Mali midgard架构的异构计算

软件三部　　张锦涛

Agenda

- 异构计算(HSA)
- Open Computing Language
- Mali midgard架构介绍
- OpenCL execution model on Mali
- Optimal OpenCL for Mali
- OpenCL Optimization Case Studies
- Developing tools

Heterogeneous computing

Wiki: Heterogeneous computing refers to systems that use more than one kind of processor or cores. These systems gain performance or energy efficiency not just by adding the same type of processors, but by adding dissimilar coprocessors, usually incorporating specialized processing capabilities to handle particular tasks.

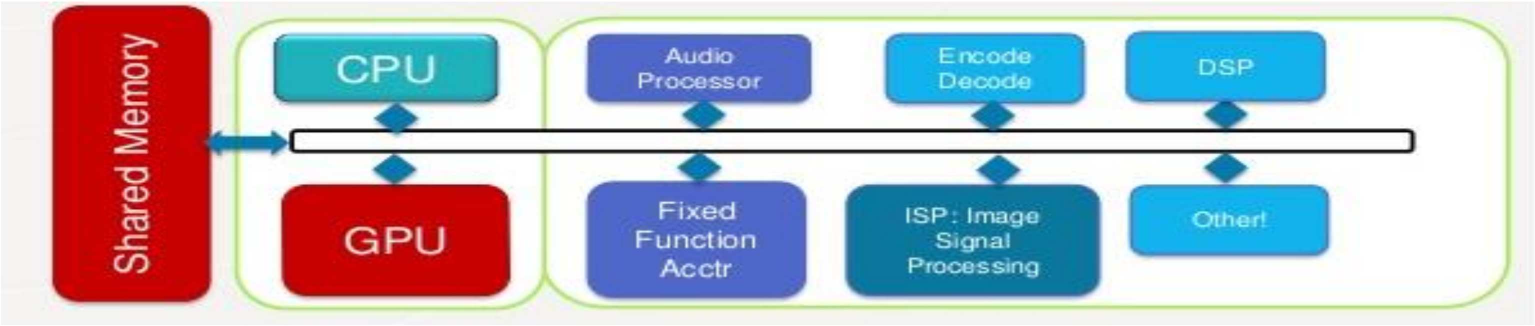


Fig 1. An efficient Heterogeneous System Architecture

Trends in the Industry

Heterogeneous multiprocessing
Established approach for SoC design
Mix of many specialized accelerators, implementing different ISAs
Diverse programming approaches lead to lack of portability(e.g. CUDA/HDL)

Parallel computation for performance and efficiency
Endorsed at all levels of computer architecture
Parallel programming traditionally difficult

General purpose programmability of GPUs
Massive parallel computation potential
Increasing programmability

What is Parallel Computing?

Simply, doing multiple tasks simultaneously

Task-Parallel computing does different tasks concurrently
Reading email, playing music, and surfing the web are all separate tasks
In a multicore system, these can execute simultaneously

Data-Parallel computing does the same operation on a collection of data concurrently
Adjusting the contrast of the pixels of an image
Each thread executes the same code but with different data
Classic SIMD (single-instruction, multiple-data)

GPU computing is perfect for data-parallel applications

GPU-based Parallel Computation is Powerful

Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	National Supercomputing Center in Wuxi	Sunway TaihuLight - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway	10,649,600	93,014.60	125,435.90	15,371
2	National Super Computer Center in Guangzhou	Tianhe-2 (MilkyWay-2) - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P	3,120,000	33,862.70	54,902.40	17,808
3	DOE/SC/Oak Ridge National Laboratory	Titan - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x	560,640	17,590.00	27,112.50	8,209
4	DOE/NNSA/LLNL	Sequoia - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom	1,572,864	17,173.20	20,132.70	7,890
5	RIKEN Advanced Institute for Computational Science (AICS)	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect	705,024	10,510.00	11,280.40	12,660
6	DOE/SC/Argonne National Laboratory	Mira - BlueGene/Q, Power BQC 16C 1.60GHz, Custom	786,432	8,586.60	10,066.30	3,945
7	DOE/NNSA/LANL/SNL	Trinity - Cray XC40, Xeon E5-2698v3 16C 2.3GHz, Aries interconnect	301,056	8,100.90	11,078.90	
8	Swiss National Supercomputing Centre (CSCS)	Piz Daint - Cray XC30, Xeon E5-2670 8C 2.600GHz, Aries interconnect , NVIDIA K20x	115,984	6,271.00	7,788.90	2,325
9	HLRS - Höchstleistungsrechenzentrum Stuttgart	Hazel Hen - Cray XC40, Xeon E5-2680v3 12C 2.5GHz, Aries interconnect	185,088	5,640.20	7,403.50	
10	King Abdullah University of Science and Technology	Shaheen II - Cray XC40, Xeon E5-2698v3 16C 2.3GHz, Aries interconnect	196,608	5,537.00	7,235.20	2,834

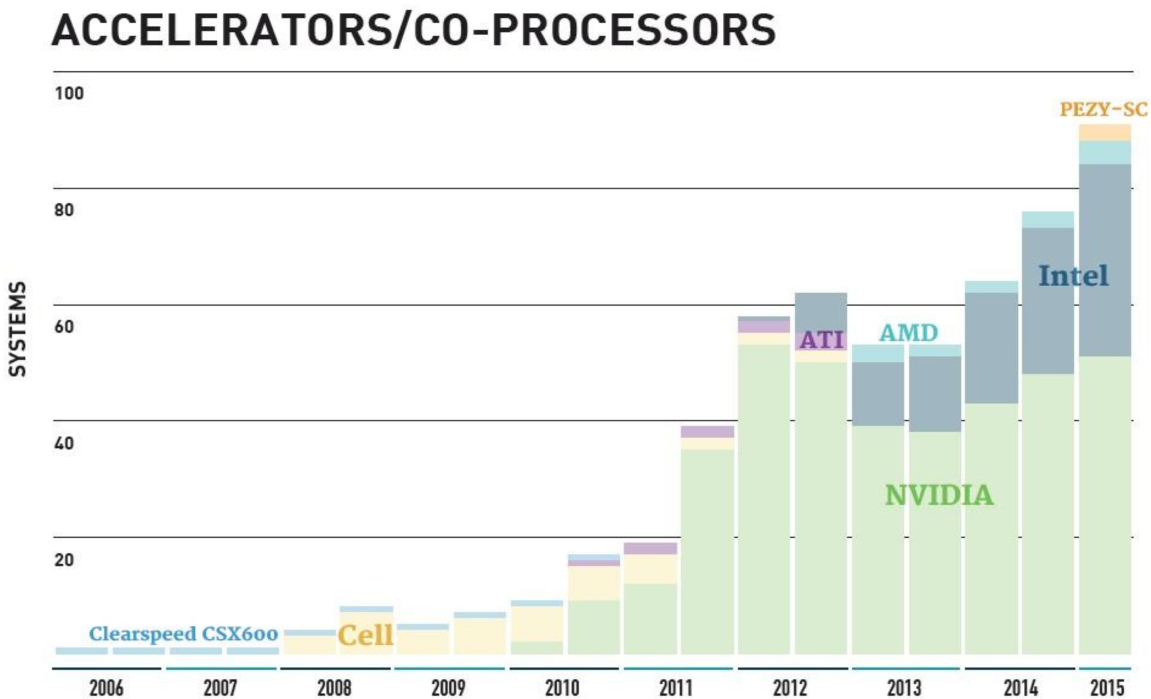
Fig 2. the TOP500 list of the world's top supercomputers (June 2016)

As well as energy-efficient

Green500 Rank	MFLOPS/W	Site	System	Total Power(kW)
1	6673.8	Advanced Center for Computing and Communication, RIKEN	ZettaScaler-1.6, Xeon E5-2618Lv3 8C 2.3GHz, Infiniband FDR, PEZY-SCnp	150
2	6195.2	Computational Astrophysics Laboratory, RIKEN	ZettaScaler-1.6, Xeon E5-2618Lv3 8C 2.3GHz, Infiniband FDR, PEZY-SCnp	46.9
3	6051.3	National Supercomputing Center in Wuxi	Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway	15371
4	5272.1	GSI Helmholtz Center	ASUS ESC4000 FDR/G2S, Intel Xeon E5-2690v2 10C 3GHz, Infiniband FDR, AMD FirePro S9150	57.2
5	4778.5	Institute of Modern Physics (IMP), Chinese Academy of Sciences	Sugon Cluster W780I, Xeon E5-2640v3 8C 2.6GHz, Infiniband QDR, NVIDIA Tesla K80	65
6	4112.1	Stanford Research Computing Center	Cray CS-Storm, Intel Xeon E5-2680v2 10C 2.8GHz, Infiniband FDR, NVIDIA K80	190
7	3775.5	Internet Service (B)	Inspur TS10000 HPC Server, Intel Xeon E5-2620v2 6C 2.1GHz, 10G Ethernet, NVIDIA Tesla K40	110
8	3775.5	Internet Service (B)	Inspur TS10000 HPC Server, Intel Xeon E5-2620v2 6C 2.1GHz, 10G Ethernet, NVIDIA Tesla K40	110
9	3775.5	Internet Service (B)	Inspur TS10000 HPC Server, Intel Xeon E5-2620v2 6C 2.1GHz, 10G Ethernet, NVIDIA Tesla K40	110
10	3775.5	Internet Service (B)	Inspur TS10000 HPC Server, Intel Xeon E5-2620v2 6C 2.1GHz, 10G Ethernet, NVIDIA Tesla K40	110

Fig 3. The Green500's energy-efficient supercomputers (June 2016)

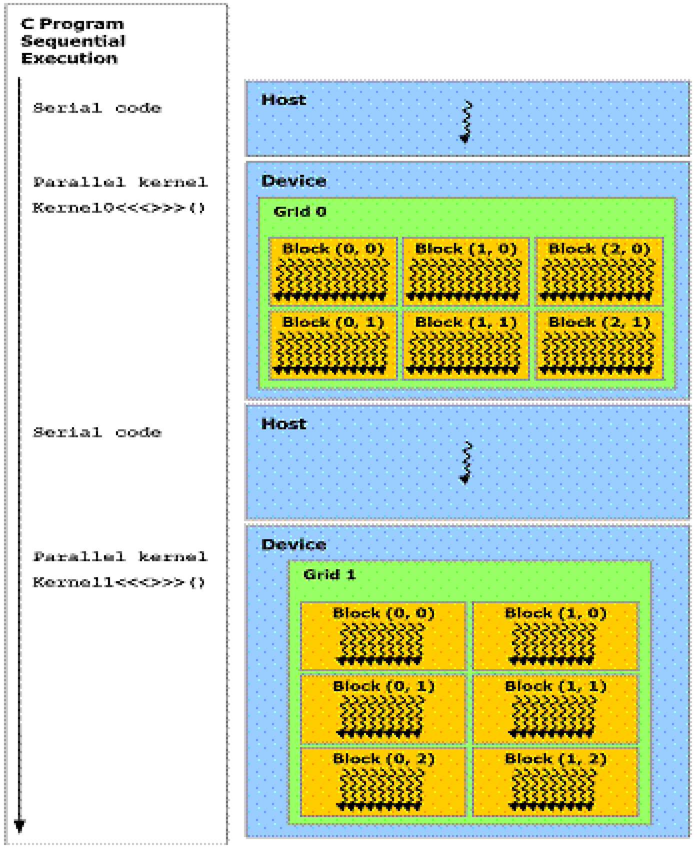
GPUs are the most common accelerators on the list



As of now 88 of the top 500 systems are using accelerators, 52 of which are leveraging NVIDIA GPUs, and the rest with Xeon Phi, although there are also 4 machines using ATI Radeon cores.

Fig 4. Changes on the list from year to year

CPU-GPU co-processing



GPU works as a co-processor within the sub-system, that is, offloading massive data-parallel tasks from CPU.

Fig 5. CPU-GPU co-processing style

In biology, a key idea is that
structure determines function.

The Right Processor for the Task

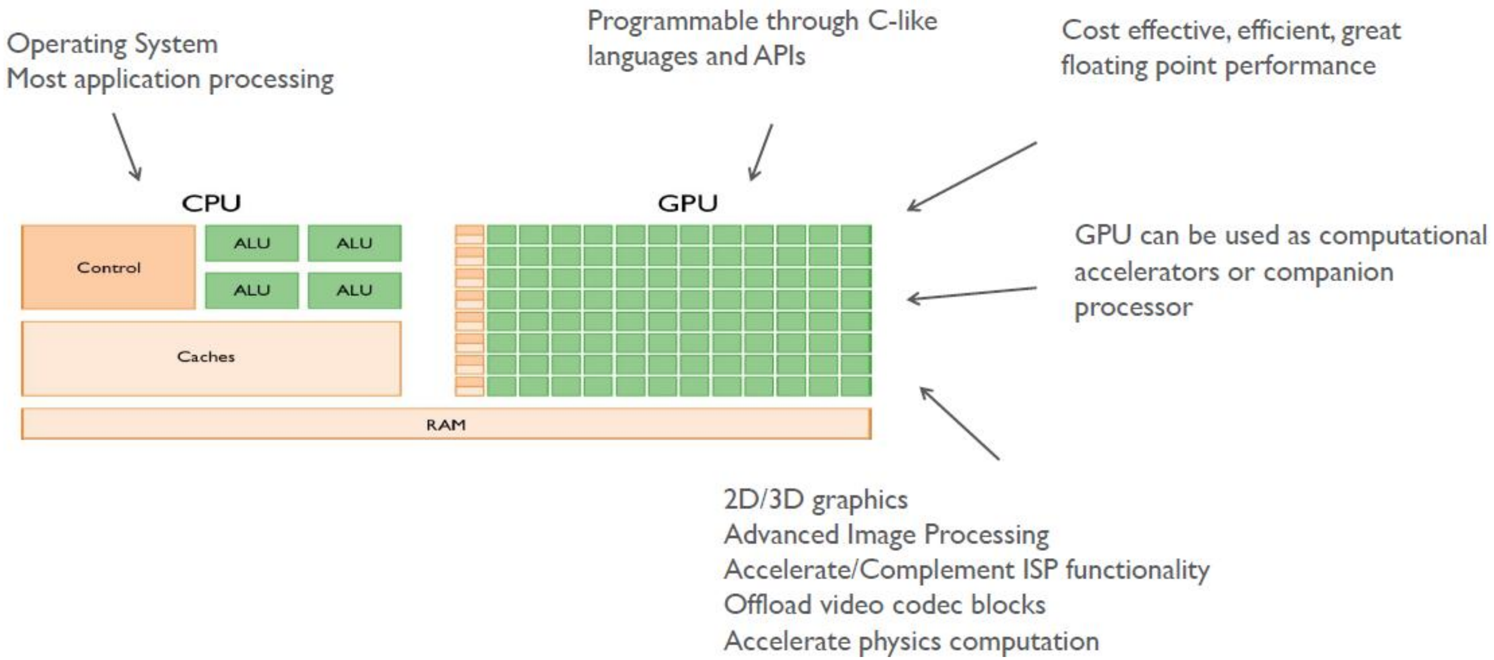
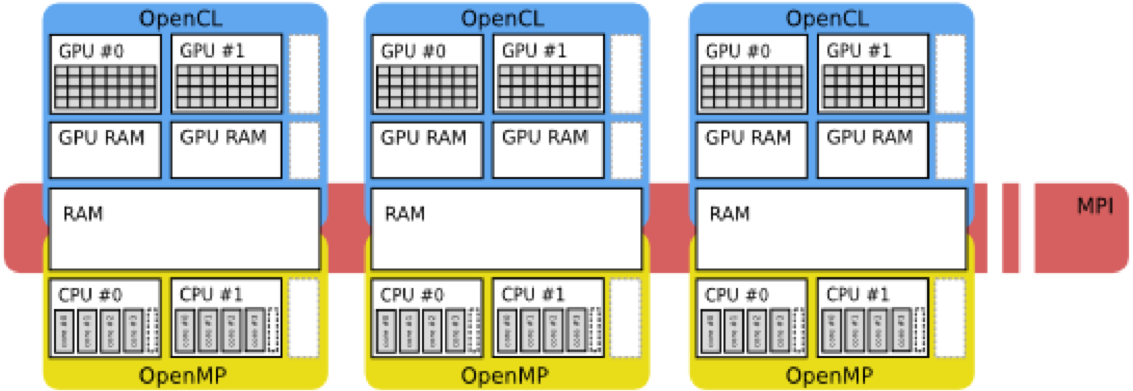


Fig 6. Alternative arch

Supercomputers exploit the multi-core nature of GPUs and CPUs



The co-processor works as a node within a computation network

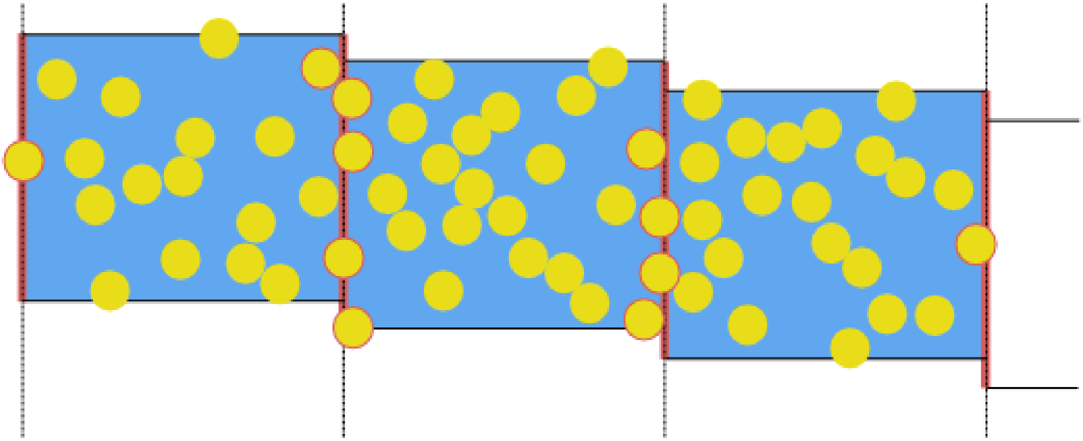


Fig 7. Topological graph of a classical GPU-based server

Performance Evaluation

In computer architecture, Amdahl's law (or Amdahl's argument) gives the theoretical speedup in latency of the execution of a task at fixed workload that can be expected of a system whose resources are improved. Amdahl's law can be formulated the following way:

$$S=1/(1-p+p/n)$$

Agenda

- 异构计算(HSA)
- Open Computing Language
- Mali midgard架构介绍
- OpenCL execution model on Mali
- Optimal OpenCL for Mali
- OpenCL Optimization Case Studies
- Developing tools

What is OpenCL?

- OpenCL™ (Open Computing Language) is the open, royalty-free standard for cross-platform, parallel programming of diverse processors found in personal computers, servers, mobile devices and embedded platforms.

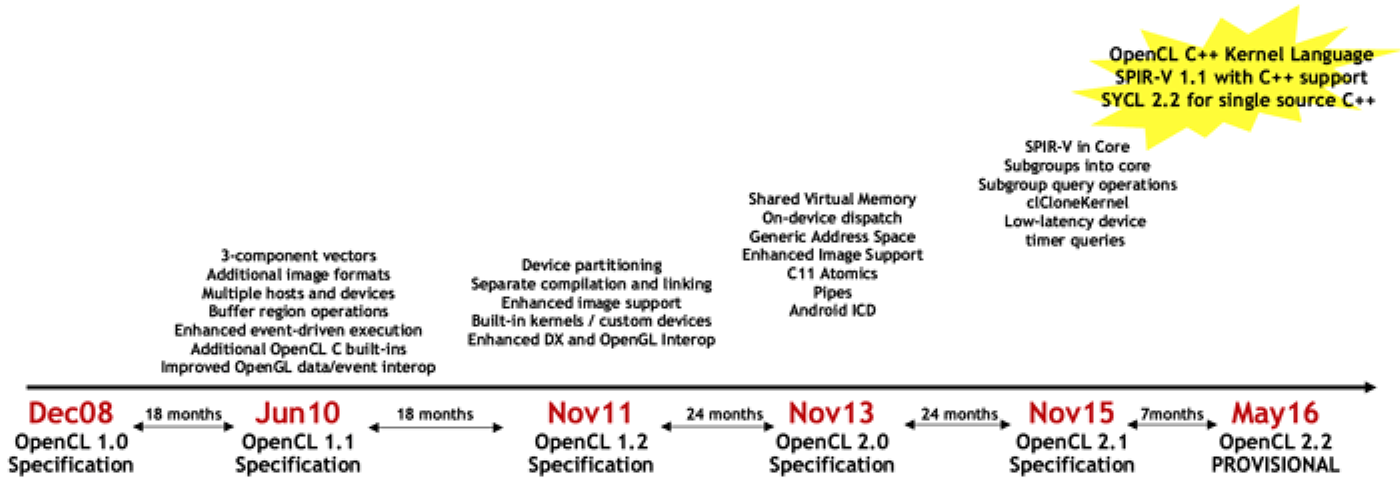


Fig 8. OpenCL Timeline

More on Multi-Platform Targeting

OpenCL

- Targets a broader range of CPU-like and GPU-like devices than CUDA
  - Targets devices produced by multiple vendors
  - Many features of OpenCL are optional and may not be supported on all devices(khronos API feature…)
- OpenCL codes must be prepared to deal with much greater hardware diversity
- A single OpenCL kernel will likely not achieve peak performance on all device types

Data Types in OpenCL

Type in OpenCL Language	API Type for Application
char	cl_char
uchar	cl_uchar
short	cl_short
ushort	cl_ushort
int	cl_int
uint	cl_uint
long	cl_long
ulong	cl_ulong
float	cl_float
double	cl_double
half	cl_half
char *n*	cl_char *n*
uchar *n*	cl_uchar *n*
short *n*	cl_short *n*
ushort *n*	cl_ushort *n*
int *n*	cl_int *n*
uint *n*	cl_uint *n*
long *n*	cl_long *n*
ulong *n*	cl_ulong *n*
float *n*	cl_float *n*
double *n*	cl_double *n*
half *n*	cl_half *n*

Table 1. Built-In Vector Data Types

Vector Components	Usable Numeric Indices
2-component	0, 1
3-component	0, 1, 2
4-component	0, 1, 2, 3
8-component	0, 1, 2, 3, 4, 5, 6, 7
16-component	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, A, b, B, c, C, d, D, e, E, f, F

Table 2. Numeric Indices for Built-In Vector Data Types

Vector Components

Vector Data Types	Accessible Components
char2, uchar2, short2, ushort2, int2, uint2, long2, ulong2, float2	.xy
char3, uchar3, short3, ushort3, int3, uint3, long3, ulong3, float3	.xyz
char4, uchar4, short4, ushort4, int4, uint4, long4, ulong4, float4	.xyzw
double2, half2	.xy
double3, half3	.xyz
double4, half4	.xyzw

Table 3. Accessing Vector Components

Vector access suffix	Returns
.lo	refers to the lower half of a given vector
.hi	refers to the upper half of a given vector
.odd	refers to the odd elements of a vector
.even	refers to the even elements of a vector.

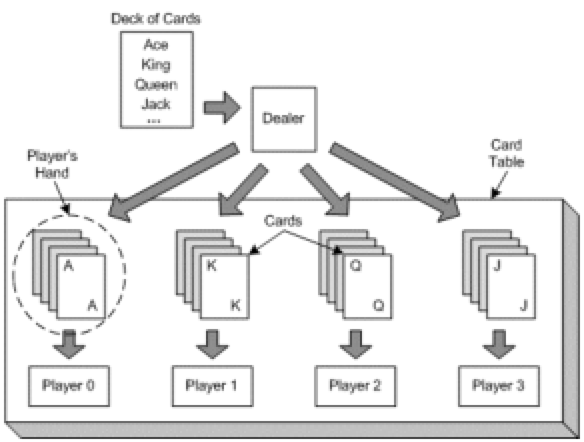
Table 4. Handy addressing of Vector Components

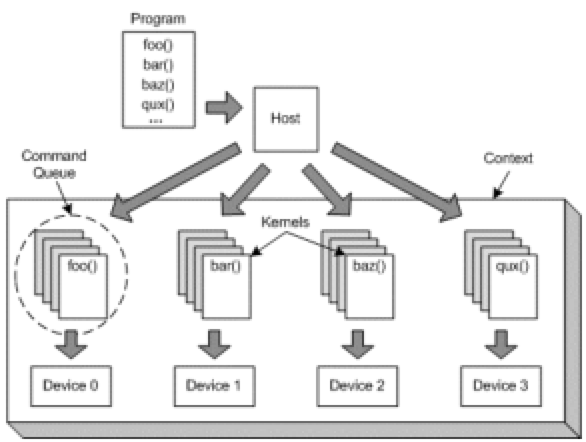
Vector Components

Example

```
float8 vf;
float4 odd = vf.odd;
float4 even = vf.even;
float2 high = vf.even.hi;
float2 low = vf.odd.lo;
// interleave L+R stereo stream
float4 left, right;
float8 interleaved;
interleaved.even = left;
interleaved.odd = right;
// deinterleave
left = interleaved.even;
right = interleaved.odd;
// transpose a 4x4 matrix
void transpose( float4 m[4] )
{
// read matrix into a float16 vector
float16 x = (float16)( m[0], m[1], m[2], m[3] );
float16 t;
//transpose
t.even = x.lo;
t.odd = x.hi;
x.even = t.lo;
x.odd = t.hi;
//write back
m[0] = x.lo.lo; // { m[0][0], m[1][0], m[2][0], m[3][0] }
m[1] = x.lo.hi; // { m[0][1], m[1][1], m[2][1], m[3][1] }
```

OpenCL likes a game of card





OpenCL™ Program Flow

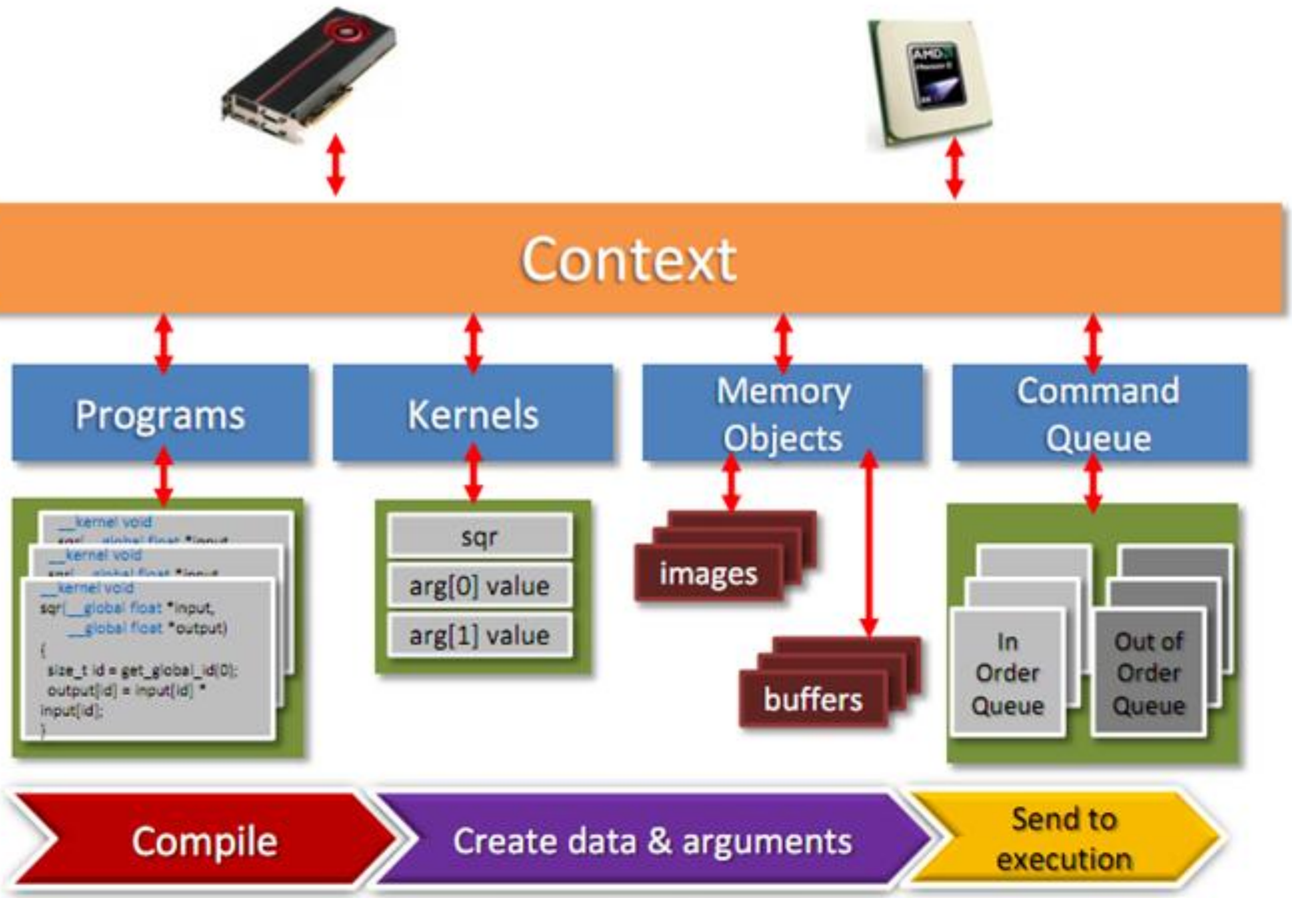


Fig 9. OpenCL Program Flow

OpenCL™ Program Flow

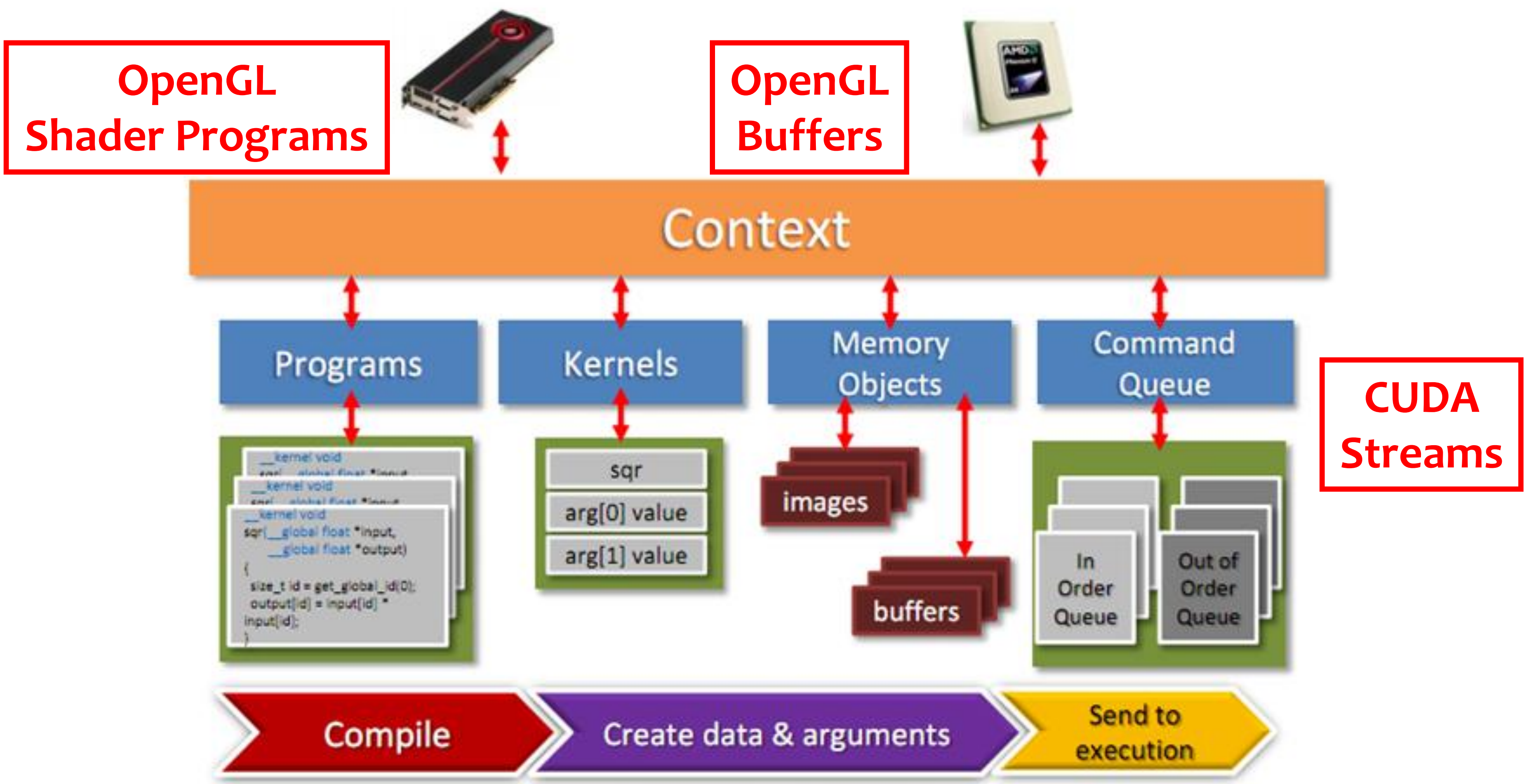


Fig 9. OpenCL Program Flow

OpenCL API

- Walkthrough OpenCL *host* code for running our `vecAdd` kernel:

```
__kernel void vecAdd(__global const
  float *a, __global const float *b,
  __global float *c)
{
   int i = get_global_id(0);
   c[i] = a[i] + b[i];
}
```

OpenCL API


```
// create OpenCL device & context
cl_context hContext;
hContext = clCreateContextFromType(0,
  CL_DEVICE_TYPE_GPU, 0, 0, 0);
```

OpenCL API


```
// create OpenCL device & context
cl_context hContext;
hContext = clCreateContextFromType(0,
  CL_DEVICE_TYPE_GPU, 0, 0, 0);
```

Create a context for a GPU

OpenCL API


```
// query all devices available to the context
size_t nContextDescriptorSize;
clGetContextInfo(hContext, CL_CONTEXT_DEVICES,
  0, 0, &nContextDescriptorSize);
cl_device_id aDevices =
  malloc(nContextDescriptorSize);
clGetContextInfo(hContext, CL_CONTEXT_DEVICES,
  nContextDescriptorSize, aDevices, 0);
```

OpenCL API


```
// query all devices available to the context
size_t nContextDescriptorSize;
clGetContextInfo(hContext, CL_CONTEXT_DEVICES,
  0, 0, &nContextDescriptorSize);
cl_device_id aDevices =
  malloc(nContextDescriptorSize);
clGetContextInfo(hContext, CL_CONTEXT_DEVICES,
  nContextDescriptorSize, aDevices, 0);
```

Retrieve an array of each GPU

OpenCL API


```
// create a command queue for first
// device the context reported
cl_command_queue hCmdQueue;
hCmdQueue =
  clCreateCommandQueue(hContext,
     aDevices[0], 0, 0);
```

OpenCL API

```
// create a command queue for first
// device the context reported
cl_command_queue hCmdQueue;
hCmdQueue =
  clCreateCommandQueue(hContext,
    aDevices[0], 0, 0);
```

Create a command queue (CUDA stream) for the first GPU

OpenCL API


```
// create & compile program
cl_program hProgram;
hProgram =
  clCreateProgramWithSource(hContext,
     1, source, 0, 0);
clBuildProgram(hProgram, 0, 0, 0, 0,
  0);
```

OpenCL API


```
// create & compile program
cl_program hProgram;
hProgram =
  clCreateProgramWithSource(hContext,
     1, source, 0, 0);
clBuildProgram(hProgram, 0, 0, 0, 0,
  0);
```

- A program contains one or more kernels.
- Provide kernel source as a string
- Can also compile offline

OpenCL API


```
// create kernel
cl_kernel hKernel;
hKernel = clCreateKernel(hProgram,
  “vecAdd”, 0);
```

OpenCL API


```
// create kernel
cl_kernel hKernel;
hKernel = clCreateKernel(hProgram,
   "vecAdd", 0);
```

Create kernel from program

OpenCL API


```
// allocate host vectors
float* pA = new float[cnDimension];
float* pB = new float[cnDimension];
float* pC = new float[cnDimension];

// initialize host memory
randomInit(pA, cnDimension);
randomInit(pB, cnDimension);
```

OpenCL API


```
cl_mem hDeviceMemA = clCreateBuffer(
   hContext,
   CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
   cnDimension * sizeof(cl_float),
  pA,  0);

cl_mem hDeviceMemB = /* ... */
```

OpenCL API


```
cl_mem hDeviceMemA = clCreateBuffer(
   hContext,
   CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
   cnDimension * sizeof(cl_float),
  pA,   0);

cl_mem hDeviceMemB = /* ... */
```

Create buffers for kernel input. Read only in the kernel. Written by the host.

OpenCL API


```
hDeviceMemC = clCreateBuffer(hContext,
  CL_MEM_WRITE_ONLY,
   cnDimension * sizeof(cl_float),
   0, 0);
```

OpenCL API


```
hDeviceMemC = clCreateBuffer(hContext,
  CL_MEM_WRITE_ONLY,
   cnDimension * sizeof(cl_float),
   0, 0);
```

Create buffer for kernel output.

OpenCL API


```
// setup parameter values
clSetKernelArg(hKernel, 0,
  sizeof(cl_mem), (void
  *)&hDeviceMemA);
clSetKernelArg(hKernel, 1,
  sizeof(cl_mem), (void
  *)&hDeviceMemB);
clSetKernelArg(hKernel, 2,
  sizeof(cl_mem), (void
  *)&hDeviceMemC);
```

OpenCL API


```
// setup parameter values
clSetKernelArg(hKernel, 0,
    sizeof(cl_mem), (void
    *)&hDeviceMemA);
clSetKernelArg(hKernel, 1,
    sizeof(cl_mem), (void
    *)&hDeviceMemB);
clSetKernelArg(hKernel, 2,
    sizeof(cl_mem), (void
    *)&hDeviceMemC);
```

Kernel arguments set by index

OpenCL API


```
// execute kernel
clEnqueueNDRangeKernel(hCmdQueue,
  hKernel, 1, 0, &cnDimension, 0, 0, 0,
  0);
// copy results from device back to host
clEnqueueReadBuffer(hContext,
  hDeviceMemC, CL_TRUE, 0,
   cnDimension * sizeof(cl_float),
   pC, 0, 0, 0);
```

OpenCL API

Let OpenCL pick work group size

```
// execute kernel
clEnqueueNDRangeKernel(hCmdQueue,
  hKernel, 1, 0, &cnDimension, 0, 0, 0,
  0);
// copy results from device back to host
clEnqueueReadBuffer(hContext,
  hDeviceMemC, CL_TRUE, 0,
  cnDimension * sizeof(cl_float),
  pC, 0, 0, 0);
```

Blocking read

OpenCL API


```
delete [] pA;
delete [] pB;
delete [] pC;
clReleaseMemObj(hDeviceMemA);
clReleaseMemObj(hDeviceMemB);
clReleaseMemObj(hDeviceMemC);
```

LEADCORE 联芯科技

Agenda

- 异构计算(HSA)
- Open Computing Language
- Mali midgard架构介绍
- OpenCL execution model on Mali
- Optimal OpenCL for Mali
- OpenCL Optimization Case Studies
- Developing tools

- Mali GPU Roadmap
- Midgard arch

Midgard-Roadmap

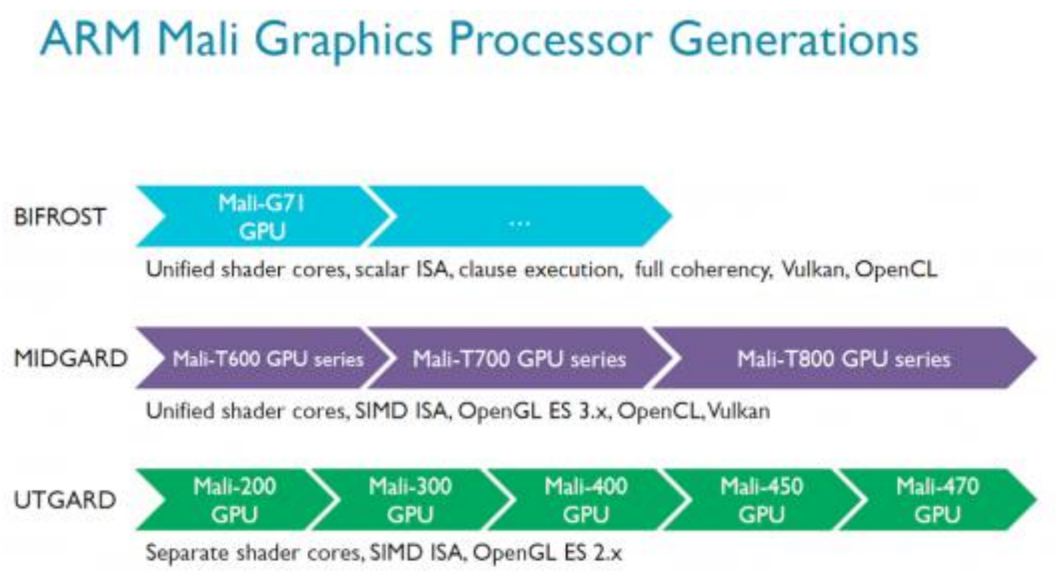


Fig 10. Mali GPU roadmap

Model	Architecture
ARM Mali-G71	Bifrost
ARM Mali-T880 MP12	Midgard (4th-gen)
ARM Mali-T860 MP2	Midgard (4th-gen)
ARM Mali-T830 MP2	Midgard (4th-gen)
ARM Mali-T760 MP8	Midgard (3rd-gen)
ARM Mali-T720 MP4	Midgard (3rd-gen)
ARM Mali-T628 MP6	Midgard (2nd Gen)
ARM Mali-T624 MP4	Midgard (2nd Gen)
ARM Mali-T604 MP4	Midgard (1st Gen)
ARM Mali-450 MP4	Utgard
ARM Mali-400 MP4	Utgard
ARM Mali-200	Utgard

Table 5.Variants

Note: specific configuration various from different HW version

- Mali GPU Roadmap
- Midgard arch

The "Midgard" family of Mali GPUs (the Mali-T600/700/800 series) use a unified shader core architecture, meaning that only a single type of shader core exists in the design. This single core can execute all types of programmable shader code, including vertex shaders, fragment shaders, and compute kernels.

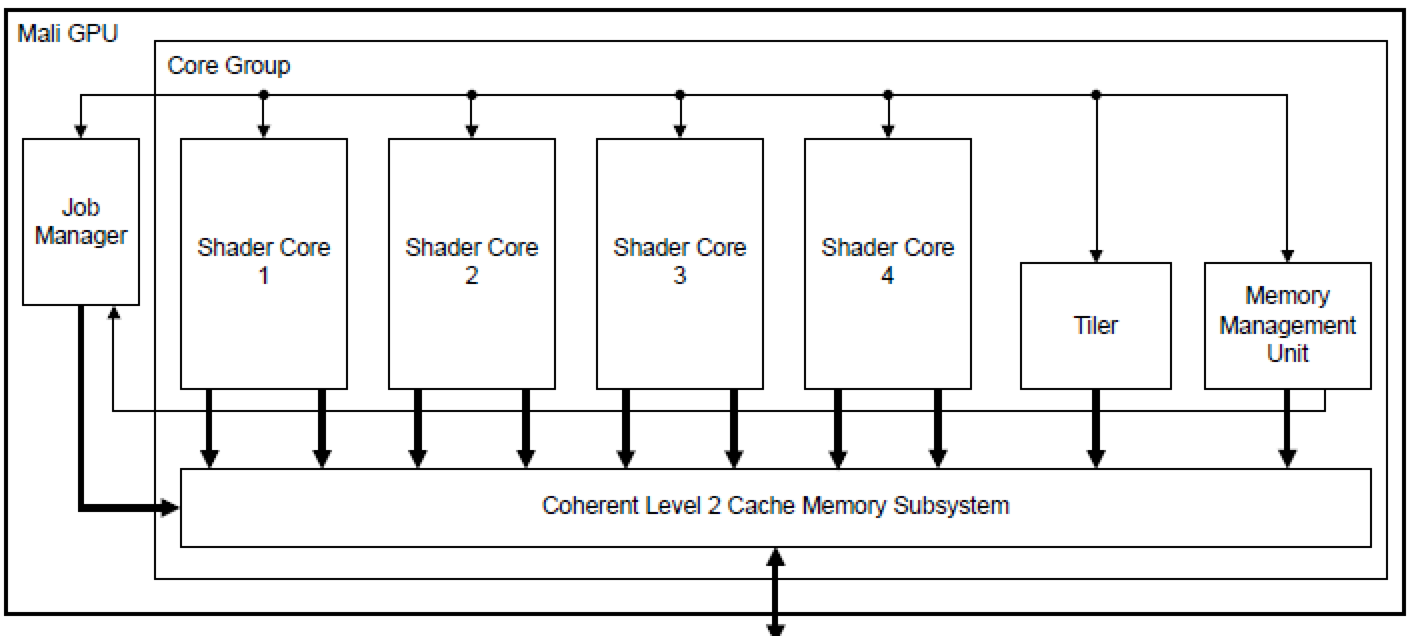


Fig 11. Mali GPU configuration

The shader core combines geometry and pixel processing into a single shader core, improving utilization and removing latency and bottlenecks.

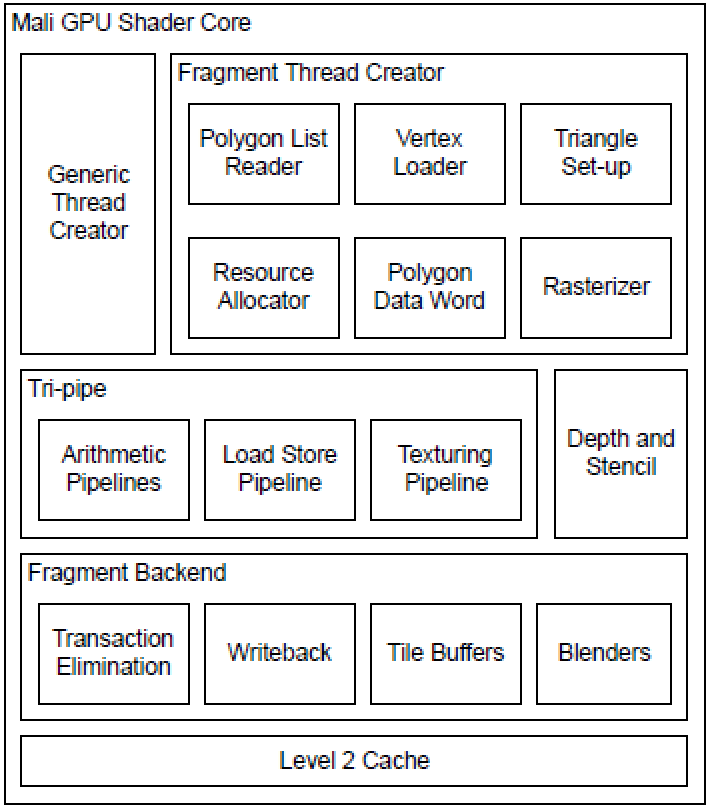


Fig 12. Mali Shader core components

Innovative GPU architecture

Tri-pipe – designed for performance and flexibility (only small parts are graphics only)

Functional units – designed with requirements of general computing

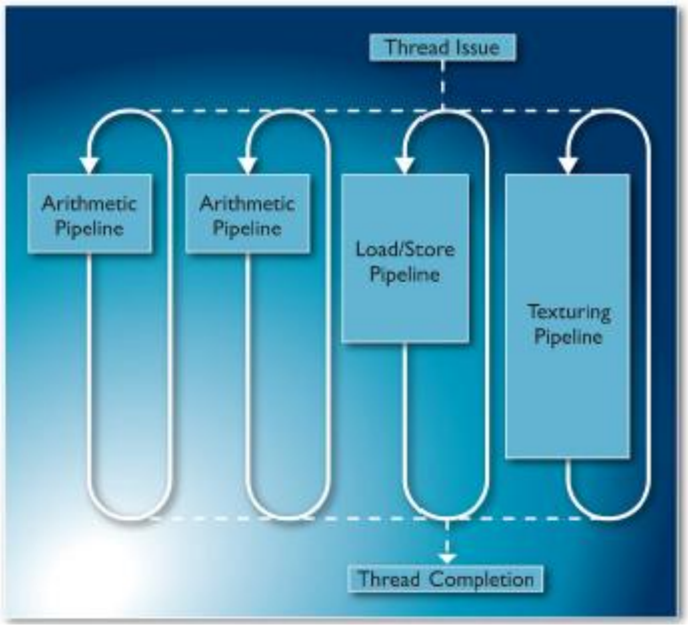


Fig 13. Mali Shader core components

Agenda

- 异构计算(HSA)
- Open Computing Language
- Mali midgard架构介绍
- OpenCL execution model on Mali
- Optimal OpenCL for Mali
- OpenCL Optimization Case Studies
- Developing tools

CL Execution model on Mali

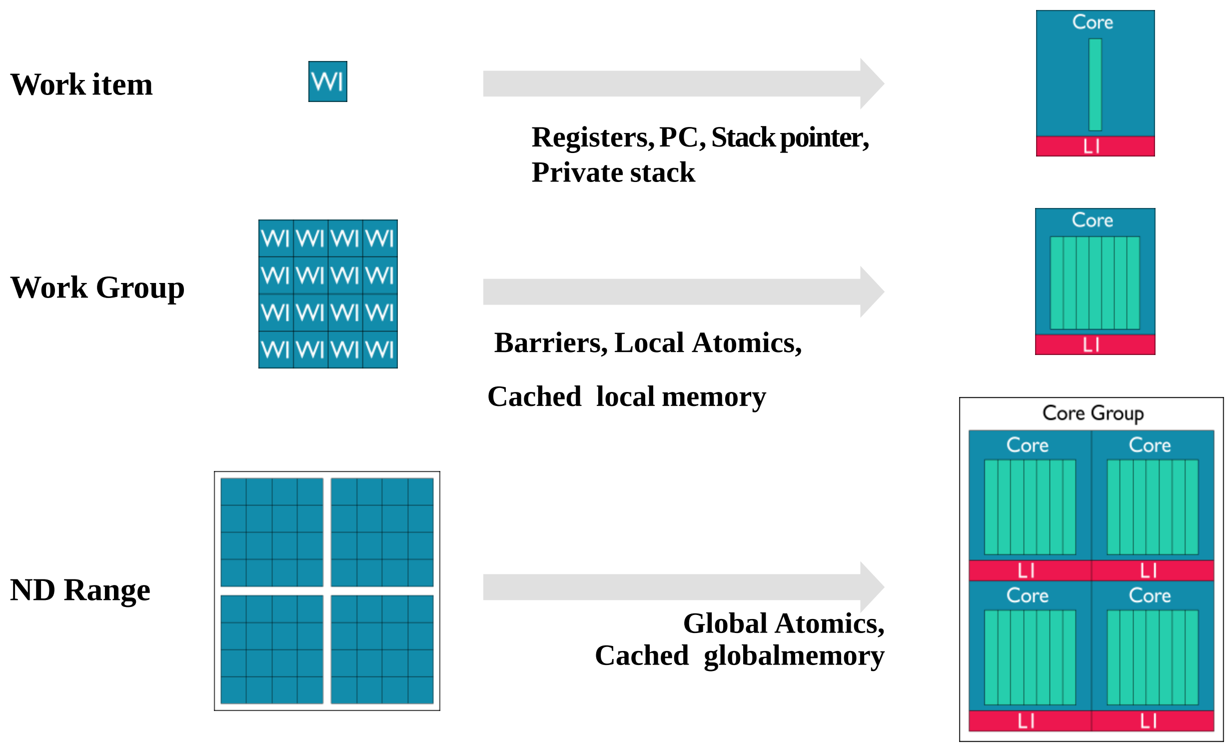


Fig 14. OpenCL Execution model on Mali

CL Execution model on Mali

- **Each work-item runs as one of the threads within a core**
  - **Every Mali-T600 thread has its own independent program counter**
  - **…which supports divergent threads from the same kernel**
  - **caused by conditional execution, variable length loops etc.**
  - **Some other GPGPU's use "WARP" architectures**
  - **These share a common program counter with a group of work-items**
  - **This can be highly scalable… but can be slow handling divergent threads**
  - **T600 effectively has a Warp size of 1**
  - **Up to 256 threads per core**
  - **Every thread has its own registers(NV CUDA registers)**
  - **Every thread has its own stack pointer and private stack**
  - **Shared read-only registers are used for kernel arguments**

CL Execution model on Mali

- A whole work-group executes on a single core
- Mali-T600 supports up to 256 work-items per work-group
- OpenCL barrier operations (which synchronise threads) are handled by the hardware
- For full efficiency you need more work-groups than cores
- To keep all of the cores fed with work
- Most GPUs require this, so most CL applications will do this
- Local and global atomic operations are available in hardware
- All memory is cached

Inside a Core

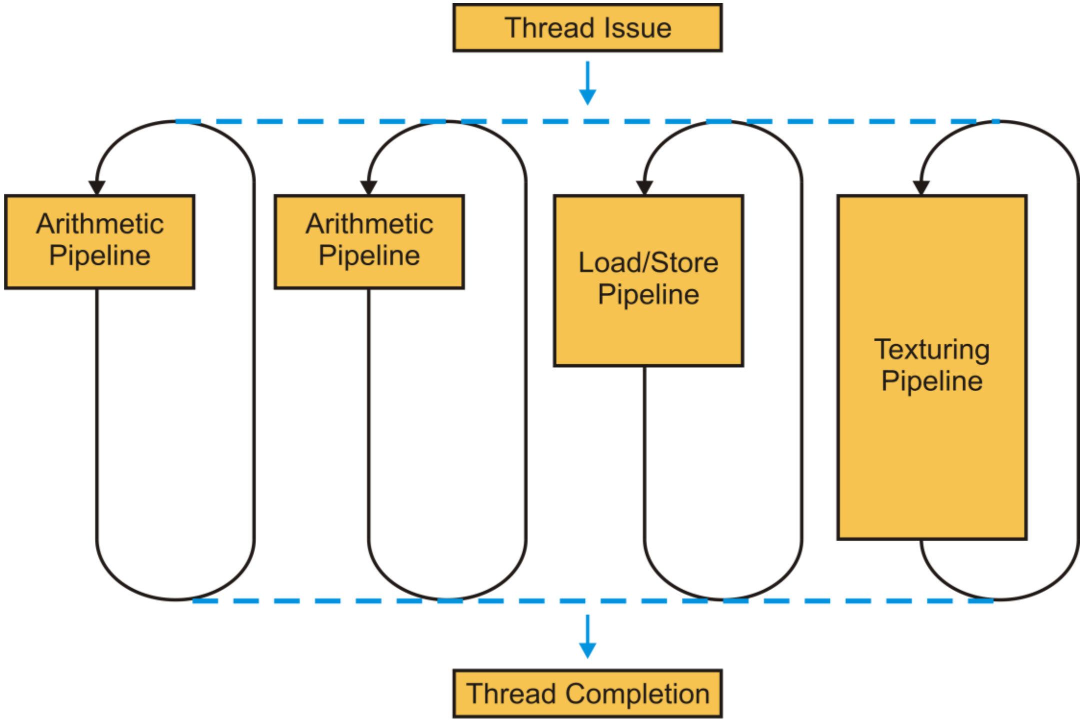


Fig 15. OpenCL Execution inside shader core

Agenda

- 异构计算(HSA)
- Open Computing Language
- Mali midgard架构介绍
- OpenCL execution model on Mali
- Optimal OpenCL for Mali
- OpenCL Optimization Case Studies
- Developing tools

- Programming Suggestions
- Optimising with DS-5 Streamline
- Optimising: Two Examples
- General Advice

Porting OpenCL code from other GPUs

- **Desktop GPUs require data to be copied to local or private memory buffers**
  - **Otherwise their performance suffers**
  - **These copy operations are expensive**
  - **These are sometimes done in the first part of a kernel, followed by a synchronisation barrier instruction, before the actual processing begins in the second half**
  - **The barrier instruction is also expensive**

- **When running on Mali just use global memory instead**
  - **Thus the copy operations can be removed**
  - **And also any barrier instructions that wait for the copy to finish**
  - **Query the device flag CL_DEVICE_HOST_UNIFIED_MEMORY if you want to write performance portable code for Mali and desktop PC's**
  - **The application can then switch whether or not it performs copying to local memory**

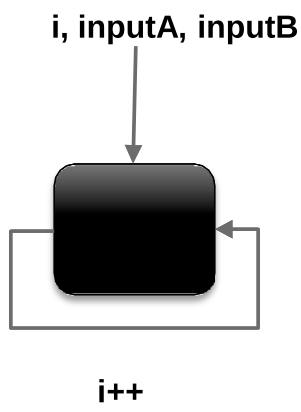
UseVectors

- Mali-T600 and T700 series GPUs have a vector capable GPU
- Mali prefers explicit vector functions
- clGetDeviceInfo

  CL_DEVICE_NATIVE_VECTOR_WIDTH_CHAR
  CL_DEVICE_NATIVE_VECTOR_WIDTH_SHORT
  CL_DEVICE_NATIVE_VECTOR_WIDTH_INT
  CL_DEVICE_NATIVE_VECTOR_WIDTH_LONG
  CL_DEVICE_NATIVE_VECTOR_WIDTH_FLOAT
  CL_DEVICE_NATIVE_VECTOR_WIDTH_DOUBLE
  CL_DEVICE_NATIVE_VECTOR_WIDTH_HALF

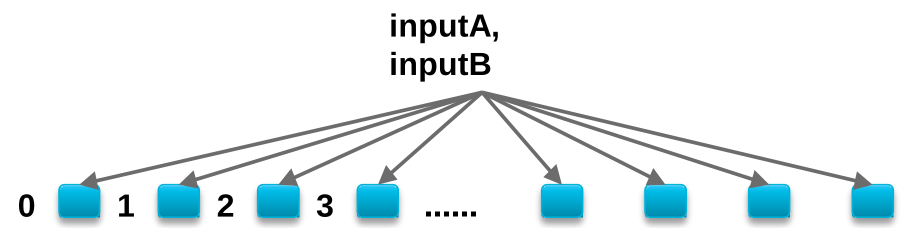
Hello OpenCL

```
for (int i = 0; i < arraySize; i++)
{
    output[i] = inputA[i] + inputB[i];
}
```



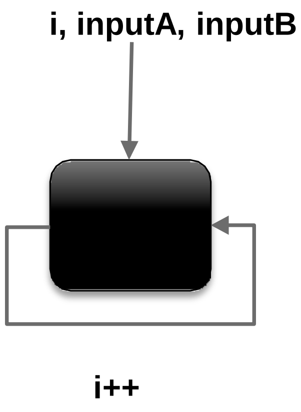

```
__kernel void kernel_name(  global int* inputA,
                           __global int* inputB,
                           __global int* output)
{
        int  i = get_global_id(0);
        output[i] = inputA[i] + inputB[i];
}

clEnqueueNDRangeKernel(..., kernel, ..., arraySize, ...)
```



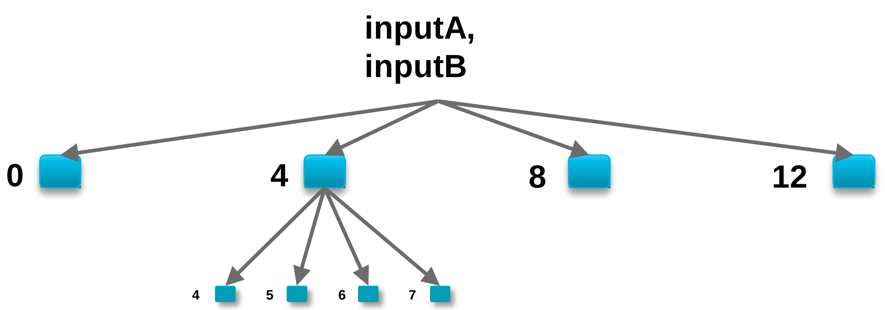
Hello OpenCL Vectors

```
for (int i = 0; i < arraySize; i++)
{
    output[i] = inputA[i] + inputB[i];
}
```




```
__kernel void kernel_name(  global int* inputA,
__global int* inputB,
__global int* output)
{
        int    i = get_global_id(0);
        int4 a = vload4(i, inputA);
        int4 b = vload4(i, inputB);
        vstore4(a + b, i, output);
}

clEnqueueNDRangeKernel(..., kernel, ..., arraySize / 4, ...)
```



Creating buffers

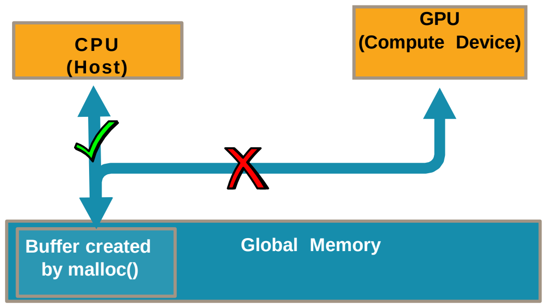
- The application creates buffer objects that pass data to and from the kernels by calling the OpenCL API clCreateBuffer()

- All CL memory buffers are allocated in global memory that is physically accessible by both CPU and GPU cores
  - However, only memory that is allocated by clCreateBuffer is mapped into both the CPU and GPU virtual memory spaces
  - Memory allocated using malloc(), etc, is only mapped onto the CPU

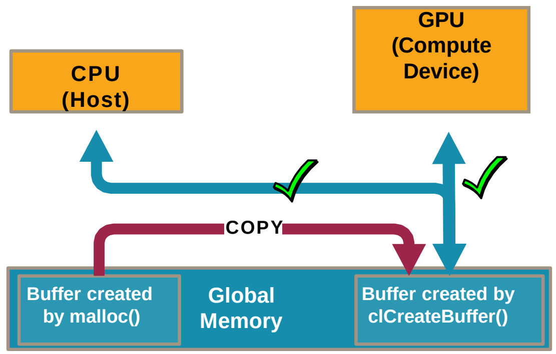
- So calling clCreateBuffer() with CL_MEM_USE_HOST_PTR and passing in a user created buffer requires the driver to create a new buffer and copy the data (identical to CL_MEM_COPY_HOST_PTR)
  - This copy reduces performance

- So where possible always use CL_MEM_ALLOC_HOST_PTR
  - This allocates memory that both CPU and GPU can use without a copy

Host data pointers



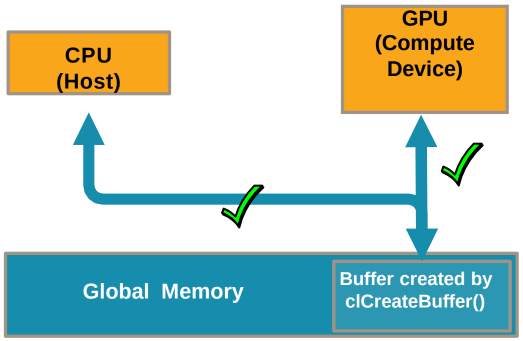
Buffers created by user (malloc) are not mapped into the GPU memory space



clCreateBuffer(CL_MEM_USE_HOST_PTR) creates a new buffer and copies the data over (but the copy operations are expensive)

Fig 16. Memory buffer created by clCreateBuffer(CL_MEM_USE_HOST_PTR)

Host data pointers



clCreateBuffer(CL_MEM_ALLOC_HOST_PTR)
creates a buffer visible by both GPU and CPU

- **Where possible don't use CL_MEM_USE_HOST_PTR**
  - **Create buffers at the start of your application**
  - **Use CL_MEM_ALLOC_HOST_PTR instead of malloc()**
  - **Then you can use the buffer on both CPU host and GPU**

Fig 17. Memory buffer created by clCreateBuffer(CL_MEM_ALLOC_HOST_PTR)

Run Time

- **Where your kernel has no preference for work-group size, for maximum performance.**
  - **either use the compiler recommended work-group size...**
    clGetKernelWorkgroupInfo(kernel, dev, CL_KERNEL_WORK_GROUP_SIZE, sizeof(size_t)...);
  - **or use a large multiple of 4**
  - **You can pass NULL, but performance might not be optimal**
- **If you want your kernel to access host memory**
  - **use mapping operations in place of read and write operations**
  - **mapping operations do not require copies so are faster and use less memory**

Compiler

- **Run-time compilation isn't free! Compile each kernel only once if possible**
  - **If your kernel source is fixed, then compile the kernel during your application's initialisation**
  - **If your application has an installation phase then cache the binary on a storage device for the application's next invocation**
  - **Keep the resultant binary ready for when you want to run the kernel**

- **clBuildProgram only partially builds the source code**
  - **If the kernels in use are known at initialization time, then also call clCreateKernel for each kernel to initiate the finalizing compile**
  - **Creating the same kernels in the future will then be faster because the finalized binary is used**

BIFLs

- Where possible use the built-in functions as the commonly occurring ones compile to fast hardware instructions

- Many will target vector versions of the instructions where available

- Using “half” or “native” versions of built-in functions
- e.g. half_sin(x)
- Specification mandates a minimum of 10-bits of accuracy
- e.g. native_sin(x)
- Accuracy and input range implementation defined

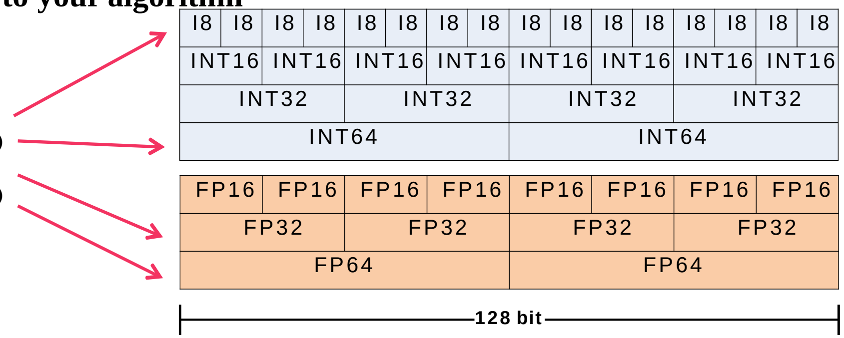
- Not always an advantage on Mali-T600 / T700… for some functions the precise versions are just as fast

Arithmetic

- **Mali-T600 / T700 has a register and ALU width of 128-bits**
  - **Avoid writing kernels that operate on single bytes or scalar values**
  - **Write kernels that work on vectors of at least 128-bits.**
  - **Smaller data types are quicker**
  - **you can fit eight shorts into 128-bits compared to four integers**

- **Integers and floating point are supported equally quickly**
  - **Don’t be afraid to use the data type best suited to your algorithm**
- **Mali-T600 / T700 can natively support all CL data types**

```
16 x 8---bit chars (char16)
2 x 64---bit integers (long2)
4 x 32---bit floats (float4)
2 x 64---bit floats (double2)
```

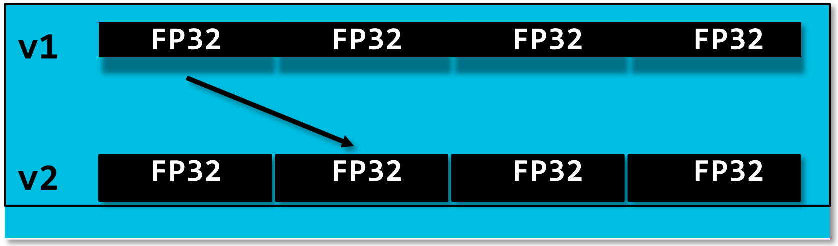


- **VLIW: Several operations per instruction word**
- **Some operations are free**

Register operations

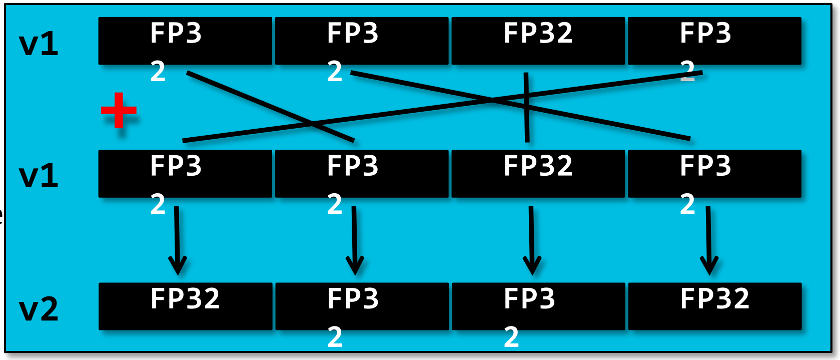
All operations can read or write any element or elements within a register

- **e.g.** float4 v1, v2;
  ...
  v2.y = v1.x



All operations can swizzle the elements in their input registers

- **e.g.** float4 v1, v2;
  ...
  v2 = v1 + v1.wxzy



These operations are mostly free, as are various data type expansion and shrinking operations

- **e.g. char - > short**

Images

- **Image data types are supported in hardware so use them!**

  Supports coordinate clipping, border colours, format conversion, etc
  Bi-linear pixel read only takes a cycle
  Happens in the texture pipeline – leaving ALU and L/S pipes free
  If you don't use it the texture unit turns off to save power
  Image stores won't use the texture unit, go through the L/S pipe instead

- **However buffers of integer arrays can be even faster**

  If you don't read off the edge of the image, and you use integer coordinates, and you don't need format conversion then…
  You can read and operate on 16 x 8-bit greyscale pixels at once Or 4 x RGBA8888 pixels at once

Load/Store Pipeline

The L1 and L2 caches are not as large as on desktop systems…and there are a great many threads

If you do a load in one instruction, by the next instruction (in the same thread) the data could possibly have been evicted

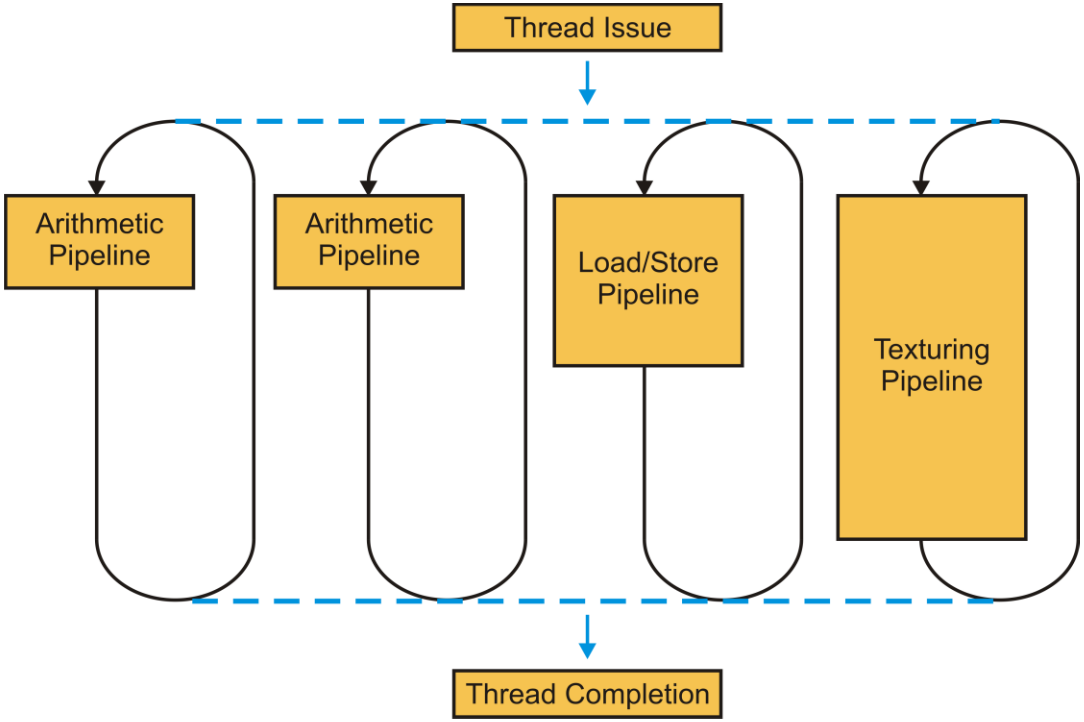
So pull as much data into registers in a single instruction as you can

One instruction is always better than using several instructions!

And a 16-byte load or store will typically take a single cycle (assuming no cache misses)

- Programming Suggestions
- Optimising with DS-5 Streamline
- Optimising: Two Examples
- General Advice

Inside a Core



$$T = \max(A_0, A_1, LS, Tex)$$

Fig 18. Find the bottleneck before make opts

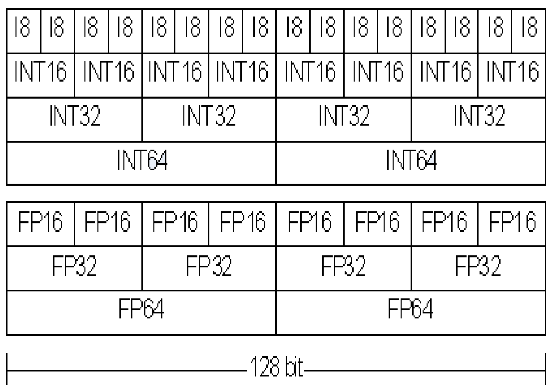
Latency Hiding by Parallelism

- **Executing a program on an ARM® Cortex®-A15 CPU**
  - **Execution of consecutive instructions overlap in time**
  - **Instruction latencies and branch predictions are important**
- **Executing a kernel on a ARM Mali™-T6xx GPU**
  - **Execution of different threads overlap in time**
  - **Execution of different instructions of a single thread never overlap**
- **This leads to latency tolerance**
  - **No need for branch predictors**
  - **No need to worry about pipeline latencies**
  - **Memory latency can still be an issue**

Arithmetic and Load/Store pipes

- SIMD: Several components per operation
  - 128-bit registers
  - highly scalable

- VLIW: Several operations per instruction word
  - Some operations are "free"



Hardware Counters

- **Counters per core**
  - **Active Cycles**
  - **Pipe activity**
  - **L1 cache**
- **Counters per Core Group**
  - **L2 caches**
  - **MMU**
- **Counters for the GPU**
  - **Active cycles**
- **Accessed through Streamline**
  - **Timeline of all hardware counters, and more**
  - **Explore the execution of the full application**
  - **Zoom in on details**

Streamline

Fig 19. Streamline

Memories

- Only one programmer controlled memory
  - Many transparent caches
- Memory copying takes time
  - It can easily dominate over kernel execution time
- Use appropriate memory allocation schemes
- Avoid synchronization points
  - Cache maintenance has a cost as well
- Streamline to the rescue
  - Visualize when kernels are executed
  - Many features not covered here

Hiding Pipeline Latency

- Needs enough threads
  - Limited by register usage
- When there are issues
  - Few instructions issued per cycle
  - Spilling of values to memory
- Symptoms
  - Low Max Local Workgroup Size in OpenCL™
  - Few instructions issued per cycle in limiting pipe
- Remedy
  - Smaller types means More values per register
  - Splitting kernels

Pipeline utilization

- Prefer small types
  - More components in 128 bits
- Prefer vector operations
  - More components per operation
- Balance work between the pipes
  - Do less – with the pipe that limits performance

$$T = \max(A_0, A_1, LS, Tex)$$

Finding the bottlenecks

- **Host application or Kernel execution**
  - **Avoid memory copying**
  - **Avoid cache flushes**
- **Which pipe is important?**
  - **Operations in other pipes incur little or no runtime cost**
- **Saving operations or saving registers**
  - **How much register pressure can we handle, and still hide the latencies?**
- **How well are we using the caches**
  - **Are instructions spinning around the LS pipe waiting for data?**

- Programming Suggestions
- Optimising with DS-5 Streamline
- Optimising: Two Examples
- General Advice

The Limiting Pipe

- Three hardware counters
  - Cycles active (#C)
  - Number of A instructions (#A)
  - Number of LS instructions (#LS)

$$\overline{y} = a\overline{x} + \overline{y}$$

- The goal
  - Similar values for #A and #LS   Both pipes used
  - Max(#A, #LS) similar to #C   Limiting pipe used every cycle
- Example:
  - #LS / #A = 1, #C up by < 10%

$$\overline{y} = 0.05\, a\overline{x} + 0.05\, a\overline{x} + 0.05\, a\overline{x} + \ldots + 0.05\, a\overline{x} + \overline{y}$$

Cache Utilization

- The Load/Store pipe hides latency
  - Many threads active

- Not always successful
  - Insufficient parallelism
  - Bad cache utilization
  - Failing threads will be reissued

- Reissue is a sign of cache-misses
  - Instruction words issued
  - Instruction words completed

- Example
  - Inter-thread stride for memory accesses

Execution order

$$\overline{y} = a\overline{x} + \overline{y}$$

- Kernel saxpy
  - Load from x
  - Load from y
  - Compute
  - Store to y

- Execution order
  - Threads 1 through N load from x
  - Threads 1 through N load from y
  - Threads 1 through N compute
  - Threads 1 through N store to y

- How many bytes should we load per thread?

A single instruction word

- We should have one load instruction word
  - The next bytes will be picked up by the next thread

- Loading less is bad
  - Does not utilize the VLIW and SIMD operations

- Loading more is bad
  - The next bytes will be loaded after all other threads have loaded their first

$$\overline{y} = a\overline{x} + \overline{y}$$

- Programming Suggestions
- Optimising with DS-5 Streamline
- Optimising: Two Examples
- General Advice

Know your bottleneck

- Use vector operations
- If you are bandwidth-limited, merge kernels
  - Avoid reloading data
- If you are register-limited, split kernels
  - Easier for the compiler to do a good job
- If you are Load-Store-limited, do less load-store
  - Compute complex expressions instead of using lookup-tables
- If you are Arithmetic-limited, do less arithmetic
  - Tabulate functions(FPGA)
  - Use polynomial approximations instead of special functions

Synchronization between threads

- Two options in OpenCL
  - Barriers inside a work-group
  - Atomics between work-groups

- We like atomics to ensure data consistency
  - But preferably on the same core

- Barriers can be useful to improve cache utilization
  - Limit divergence between threads
  - Keeping jobs small serves the same purpose

- We see examples of large jobs with many barriers
  - We often prefer small jobs with dependencies

Agenda

- 异构计算(HSA)
- Open Computing Language
- Mali midgard架构介绍
- OpenCL execution model on Mali
- Optimal OpenCL for Mali
- OpenCL Optimization Case Studies
- Developing tools

Case-study

Laplace filters are typically used in image processing

… often used for edge detection or image sharpening and can be part of a computer vision filter chain (LC_HDR lib e.g.)

This case study will go through a number of stages…

demonstrating a variety of optimization techniques and showing the change in performance at each stage

Our example will process and output 24-bit images

and we'll measure performance across a range of image sizes

But first, a couple of images samples showing the effect of the filter we are using…

Original

Filtered

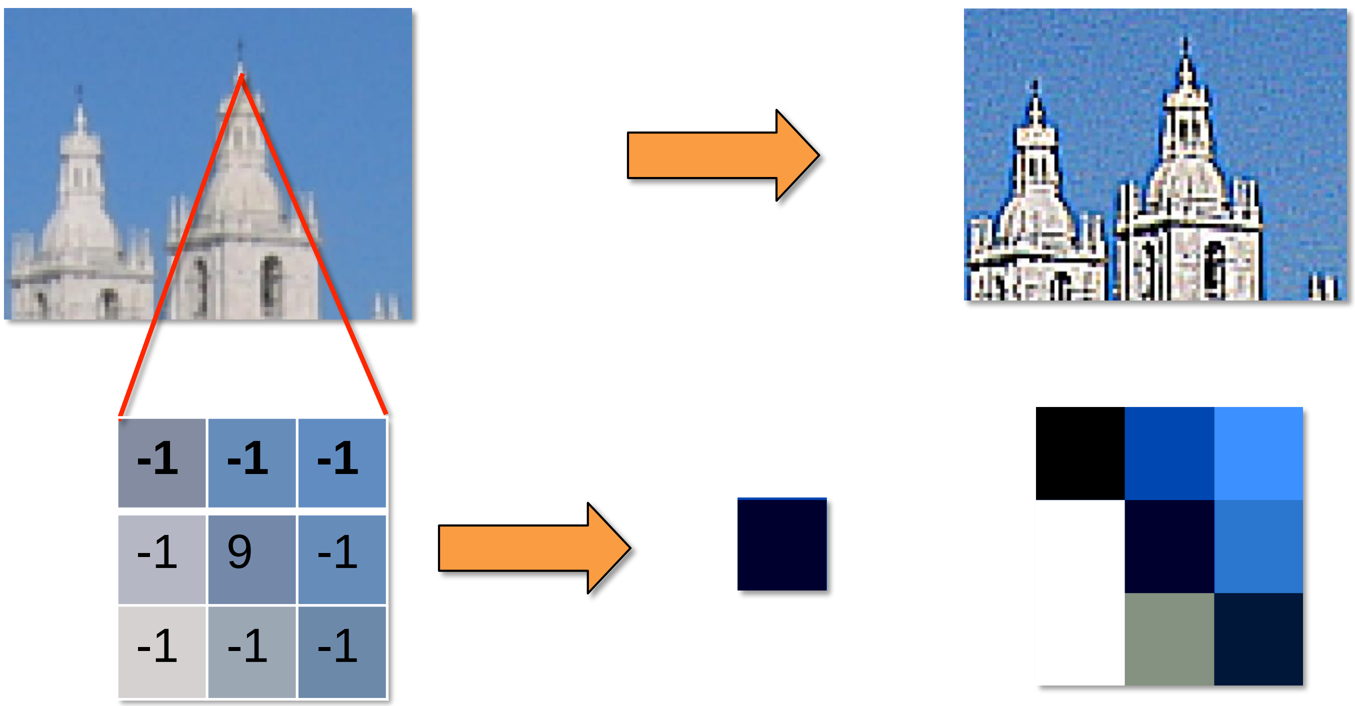


Fig 20. Laplace filters operator

Case-study

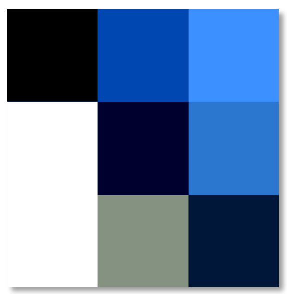
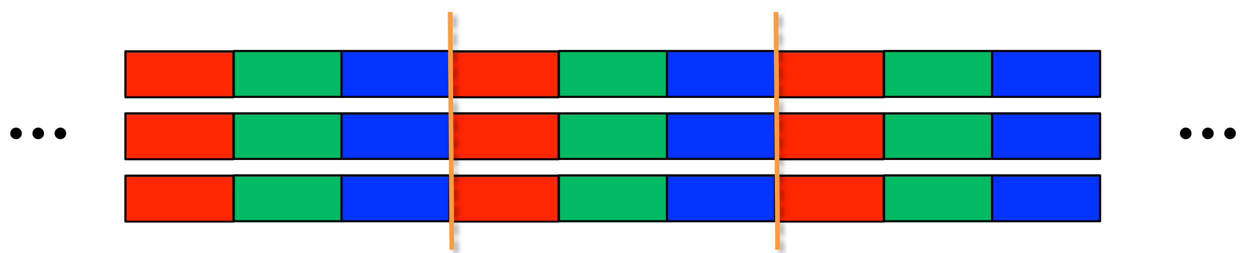


image “stride” = width x 3



Case-study

```
#define MAX(a,b) ((a)>(b)?(a):(b))
#define MIN(a,b) ((a)<(b)?(a):(b))

__kernel void math(global unsigned char *pdst, global unsigned char *psrc, int width, int height)
{
    int y         = get_global_id(0);
    int x         = get_global_id(1);
    int w         = width;
    int h         = height;
    int ind       = 0;
    int xBoundary = w - 2;
    int yBoundary = h - 2;

    if (x >= xBoundary || y >= yBoundary)
    {
        ind          = 3 * (x + w * y);
        pdst[ind]     = psrc[ind];
        pdst[ind + 1] = psrc[ind + 1];
        pdst[ind + 2] = psrc[ind + 2];
        return;
    }

    int bColor = 0, gColor = 0, rColor = 0;
    ind        = 3 * (x + w * y);

    bColor = bColor - psrc[ind] - psrc[ind+3] - psrc[ind+6] - psrc[ind+3*w] + psrc[ind+3*(1+w)] * 9 –  psrc[ind+3*(2+w)] - psrc[ind+3*2*w] -
             psrc[ind+3*(1+2*w)] - psrc[ind+3*(2+2*w)];
    gColor = gColor - psrc[ind+1] - psrc[ind+4] - psrc[ind+7] - psrc[ind+3*w+1] + psrc[ind+3*(1+w)+1] * 9 – psrc[ind+3*(2+w)+1] - psrc[ind+3*2*w+1] -
             psrc[ind+3*(1+2*w)+1] - psrc[ind+3*(2+2*w)+1];
    rColor = rColor - psrc[ind+2] - psrc[ind+5] - psrc[ind+8] - psrc[ind+3*w+2] + psrc[ind+3*(1+w)+2] * 9 –  psrc[ind+3*(2+w)+2] - psrc[ind+3*2*w+2] -
             psrc[ind+3*(1+2*w)+2] - psrc[ind+3*(2+2*w)+2];

    unsigned char blue  = (unsigned char)MAX(MIN(bColor, 255), 0);
    unsigned char green = (unsigned char)MAX(MIN(gColor, 255), 0);
    unsigned char red   = (unsigned char)MAX(MIN(rColor, 255), 0);
    ind           = 3 * (x + 1 + w * (y + 1));
    pdst[ind]     = blue;
    pdst[ind + 1] = green;  pdst[ind + 2] = red;

}
```

Case-study

```
#define MAX(a,b) ((a)>(b)?(a):(b))
#define MIN(a,b) ((a)<(b)?(a):(b))

__kernel void math(global unsigned char *pdst, global unsigned char *psrc, int width, int height)
{
    int y        = get_global_id(0);
    int x        = get_global_id(1);
    int w        = width;
    int h        = height;
    int ind       = 0;
    int xBoundary = w - 2;
    int yBoundary = h - 2;

    if (x >= xBoundary || y >= yBoundary)
    {
        ind          = 3 * (x + w * y);
        pdst[ind]     = psrc[ind];
        pdst[ind + 1] = psrc[ind + 1];
        pdst[ind + 2] = psrc[ind + 2];
        return;
    }

    int bColor = 0, gColor = 0, rColor = 0;
    ind         = 3 * (x + w * y);

    bColor = bColor - psrc[ind] - psrc[ind+3] - psrc[ind+6] - psrc[ind+3*w] + psrc[ind+3*(1+w)] * 9 –  psrc[ind+3*(2+w)] - psrc[ind+3*2*w] -
             psrc[ind+3*(1+2*w)] - psrc[ind+3*(2+2*w)];
    gColor = gColor - psrc[ind+1] - psrc[ind+4] - psrc[ind+7] - psrc[ind+3*w+1] + psrc[ind+3*(1+w)+1] * 9 – psrc[ind+3*(2+w)+1] - psrc[ind+3*2*w+1] -
             psrc[ind+3*(1+2*w)+1] - psrc[ind+3*(2+2*w)+1];
    rColor = rColor - psrc[ind+2] - psrc[ind+5] - psrc[ind+8] - psrc[ind+3*w+2] + psrc[ind+3*(1+w)+2] * 9 –  psrc[ind+3*(2+w)+2] - psrc[ind+3*2*w+2] -
             psrc[ind+3*(1+2*w)+2] - psrc[ind+3*(2+2*w)+2];

    unsigned char blue  = (unsigned char)MAX(MIN(bColor, 255), 0);
    unsigned char green = (unsigned char)MAX(MIN(gColor, 255), 0);
    unsigned char red   = (unsigned char)MAX(MIN(rColor, 255), 0);
    ind         = 3 * (x + 1 + w * (y + 1));
    pdst[ind]     = blue;
    pdst[ind + 1] = green;  pdst[ind + 2] = red;

}
```

Destination buffer

Source buffer

Image width

Image height

Case-study

```
#define MAX(a,b) ((a)>(b)?(a):(b))
#define MIN(a,b) ((a)<(b)?(a):(b))

__kernel void math(global unsigned char *pdst, global unsigned char *psrc, int width, int height)
{
    int y         = get_global_id(0);
    int x         = get_global_id(1);
    int w         = width;
    int h         = height;
    int ind       = 0;
    int xBoundary = w - 2;
    int yBoundary = h - 2;

    if (x >= xBoundary || y >= yBoundary)
    {
        ind           = 3 * (x + w * y);
        pdst[ind]     = psrc[ind];
        pdst[ind + 1] = psrc[ind + 1];
        pdst[ind + 2] = psrc[ind + 2];
        return;
    }

    int bColor = 0, gColor = 0, rColor = 0;
    ind        = 3 * (x + w * y);

    bColor = bColor - psrc[ind] - psrc[ind+3] - psrc[ind+6] - psrc[ind+3*w] + psrc[ind+3*(1+w)] * 9 –  psrc[ind+3*(2+w)] - psrc[ind+3*2*w] -
             psrc[ind+3*(1+2*w)] - psrc[ind+3*(2+2*w)];
    gColor = gColor - psrc[ind+1] - psrc[ind+4] - psrc[ind+7] - psrc[ind+3*w+1] + psrc[ind+3*(1+w)+1] * 9 – psrc[ind+3*(2+w)+1] - psrc[ind+3*2*w+1] -
             psrc[ind+3*(1+2*w)+1] - psrc[ind+3*(2+2*w)+1];
    rColor = rColor - psrc[ind+2] - psrc[ind+5] - psrc[ind+8] - psrc[ind+3*w+2] + psrc[ind+3*(1+w)+2] * 9 –  psrc[ind+3*(2+w)+2] - psrc[ind+3*2*w+2] -
             psrc[ind+3*(1+2*w)+2] - psrc[ind+3*(2+2*w)+2];

    unsigned char blue  = (unsigned char)MAX(MIN(bColor, 255), 0);
    unsigned char green = (unsigned char)MAX(MIN(gColor, 255), 0);
    unsigned char red   = (unsigned char)MAX(MIN(rColor, 255), 0);
    ind           = 3 * (x + 1 + w * (y + 1));
    pdst[ind]     = blue;
    pdst[ind + 1] = green;  pdst[ind + 2] = red;

}
```

Boundary checking… ideally we don't want to calculate for values at the right and bottom edges.
(But this might not be the best place to handle this.)

Case-study

```
#define MAX(a,b) ((a)>(b)?(a):(b))
#define MIN(a,b) ((a)<(b)?(a):(b))

__kernel void math(global unsigned char *pdst, global unsigned char *psrc, int width, int height)
{
    int y        = get_global_id(0);
    int x        = get_global_id(1);
    int w        = width;
    int h        = height;
    int ind       = 0;
    int xBoundary = w - 2;
    int yBoundary = h - 2;

    if (x >= xBoundary || y >= yBoundary)
    {
        ind          = 3 * (x + w * y);
        pdst[ind]     = psrc[ind];
        pdst[ind + 1] = psrc[ind + 1];
        pdst[ind + 2] = psrc[ind + 2];
        return;
    }

    int bColor = 0, gColor = 0, rColor = 0;
    ind        = 3 * (x + w * y);

    bColor = bColor - psrc[ind] - psrc[ind+3] - psrc[ind+6] - psrc[ind+3*w] + psrc[ind+3*(1+w)] * 9 –  psrc[ind+3*(2+w)] - psrc[ind+3*2*w] -
             psrc[ind+3*(1+2*w)] - psrc[ind+3*(2+2*w)];
    gColor = gColor - psrc[ind+1] - psrc[ind+4] - psrc[ind+7] - psrc[ind+3*w+1] + psrc[ind+3*(1+w)+1] * 9 – psrc[ind+3*(2+w)+1] - psrc[ind+3*2*w+1] -
             psrc[ind+3*(1+2*w)+1] - psrc[ind+3*(2+2*w)+1];
    rColor = rColor - psrc[ind+2] - psrc[ind+5] - psrc[ind+8] - psrc[ind+3*w+2] + psrc[ind+3*(1+w)+2] * 9 –  psrc[ind+3*(2+w)+2] - psrc[ind+3*2*w+2] -
             psrc[ind+3*(1+2*w)+2] - psrc[ind+3*(2+2*w)+2];

    unsigned char blue  = (unsigned char)MAX(MIN(bColor, 255), 0);
    unsigned char green = (unsigned char)MAX(MIN(gColor, 255), 0);
    unsigned char red   = (unsigned char)MAX(MIN(rColor, 255), 0);
    ind          = 3 * (x + 1 + w * (y + 1));
    pdst[ind]     = blue;
    pdst[ind + 1] = green;  pdst[ind + 2] = red;

}
```

The main calculation… we need to perform this for the red, green and blue color components…

Case-study

```
#define MAX(a,b) ((a)>(b)?(a):(b))
#define MIN(a,b) ((a)<(b)?(a):(b))

__kernel void math(global unsigned char *pdst, global unsigned char *psrc, int width, int height)
{
    int y        = get_global_id(0);
    int x        = get_global_id(1);
    int w        = width;
    int h        = height;
    int ind      = 0;
    int xBoundary = w - 2;
    int yBoundary = h - 2;

    if (x >= xBoundary || y >= yBoundary)
    {
        ind          = 3 * (x + w * y);
        pdst[ind]    = psrc[ind];
        pdst[ind + 1] = psrc[ind + 1];
        pdst[ind + 2] = psrc[ind + 2];
        return;
    }

    int bColor = 0, gColor = 0, rColor = 0;
    ind        = 3 * (x + w * y);

    bColor = bColor - psrc[ind] - psrc[ind+3] - psrc[ind+6] - psrc[ind+3*w] + psrc[ind+3*(1+w)] * 9 –  psrc[ind+3*(2+w)] - psrc[ind+3*2*w] -
             psrc[ind+3*(1+2*w)] - psrc[ind+3*(2+2*w)];
    gColor = gColor - psrc[ind+1] - psrc[ind+4] - psrc[ind+7] - psrc[ind+3*w+1] + psrc[ind+3*(1+w)+1] * 9 – psrc[ind+3*(2+w)+1] - psrc[ind+3*2*w+1] -
             psrc[ind+3*(1+2*w)+1] - psrc[ind+3*(2+2*w)+1];
    rColor = rColor - psrc[ind+2] - psrc[ind+5] - psrc[ind+8] - psrc[ind+3*w+2] + psrc[ind+3*(1+w)+2] * 9 –  psrc[ind+3*(2+w)+2] - psrc[ind+3*2*w+2] -
             psrc[ind+3*(1+2*w)+2] - psrc[ind+3*(2+2*w)+2];

    unsigned char blue  = (unsigned char)MAX(MIN(bColor, 255), 0);
    unsigned char green = (unsigned char)MAX(MIN(gColor, 255), 0);
    unsigned char red   = (unsigned char)MAX(MIN(rColor, 255), 0);
    ind         = 3 * (x + 1 + w * (y + 1));
    pdst[ind]    = blue;
    pdst[ind + 1] = green;  pdst[ind + 2] = red;

}
```

Finally we clamp the results to make sure they lie between 0 and 255… and then write out to the destination…

Case-study

Results

Image	Pixels	Time (s)
768 x 432	331,776	0.0107
2560 x 1600	4,096,000	0.0850
2048 x 2048	4,194,304	0.0865
5760 x 3240	18,662,400	0.382
7680 x 4320	33,177,600	0.680

Mali T604 @ 533MHz

CPU
0.0229 x0.5
0.125 x0.7
0.128 x0.7
0.572 x0.7
1.02 x0.7

Single A15 @ 1.7GHz

Use the offline compiler mali_clcc to analyse the kernel

```
zhangjintao@COMIP-LDD5:~/laplace# mali_clcc -v laplace.cl

8 work registers used, 8 uniform registers used


Pipelines:                               A / L / T / Overall
Number of instruction words emitted:     54 +31 + 0 = 85
Number of cycles for shortest code       3 / 4 / 0 =4  (L bound)
path:  Number of cycles for longest      25.5 /28 / 0 = 28  (L bound)
code path:
Note: The cycle counts do not include possible stalls due to cache misses.
```

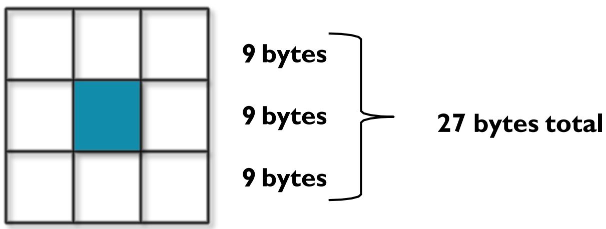
LEADCORE 联芯科技

Case-study

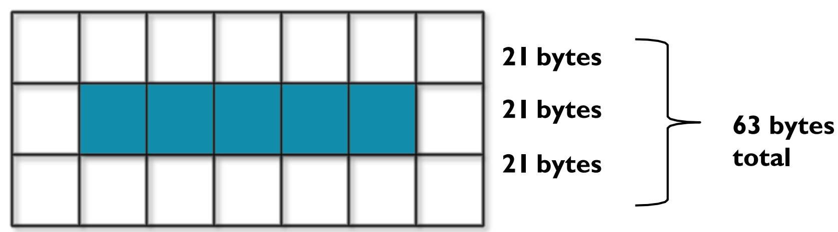
- **Replace the data fetch (= psrc[index]) with vloadN**
  - **Each vload16 can load 5 pixels at a time (at 3 bytes-per-pixel)**
  - **This load should complete in a single cycle**
- **Perform the Laplace calculation as a vector calculation**
  - **Then Mali works on all 5 pixels at once**
- **Replace the data store (pdst[index] =) with vstoreN**
  - **Allows us to write out multiple values at a time**
  - **Need to be careful to only output 15 bytes (3 pixels)**
- **As we'll be running 5 times fewer work items, we'll need to update the globalWorkSize values…**

```
globalWorkSize[0] = image_height;  globalWorkSize[1] = (image_width / 5);
```

From processing 1 pixel…

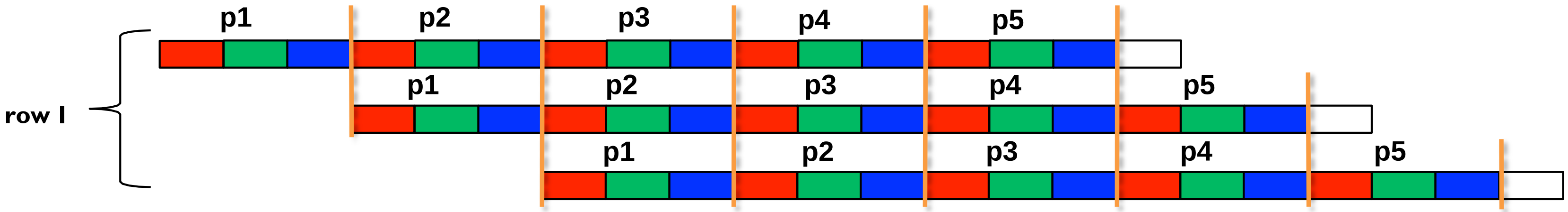


…to processing 5 pixels...

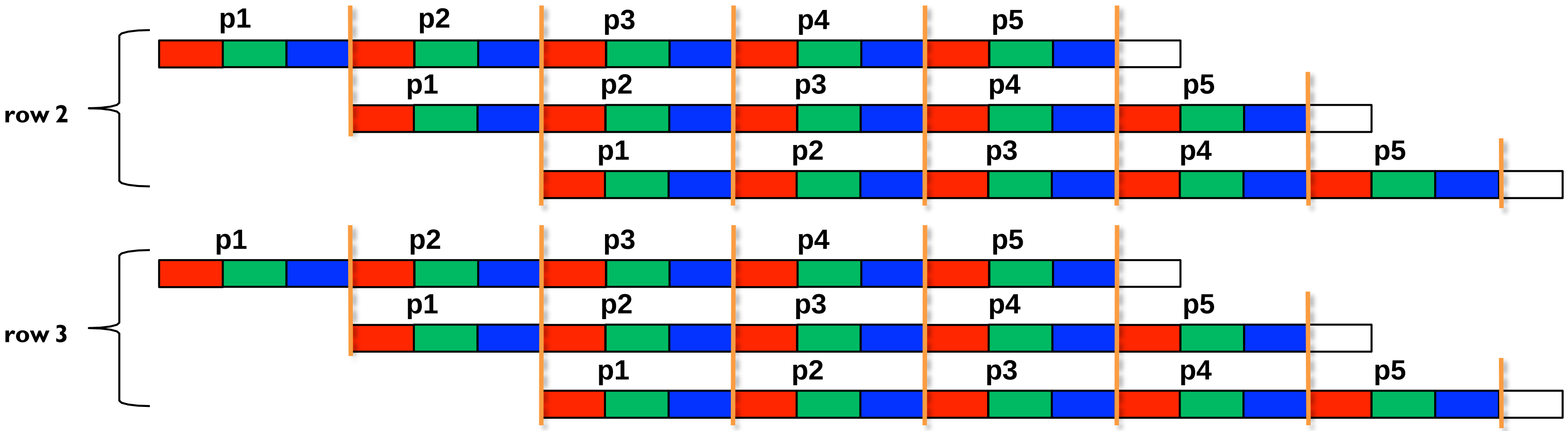


But we would like to load this data in a way that allows us to efficiently calculate the results in a single vector calculation…

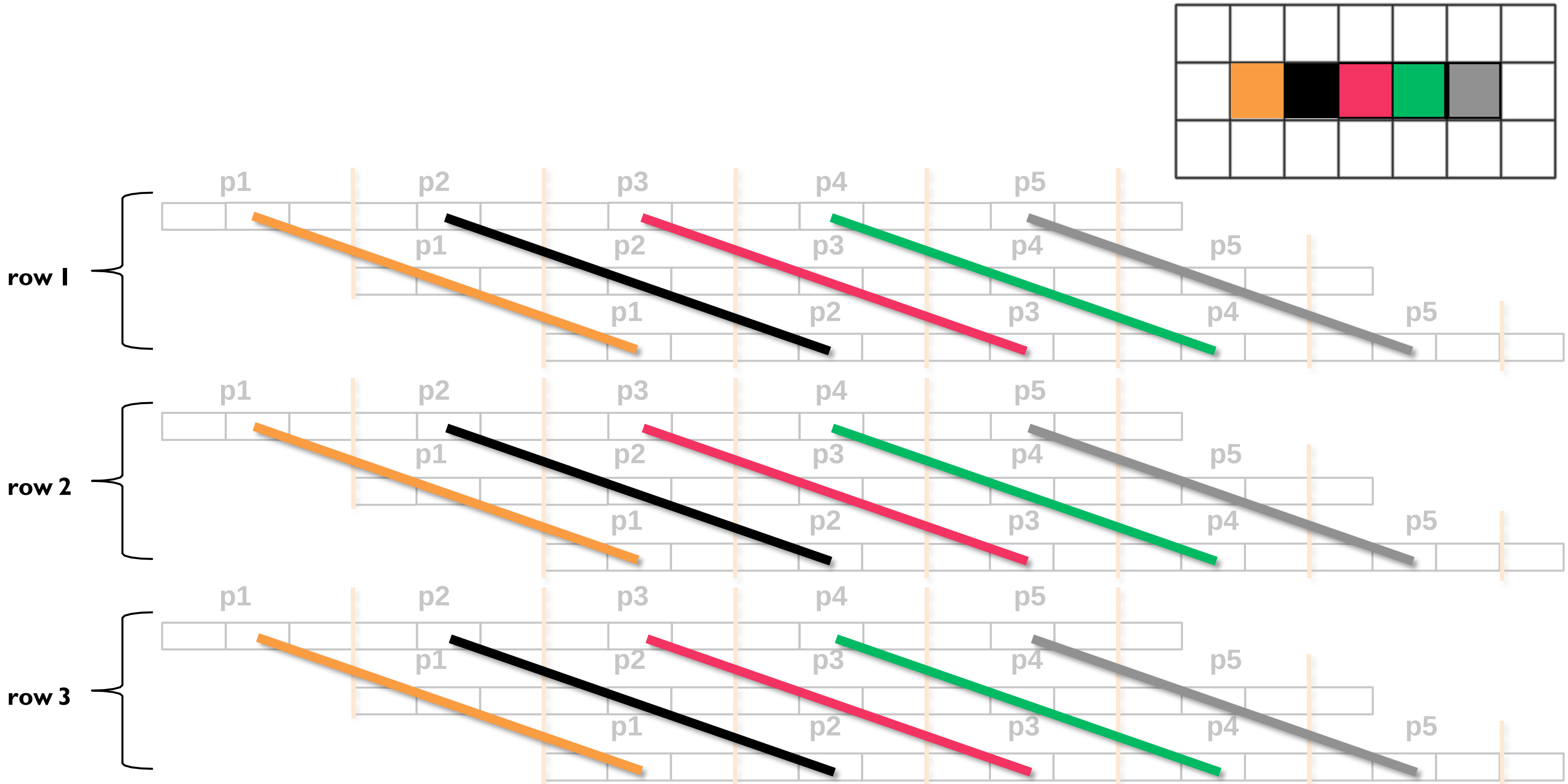
3 x overlapping, 16-byte reads from row1 (`vload16`)…



And the same for rows 2 and 3…



Case-study



Case-study

```
__kernel void math(global unsigned char *pdst, global unsigned char *psrc, int width, int height)
{
    int y        = get_global_id(0);
    int x        = get_global_id(1);
    int w        = width;
    int h        = height;
    int          = x * 5 * 3 + w * y * 3;
    ind
    uchar16 row1a_ = vload16(0, psrc + ind);
    uchar16  row1b_ = vload16(0, psrc + ind + 3);
    uchar16 row1c_ = vload16(0, psrc + ind + 6);
    uchar16 row2a_ = vload16(0, psrc + ind + (w * 3));
    uchar16 row2b_ = vload16(0, psrc + ind + (w * 3) + 3);
    uchar16 row2c_ = vload16(0, psrc + ind + (w * 3) + 6);
    uchar16 row3a_ = vload16(0, psrc + ind + (w * 6));
    uchar16 row3b_ = vload16(0, psrc + ind + (w * 6) + 3);
    uchar16  row3c_ = vload16(0, psrc + ind + (w * 6) + 6);

    int16  row1a = convert_int16(row1a_);
    int16  row1b = convert_int16(row1b_);
    int16  row1c = convert_int16(row1c_);
    int16  row2a = convert_int16(row2a_);
    int16  row2b = convert_int16(row2b_);
    int16  row2c = convert_int16(row2c_);
    int16  row3a = convert_int16(row3a_);
    int16  row3b = convert_int16(row3b_);
    int16   row3c = convert_int16(row3c_);

    int16    res     = (int)0 - row1a - row1b - row1c - row2a - row2b * (int)9 - row2c - row3a - row3b - row3c;
    res              = clamp(res, (int16)0, (int16)255);
    uchar16 res_row  = convert_uchar16(res);

    vstore8(res_row.s01234567, 0, pdst + ind);
    vstore4(res_row.s89ab,     0, pdst + ind + 8);
    vstore2(res_row.scd,       0, pdst + ind + 12);
    pdst[ind + 14]    = res_row.se;
}
```

Parameter 3 now refers to the width of the image / 5.

3 overlapping 16-byte reads for each of the 3 rows
(5 pixels-worth in each read)

Convert each 16-byte uchar vector to int16 vectors

Case-study

```
__kernel void math(global unsigned char *pdst, global unsigned char *psrc, int width, int height)
{
    int y        = get_global_id(0);
    int x        = get_global_id(1);
    int w        = width;
    int h        = height;
    int          = x * 5 * 3 + w * y * 3;
    ind
    uchar16 row1a_ = vload16(0, psrc + ind);
    uchar16  row1b_ = vload16(0, psrc + ind + 3);
    uchar16 row1c_ = vload16(0, psrc + ind + 6);
    uchar16 row2a_ = vload16(0, psrc + ind + (w * 3));
    uchar16 row2b_ = vload16(0, psrc + ind + (w * 3) + 3);
    uchar16 row2c_ = vload16(0, psrc + ind + (w * 3) + 6);
    uchar16 row3a_ = vload16(0, psrc + ind + (w * 6));
    uchar16 row3b_ = vload16(0, psrc + ind + (w * 6) + 3);
    uchar16  row3c_ = vload16(0, psrc + ind + (w * 6) + 6);

    int16  row1a = convert_int16(row1a_);
    int16  row1b = convert_int16(row1b_);
    int16  row1c = convert_int16(row1c_);
    int16  row2a = convert_int16(row2a_);
    int16  row2b = convert_int16(row2b_);
    int16  row2c = convert_int16(row2c_);
    int16  row3a = convert_int16(row3a_);
    int16  row3b = convert_int16(row3b_);
    int16   row3c  = convert_int16(row3c_);

    int16    res     = (int)0 - row1a - row1b - row1c - row2a - row2b * (int)9 - row2c - row3a - row3b - row3c;
    res              = clamp(res, (int16)0, (int16)255);
    uchar16 res_row  = convert_uchar16(res);

    vstore8(res_row.s01234567, 0, pdst + ind);
    vstore4(res_row.s89ab,     0, pdst + ind + 8);
    vstore2(res_row.scd,       0, pdst + ind + 12);
    pdst[ind + 14]    = res_row.se;
}
```

Perform the Laplace calculation on all five pixels at once
Then clamp the values between 0 and 255 (using the BIFL!)

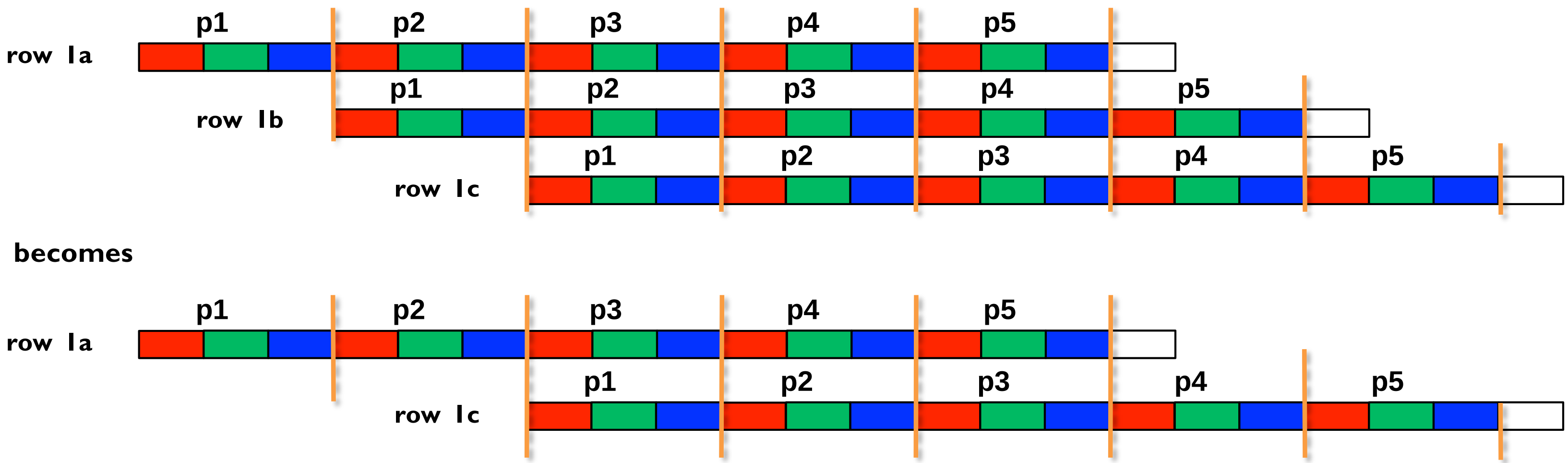
Convert back to uchar16… and then write 5 pixels to destination buffer

Vectorization Results

Image	Pixels	Original	Opt I
768 x 432	331,776	0.0107	x1.4
2560 x 1600	4,096,000	0.0850	x4.5
2048 x 2048	4,194,304	0.0865	x1.7
5760 x 3240	18,662,400	0.382	x6.0
7680 x 4320	33,177,600	0.680	x6.2
	Work registers:	8	8+
	ALU cycles:	25.5	22.5
	L/S cycles:	28	13

Case-study

- **We can reduce the number of loads**
  - **by synthesizing the middle vector row from the left and right rows…**



row 1b ⟵ row1(p2, p3, p4, p5) + row2(p6)

- **We can reduce the number of loads**
  - **by synthesizing the middle vector row from the left and right rows…**

```
uchar16  row1a_   = vload16(0,  psrc + ind);
uchar16  row1b_   = vload16(0,  psrc + ind + 3);
uchar16  row1c_   = vload16(0,  psrc + ind + 6);
uchar16  row2a_   = vload16(0,  psrc + ind + (w   * 3));
uchar16  row2b_   = vload16(0,  psrc + ind + (w * 3) + 3);
uchar16  row2c_   = vload16(0,  psrc + ind + (w * 3) + 6);
uchar16  row3a_   = vload16(0,  psrc + ind + (w   * 6));
uchar16  row3b_   = vload16(0,  psrc + ind + (w * 6) + 3);
uchar16  row3c_   = vload16(0,  psrc + ind + (w * 6) + 6);
```

becomes…

```
uchar16 row1a_   = vload16(0, psrc + ind);
uchar16 row1c_   = vload16(0, psrc + ind + 6);
uchar16 row1b_   = (uchar16)(row1a_.s3456789a, row1c_.s56789abc);
uchar16 row2a_   = vload16(0, psrc + ind + (w * 3));
uchar16 row2c_   = vload16(0, psrc + ind + (w * 3) + 6);
uchar16 row2b_   = (uchar16)(row2a_.s3456789a, row2c_.s56789abc);
uchar16 row3a_   = vload16(0, psrc + ind + (w * 6));
uchar16 row3c_   = vload16(0, psrc + ind + (w * 6) + 6);
uchar16 row3b_   = (uchar16)(row3a_.s3456789a, row3c_.s56789abc);
```

Synthesize Loads Results

Image	Pixels	Original	Opt 1 (Vectorize)	Opt 2 (Synth. loads)
768 x 432	331,776	0.0107	x1.4	x1.4
2560 x 1600	4,096,000	0.0850	x4.5	x4.5
2048 x 2048	4,194,304	0.0865	x1.7	x2.0
5760 x 3240	18,662,400	0.382	x6.0	x6.0
7680 x 4320	33,177,600	0.680	x6.2	x6.3
	Work registers:	8	8+	8
	ALU cycles:	25.5	22.5	24.5
	L/S cycles:	28	13	8

- **Use short16instead of int16**
  - **smaller register use allows for a larger CL_KERNEL_WORK_GROUP_SIZE available for kernel execution**

```
int16   row1a= convert_int16(row1a_);
int16   row1b= convert_int16(row1b_);
int16   row1c= convert_int16(row1c_);
int16   row2a= convert_int16(row2a_);
int16   row2b= convert_int16(row2b_);
int16   row2c  = convert_int16(row2c_);
int16   row3a= convert_int16(row3a_);
int16   row3b= convert_int16(row3b_);
int16   row3c  = convert_int16(row3c_);

int16   res    = (int)0 - row1a - row1b - row1c - row2a - row2b * (int)9 - row2c - row3a - row3b - row3c;
res            = clamp(res, (int16)0, (int16)255);
uchar16 res_row  = convert_uchar16(res);
```

becomes

```
short16  row1a= convert_short16(row1a_);
short16  row1b= convert_short16(row1b_);
short16   row1c = convert_short16(row1c_);
short16  row2a= convert_short16(row2a_);
short16  row2b= convert_short16(row2b_);
short16  row2c= convert_short16(row2c_);
short16  row3a= convert_short16(row3a_);
short16  row3b= convert_short16(row3b_);
short16   row3c = convert_short16(row3c_);

short16   res    = (short)0 - row1a - row1b - row1c - row2a - row2b * (short)9 - row2c - row3a - row3b - row3c;
res              = clamp(res, (short16)0, (short16)255);
uchar16 res_row  = convert_uchar16(res);
```

Using Short Ints Results

Image	Pixels	Original	Opt 1 (Vectorize)	Opt 2 (Synth. loads)	Opt 3 (Shorts)
768 x 432	331,776	0.0107	x1.4	x1.4	x1.5
2560 x 1600	4,096,000	0.0850	x4.5	x4.5	x6.2
2048 x 2048	4,194,304	0.0865	x1.7	x2.0	x1.9
5760 x 3240	18,662,400	0.382	x6.0	x6.0	x8.5
7680 x 4320	33,177,600	0.680	x6.2	x6.3	x9.0
	Work registers:	8	8+	8	7
	ALU cycles:	25.5	22.5	24.5	13.5
	L/S cycles:	28	13	8	9

- Try 4-pixels per work-item rather than 5
- With some image sizes perhaps the driver can optimize more efficiently when 4 pixels are being calculated

```
__kernel void math(global unsigned char *pdst, global unsigned char *psrc, int width, int height)
{
    int y         = get_global_id(0);
    int x         = get_global_id(1);
    int w         = width;
    int h         = height;
    int           = x * 5 * 3 + w * y * 3;
    ind

    ...
```

becomes…

```
__kernel void math(global unsigned char *pdst, global unsigned char *psrc, int width, int height)
{
    int y         = get_global_id(0);
    int x         = get_global_id(1);
    int w         = width;
    int h         = height;
    int           = x * 4 * 3 + w * y * 3;
    ind
    ...
```

- And our date write out becomes simpler…

```
...

vstore8(res_row.s01234567, 0, pdst + ind);
vstore4(res_row.s89ab, 0, pdst + ind + 8);
vstore2(res_row.scd, 0, pdst + ind + 12);
pdst[ind + 14] = res_row.se;
```

becomes…

```
vstore8(res_row.s01234567, 0, pdst + ind);
vstore4(res_row.s89ab, 0, pdst + ind + 8);
```

- and we need to adjust the setup code to adjust the work-item count.

Computing 4 Pixels Results

			Vectorize	Synth. loads	Shorts	4 Pixels
Image	**Pixels**	**Original**	**Opt 1**	**Opt 2**	**Opt 3**	**Opt 4**
768 x 432	331,776	0.0107	x1.4	x1.4	x1.5	x1.6
2560 x 1600	4,096,000	0.0850	x4.5	x4.5	x6.2	x5.2
2048 x 2048	4,194,304	0.0865	x1.7	x2.0	x1.9	x5.3
5760 x 3240	18,662,400	0.382	x6.0	x6.0	x8.5	x7.2
7680 x 4320	33,177,600	0.680	x6.2	x6.3	x9.0	x7.5
	Work registers:	8	8+	8	7	6
	ALU cycles:	25.5	22.5	24.5	13.5	14
	L/S cycles:	28	13	8	9	6

Case-study

- **How about 8 pixels per work-item?**

Case-study

```
__kernel void math(global unsigned char *pdst, global unsigned char *psrc, int w, int h)
{
    const int y        = get_global_id(0);
    const int x        = get_global_id(1) * 8;
    int       ind      = (x + w * y) * 3;
    short16   acc_xy;
    short8    acc_z;

    uchar16 l_0     = vload16(0, psrc + ind);
    uchar16 r_0     = vload16(0, psrc + ind + 14);
    short16 a_xy_0  = convert_short16((uchar16)(l_0.s0123456789abcdef));
    short8  a_z_0   = convert_short8((uchar8)(r_0.s23456789));
    short16 b_xy_0  = convert_short16((uchar16)(l_0.s3456789a, l_0.sbcde, r_0.s1234));
    short8  b_z_0   = convert_short8((uchar8)(r_0.s56789abc));
    short16 c_xy_0  = convert_short16((uchar16)(l_0.s6789abcd, r_0.s01234567));
    short8  c_z_0   = convert_short8((uchar8)(r_0.s89abcdef));
    acc_xy          = – a_XY_0 – b_xy_0 – c_xy_0;
    acc_z           = – a_z_0  — b_z_0  –c_z_0;

    uchar16 l_1     = vload16(0, psrc + ind + (w * 3));
    uchar16 r_1     = vload16(0, psrc + ind + (w * 3) + 14);
    short16 a_xy_1  = convert_short16((uchar16)(l_1.s0123456789abcdef));
    short8  a_z_1   = convert_short8((uchar8)(r_1.s23456789));
    short16 b_xy_1  = convert_short16((uchar16)(l_1.s3456789a, l_0.sbcde, r_0.s1234));
    short8  b_z_1   = convert_short8((uchar8)(r_1.s56789abc));
    short16 c_xy_1  = convert_short16((uchar16)(l_1.s6789abcd, r_0.s01234567));
    short8  c_z_1   = convert_short8((uchar8)(r_1.s89abcdef));
    acc_xy          = - a_xy_1 + b_xy_1 * (short)9 — c_xy_1;
    acc_z          += - a_z_1  + b_z_1  * (short)9 —c_z_1;

    uchar16 l_2     = vload16(0, psrc + ind + (w * 6));
    uchar16 r_2     = vload16(0, psrc + ind + (w * 6) + 14);
    short16 a_xy_2  = convert_short16((uchar16)(l_2.s0123456789abcdef));
    short8  a_z_2   = convert_short8((uchar8)(r_2.s23456789));
    short16 b_xy_2  = convert_short16((uchar16)(l_2.s3456789a, l_0.sbcde, r_0.s1234));
    short8  b_z_2   = convert_short8((uchar8)(r_2.s56789abc));
    short16 c_xy_2  = convert_short16((uchar16)(l_2.s6789abcd, r_0.s01234567));
    short8  c_z_2   = convert_short8((uchar8)(r_2.s89abcdef));
    acc_xy         += - a_xy_2 — b_xy_2 — c_xy_2;
    acc_z          += - a_z_2 – b_z_2 -c_z_2;

    short16 res_xy  = clamp(acc_xy, (short16)0, (short16)255);
    short8 res_z    = clamp(acc_z,  (short8)0, (short8)255);

    vstore16(convert_uchar16(res_xy), 0, pdst + ind);
    vstore8(convert_uchar8(res_z), 0, pdst + ind + 16);
}
```

Computing 8 Pixels: Results

Image	Pixels	Original	Opt 1 (Vectorize)	Opt 2 (Synth. loads)	Opt 3 (Shorts)	Opt 4 (4 Pixels)	Opt 5 (8 Pixels)
768 x 432	331,776	0.0107	x1.4	x1.4	x1.5	x1.6	x1.2
2560 x 1600	4,096,000	0.0850	x4.5	x4.5	x6.2	x5.2	x5.6
2048 x 2048	4,194,304	0.0865	x1.7	x2.0	x1.9	x5.3	x5.8
5760 x 3240	18,662,400	0.382	x6.0	x6.0	x8.5	x7.2	x8.4
7680 x 4320	33,177,600	0.680	x6.2	x6.3	x9.0	x7.5	x9.1
	Work registers:	8	8+	8	7	6	8+
	ALU cycles:	25.5	22.5	24.5	13.5	14	24
	L/S cycles:	28	13	8	9	6	11

Case-study

Original version: Scalar code

Optimization 1: Vectorize

Process 5 pixels per work-item

Vector loads (vload**n**) and vector stores (vstore**n**)

Much better use of the GPU ALU: Up to **x6.2** performance increase

Optimization 2: Synthesised loads

Reduce the number of loads by synthesising values

Performance increase: up to **x6.3** over original

Optimization 3: Replace int16 with short16

Reduces the kernel register count

Performance increase: up to **x9.0** over original

Optimization 4: Try 4 pixels per work-item rather than 5

Performance increase: up to **x7.5** over original

but it depends on the image size

Optimization 5: Try 8 pixels per work-item

Performance increase: up to **x9.1** over original… but a mixed bag.

Agenda

- 异构计算(HSA)
- Open Computing Language
- Mali midgard架构介绍
- OpenCL execution model on Mali
- Optimal OpenCL for Mali
- OpenCL Optimization Case Studies
- Developing tools

Developing Tools

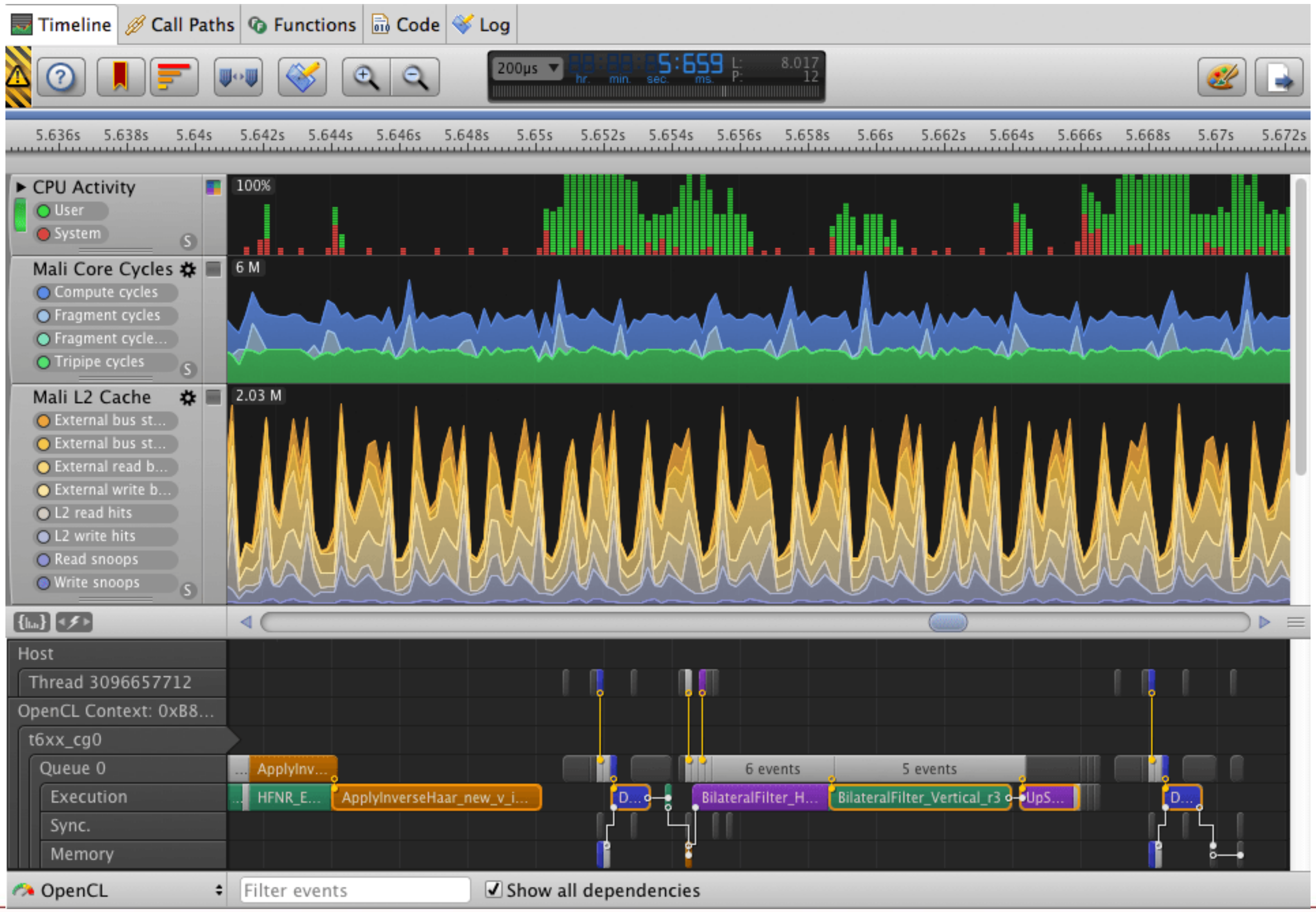
DS-5 streamline

Off-line compiler： http://malideveloper.arm.com/resources/tools/mali-offline-compiler/

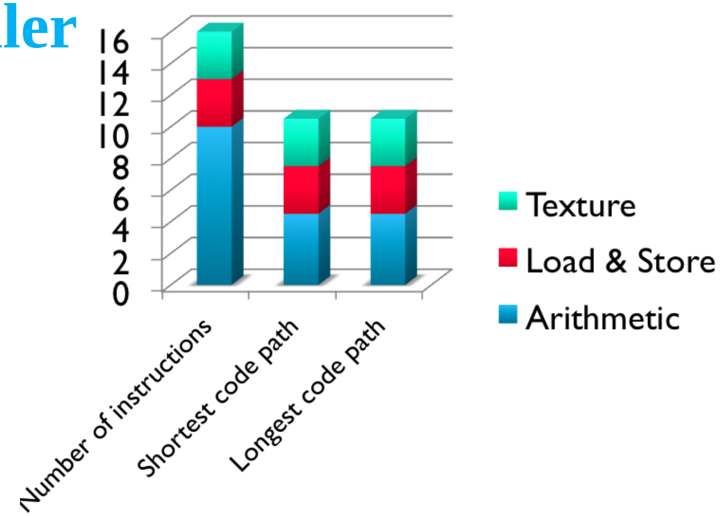
Off-line compiler： mali_clcc

Mali graphic debugger

DS-5 streamline



Offline compiler

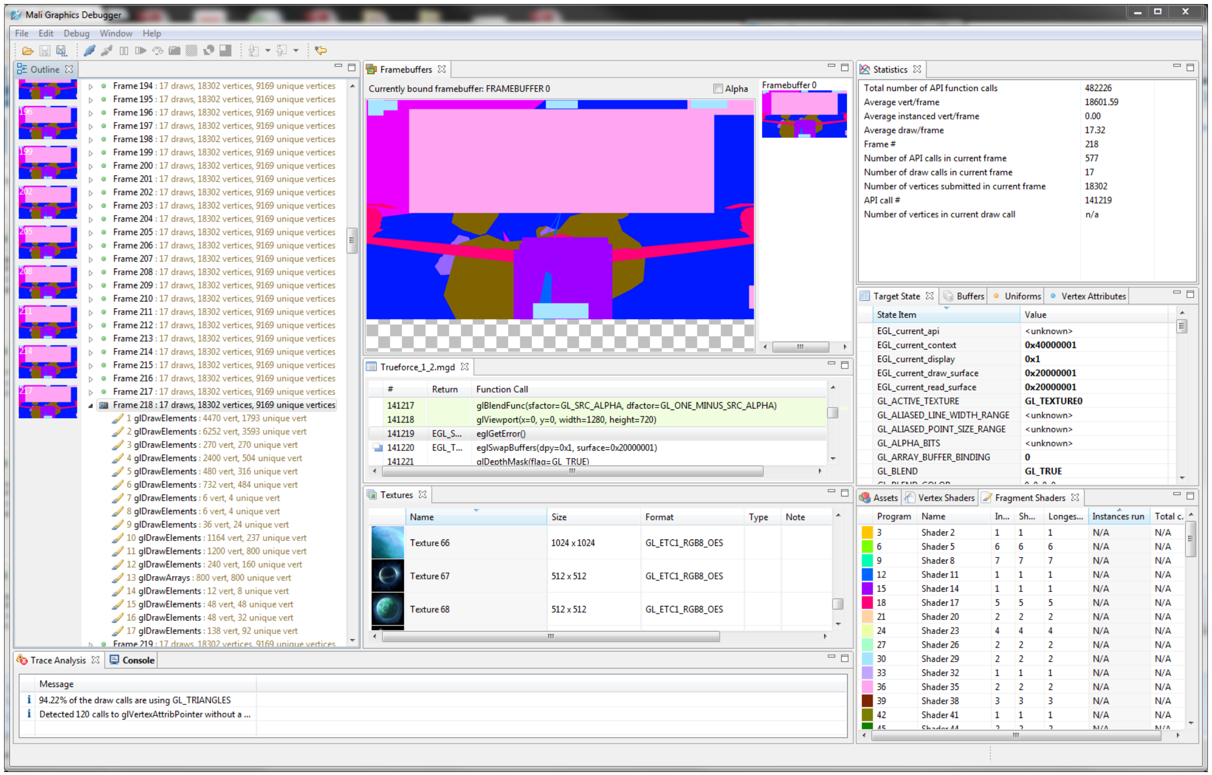



```
C:\Program Files (x86)\ARM\Mali Developer Tools\Mali Offline Shader Compiler v4.
0.0\bin>malisc.exe -v --frag --core=Mali-T600 "C:\Documents\Presentations\Own\gd
c\Example_FresnelFp.glsles.OLD"
0 error(s), 0 warning(s)

2 work registers used, 1 uniform registers used

Pipelines:                              A / L / T / Overall
Number of instruction words emitted:    10 + 3 + 3 = 16
Number of cycles for shortest code path: 4.5 / 3 / 3 = 4.5 (A bound)
Number of cycles for longest code path:  4.5 / 3 / 3 = 4.5 (A bound)
Note: The cycle counts do not include possible stalls due to cache misses.
```

Mali graphic debugger



Thank you!