Optimising OpenCL kernels for the ARM® Mali™-T600 GPUs

Johan Gronqvist and Anton Lokhmotov,
ARM

OpenCL is a relatively young industry-backed standard API that aims to provide *functional portability* across systems equipped with computational accelerators such as GPUs: a standard-conforming OpenCL program can be executed on any standard-conforming OpenCL implementation.

OpenCL, however, does not address the issue of *performance portability*: transforming an OpenCL program to achieve higher performance on one device may actually lead to lower performance on another device, since performance may depend significantly on low-level details, such as iteration space mapping and data layout [Howes et al. 10, Ryoo et al. 08].

Due to the popularity of certain GPU architectures, some optimisations have become hallmarks of GPU computing, *e.g.* coalescing global memory accesses or using local memory. Emerging mobile and embedded OpenCL-capable GPUs, however, have rather different organisation. Therefore, even seasoned GPU developers may need to forgo their instincts and learn new techniques when optimising for battery-powered GPU brethren.

In this chapter, we introduce the ARM Mali-T600 GPU series (Section 1.2) and discuss performance characteristics of several versions of the Sobel edge detection filter (Section 1.3) and the general matrix multiplication (Section 1.4).[1]

We make no claim that the presented versions are the fastest possible implementations of the selected algorithms. Rather, we aim to provide an insight into which transformations should be considered when optimising kernel code for the Mali-T600 GPUs. Therefore, the described behaviour may differ from the actual behaviour for expository purposes.

We perform our experiments on an Arndale development board[2] powered by the Samsung Exynos 5250 chip. Exynos 5250 comprises a dual-core Cortex-A15 CPU at 1.7 GHz and a quad-core Mali-T604 GPU at 533 MHz. The OpenCL driver is of version 3.0 Beta.

[1]Source code for some versions is available in the Mali OpenCL SDK [ARM Limited 13].

[2]http://www.arndaleboard.org

1.1 Overview of the OpenCL programming model

In the OpenCL programming model, the host (*e.g.* a CPU) manages one or more devices (*e.g.* a GPU) through calls to the OpenCL API. A call to the `clEnqueueNDRange()` API function submits a job that executes on a selected device the same program (*kernel*)[3] as a collection of *work-items*.

Each work-item has a unique index (*global id*) in a multi-dimensional iteration space (*ND-range*), specified as the *global work size* argument to `clEnqueueNDRange()`. The *local work size* argument to `clEnqueueNDRange()` determines how the ND-range is partitioned into uniformly sized *work-groups*. Work-items within the same work-group can synchronise using the `barrier()` built-in function which must be executed by all work-items in the work-group before any work-item continues execution past the barrier.

In our examples, the ND-range is two-dimensional: work-items iterate over the pixels of the output image (Section 1.3) or the elements of the output matrix (Section 1.4), while work-groups iterate over partitions (or *tiles*) of the same. For example, the local work size of $(4, 16)$ would result in 64 work-items per work-group, with the first work-group having global ids (x, y), where $0 \le x < 4$ and $0 \le y < 16$.

[3]The kernel is typically written in the OpenCL C language, which is a superset of a subset of the C99 language standard.

1.2 ARM Mali-T600 GPU series

The ARM Mali-T600 GPU series based on the Midgard architecture is designed to meet the growing needs of graphics and compute applications for a wide range of consumer electronics: from phones and tablets to TVs and beyond. The Mali-T604 GPU, the first implementation of the Mali-T600 series, was the first mobile and embedded class GPU to pass the OpenCL v1.1 Full Profile conformance tests.

1.2.1 Architecture overview

The Mali-T604 GPU is formed of four identical cores, each supporting up to 256 concurrently executing (*active*) threads.[4]

Each core contains a tri-pipe containing two arithmetic (A) pipelines, one load-store (LS) pipeline and one texture (T) pipeline. Thus, the peak throughput of each core is two A instruction words, one LS instruction word and one T instruction word per cycle. Midgard is a VLIW (Very Long Instruction Word) architecture, so that each pipe contain multiple units and most instruction words contain instructions for multiple units. In addition, Midgard is a SIMD (Single Instruction Multiple Data) architecture, so that most instructions operate on multiple data elements packed in 128-bit vector registers.

1.2.2 Execution constraints

The architectural maximum number of work-items active on a single core is $\max(I) = 256$. The actual maximum number of active work-items I is determined by the number of registers R that kernel code uses:

$$I = \begin{cases} 256, 0 < R \leq 4 \\ 128, 4 < R \leq 8 \\ 64, 8 < R \leq 16 \end{cases}$$

For example, kernel A that uses $R_A = 5$ registers and kernel B that uses $R_B = 8$ registers can both be executed by *no more than* 128 work-items.[5]

1.2.3 Thread scheduling

The GPU schedules work-groups onto cores in batches, whose size is chosen by the driver depending on the characteristics of the job. The hardware

[4] In what follows, we assume that a single hardware thread executes a single work-item. A program transformation known as *thread coarsening* can result in a single hardware thread executing multiple work-items *e.g.* in different vector lanes.

[5] Therefore, the compiler may prefer to spill a value to memory rather than use an extra register when the number of used registers approaches 4, 8 or 16.

schedules batches onto cores in a round-robin fashion. A batch consists of a number of "adjacent" work-groups.[6]

Each core first creates threads for the first scheduled work-group and then continues to create threads for the other scheduled work-groups until either the maximum number of active threads has been reached or all threads for the scheduled work-groups have been created. When a thread returns, a new thread can be scheduled in its place.

Created threads enter the tripipe in a round-robin order. A core switches between threads on every cycle: when a thread has executed one instruction it then waits while all other threads execute one instruction[7]. Sometimes a thread can stall waiting for a cache miss and another thread will overtake it, changing the ordering between threads. (We will discuss this aspect later in Section 1.4.)

1.2.4 Guidelines for optimising performance

A compute program (kernel) typically consists of a mix of A and LS instruction words.[8] Achieving high performance on the Mali-T604 involves:

- Using a sufficient number of active threads to hide the execution latency of instructions (pipeline depth). The number of active threads depends on the number of registers used by kernel code, so may be limited for complex kernels.

- Using vector operations in kernel code to allow for straightforward mapping to vector instructions by the compiler.

- Having sufficient instruction level parallelism in kernel code to allow for dense packing of instructions into instruction words by the compiler.

- Having a balance between A and LS instruction words. Without cache misses, the ratio of 2:1 of A-words to LS-words would be optimal; with cache misses, a higher ratio is desirable. For example, a kernel consisting of 15 A-words and 7 LS-words is still likely to be bound by the LS-pipe.

[6]In our examples using two-dimensional ND-ranges, two "adjacent" work-groups have work-items that are adjacent in the two-dimensional space of global ids, see Section 1.4.6 for a more detailed description.

[7]There is more parallelism in the hardware than this sentence mentions, but the description here suffices for the current discussion.

[8]The texture (T) pipeline is rarely used for compute kernels, with a notable exception of executing barrier operations (see Section 1.4). The main reason is that using vector instructions in the LS pipeline results in higher memory bandwidth (bytes per cycle) than using instructions in the T pipeline for kernels requiring no sampling.

In several respects, programming for the Mali-T604 GPU embedded on a System-on-Chip (SoC) is easier than programming for desktop class GPUs:

- The `global` and `local` OpenCL address spaces get mapped to the same physical memory (the system RAM), backed by caches transparent to the programmer. This often removes the need for explicit data copying and associated barrier synchronisation.
- All threads have individual program counters. This means that branch divergence is less of an issue than for warp-based architectures.

1.2.5 A note on power

We discuss performance in terms of the time it takes to complete a computation. When analysing the performance of an OpenCL kernel running on a mobile device, it is also important to consider the power and energy required for the execution. Often, the mobile device's thermal dissipation capacity will determine the maximum DVFS operating point (voltage and frequency) that the GPU can be run at. On Mali-T600 GPUs, it is often sufficient to characterise the performance of the *kernel* in terms of the number of cycles required for the execution. To determine the overall performance, one also has to factor in the GPU clock rate.

We focus on the cycle count of kernel execution, and consider this sufficient for our optimisation purposes. We posit that GPU power is (to a broad approximation) constant across sustained GPGPU workloads at a fixed operating point. [9] Energy consumed for a given workload is therefore determined by performance - both the number of cycles for that workload, and the operating point required to meet the required performance target.

[9]This assumption holds broadly on Mali-T6xx series GPUs, as a result of the pipeline architecture, aggressive clock gating, and the GPU's ability to hide memory latency effectively.

1.3 Optimising the Sobel image filter

1.3.1 Algorithm

Our first example is the Sobel 3×3 image filter used within edge detection algorithms. Technically speaking, the Sobel filter is a $(2K+1) \times (2K+1)$ convolution of an input image $\mathbf{I}$ with a constant mask $\mathbf{C}$:

$$\mathbf{O}_{y,x} = \sum_{u=-K}^{K} \sum_{v=-K}^{K} \mathbf{I}_{y+u,x+v} \cdot \mathbf{C}_{u,v},$$

taking an image containing the luminosity values and producing two images containing the discretised gradient values along the horizontal and vertical directions:

$$\mathbf{O}^{\mathbf{dx}}_{y,x} = \sum_{u=-1}^{1} \sum_{v=-1}^{1} \mathbf{I}_{y+u,x+v} \cdot \mathbf{C}^{\mathbf{dx}}_{u,v}, \quad \text{where} \quad \mathbf{C}^{\mathbf{dx}} = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}$$

$$\mathbf{O}^{\mathbf{dy}}_{y,x} = \sum_{u=-1}^{1} \sum_{v=-1}^{1} \mathbf{I}_{y+u,x+v} \cdot \mathbf{C}^{\mathbf{dy}}_{u,v}, \quad \text{where} \quad \mathbf{C}^{\mathbf{dy}} = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix}$$

1.3.2 Implementation details

In our implementation, the input image $\mathbf{I}$ is an $H \times W$ array of unsigned 8-bit integers (`uchar`'s) and the output images $\mathbf{O}^{\mathbf{dx}}$ and $\mathbf{O}^{\mathbf{dy}}$ are $H \times W$ arrays of signed 8-bit integers (`char`'s). The results are computed as signed 16-bit integers (`short`'s) and are normalised by dividing by 8 (*i.e.* shifting right by 3). The results are only computed for $(H-2) \times (W-2)$ inner pixels of the output images, leaving pixels on the one-pixel wide border intact [10].

1.3.3 Performance characteristics

Table 1.1 shows the results for all the kernel versions discussed in this section on a 512×512 input image. We present the following numbers for each kernel version:

- The number of input pixels per work-item.

[10] Our test image has a size of 512×512, which means that the interior has a size of only 510×510. We have vectorized versions that handle 8 or 16 pixels per work-item, and for those implementations, we have an (interior) image size that is not a multiple of the number of pixels per work-item. In a real application, we would have to take care to handle the border separately, but as this affects only a small percentage of the image, we will instead ignore that discussion, as it is not important for our qualitative performance considerations.

- The maximal local work size max(LWS), *i.e.* the maximal number of active work-items per core.
- The number of arithmetic and load-store instruction words per work-item.
- The number of arithmetic and load-store instruction words per cycle.
- The number of pixels processed per cycle.

Achieving a high proportion of the peak numbers of A-words and LS-words executed per cycle (8 and 4, respectively, for the Mali-T604) may be challenging due to memory and other effects.

1.3.4 Using scalars

The `char` kernel in Listing 1.1 is a naïve, scalar version. Each work-item reads a 3×3 square of input pixels around its coordinate. The computation is performed using 16-bit integer arithmetic to avoid overflow. No vector operations are used.

Table 1.1 shows that each work-item has to pass 13 times through an A-pipe, and at least 7 times through a LS-pipe (possibly more due to cache misses). Having 256 active work-items ensures good utilisation: indeed, this version reaches the highest number of LS-words per cycle. However, the overall performance is poor (0.3 pixels per cycle) due to not using vector operations.

1.3.5 Using vectors

Eight Components Each work-item of the `char8` kernel in Listing 1.2 performs 8 `char8` load operations to read a 3×10 region of input pixels, and computes two `char8` vectors of output pixels. The conversion and compute instructions operate on `short8` data in 128-bit registers.

Table 1.1 shows that while the `char8` kernel computes 8 times more data, it performs only about 30% more arithmetic and memory instructions than the `char` kernel, resulting in a 6 times increase in performance (1.8 pixels per cycle). The ratio between the arithmetic and memory instructions is close to the 2:1, optimal for cases with few cache misses. Due to the increase in complexity, the kernel can only be executed with up to 128 work-items. Still, the number of instruction words executed per cycle is nearly the same as for the scalar version.

Sixteen Components The number of operations per pixel is reduced even further by using `char16` memory operations (full vector register width) and `short16` arithmetic operations (broken into `short8` operations by the

```
kernel void sobel_char(
  global const uchar * restrict in, //< Input.
  global char * restrict dx,        //< X gradient.
  global char * restrict dy,        //< Y gradient.
  const int width)
{
  // X and Y gradient accumulators.
  short _dx, _dy;

  // Left, middle and right pixels: loaded as
  // unsigned 8-bit, converted to signed 16-bit.
  uchar lLoad, mLoad, rLoad;
  short lData, mData, rData;

  // Compute (column,row) position and offset.
  const int column = get_global_id(0);
  const int row    = get_global_id(1);
  const int offset = row * width + column;

  // Compute contribution from first row.
  lLoad = *(in + (offset + width * 0 + 0));
  mLoad = *(in + (offset + width * 0 + 1));
  rLoad = *(in + (offset + width * 0 + 2));

  lData = convert_short(lLoad);
  mData = convert_short(mLoad);
  rData = convert_short(rLoad);

  _dx = rData - lData;
  _dy = rData + lData + mData * (short)2;

  // Compute contribution from second row.
  lLoad = *(in + (offset + width * 1 + 0));
  rLoad = *(in + (offset + width * 1 + 2));

  lData = convert_short(lLoad);
  rData = convert_short(rLoad);

  _dx += (rData - lData) * (short)2;

  // Compute contribution from third row.
  lLoad = *(in + (offset + width * 2 + 0));
  mLoad = *(in + (offset + width * 2 + 1));
  rLoad = *(in + (offset + width * 2 + 2));

  lData = convert_short(lLoad);
  mData = convert_short(mLoad);
  rData = convert_short(rLoad);

  _dx += rData - lData;
  _dy -= rData + lData + mData * (short)2;

  // Store the results.
  *(dx + offset + width + 1) = convert_char(_dx >> 3);
  *(dy + offset + width + 1) = convert_char(_dy >> 3);
}
```

Listing 1.1. Initial scalar implementation: `char`.

```
kernel void sobel_char8(
  global const uchar * restrict in, //< Input.
  global char * restrict dx,        //< X gradient.
  global char * restrict dy,        //< Y gradient.
  const int width)
{
  // X and Y gradient accumulators.
  short8 _dx, _dy;

  // Left, middle and right pixels: loaded as
  // unsigned 8-bit, converted to signed 16-bit.
  uchar8 lLoad, mLoad, rLoad;
  short8 lData, mData, rData;

  // Compute (column,row) position and offset.
  const int column = get_global_id(0) * 8;
  const int row    = get_global_id(1) * 1;
  const int offset = row * width + column;

  // Compute contribution from first row.
  lLoad = vload8(0, in + (offset + width * 0 + 0));
  mLoad = vload8(0, in + (offset + width * 0 + 1));
  rLoad = vload8(0, in + (offset + width * 0 + 2));

  lData = convert_short8(lLoad);
  mData = convert_short8(mLoad);
  rData = convert_short8(rLoad);

  _dx = rData - lData;
  _dy = rData + lData + mData * (short8)2;

  // Compute contribution from second row.
  lLoad = vload8(0, in + (offset + width * 1 + 0));
  rLoad = vload8(0, in + (offset + width * 1 + 2));

  lData = convert_short8(lLoad);
  rData = convert_short8(rLoad);

  _dx += (rData - lData) * (short8)2;

  // Compute contribution from third row.
  lLoad = vload8(0, in + (offset + width * 2 + 0));
  mLoad = vload8(0, in + (offset + width * 2 + 1));
  rLoad = vload8(0, in + (offset + width * 2 + 2));

  lData = convert_short8(lLoad);
  mData = convert_short8(mLoad);
  rData = convert_short8(rLoad);

  _dx += rData - lData;
  _dy -= rData + lData + mData * (short8)2;

  // Store the results.
  vstore8(convert_char8(_dx >> 3), 0, dx + offset + width + 1);
  vstore8(convert_char8(_dy >> 3), 0, dy + offset + width + 1);
}
```

Listing 1.2. Initial vector implementation: `char8`.

```
// Compute contribution from third row.
lLoad = vload16(0, in + (offset + width * 2 + 0));
mLoad = vload16(0, in + (offset + width * 2 + 1));
rLoad = vload16(0, in + (offset + width * 2 + 2));

lData = convert_short16(lLoad);
mData = convert_short16(mLoad);
rData = convert_short16(rLoad);

_dx += rData - lData;
_dy -= rData + lData + mData * (short16)2;

// Store the results.
vstore16(convert_char16(_dx >> 3), 0, dx + offset + width + 1);
vstore16(convert_char16(_dy >> 3), 0, dy + offset + width + 1);
```

Listing 1.3. Computing contribution from the third row: `char16`.

```
// Compute contribution from third row.
load = vload16(0, in + (offset + width * 2 + 0));

lData = convert_short8(load.s01234567);
mData = convert_short8(load.s12345678);
rData = convert_short8(load.s23456789);

_dx += rData - lData;
_dy -= rData + lData + mData * (short8)2;

// Store the results.
vstore8(convert_char8(_dx >> 3), 0, dx + offset + width + 1);
vstore8(convert_char8(_dy >> 3), 0, dy + offset + width + 1);
```

Listing 1.4. Computing contribution from the third row: `char8_load16`.

compiler) in the `char16` kernel partially shown in Listing 1.3. Performance, however, does not increase, because this kernel can only be executed with up to 64 work-items, which introduces bubbles into the pipelines due to reduced latency-hiding capability.

1.3.6 Reusing loaded data

Larger load operations The `char8` kernel performed 8 `char8` load operations. The `char8_load16` kernel, partially shown in Listing 1.4, performs only 3 `char16` load operations: the required subcomponents are extracted by swizzle operations which are often free on the Midgard architecture. Table 1.1 confirms that the number of memory operations per pixel is decreased, while still allowing the kernel to be launched with up to 128 work-items.

```
// Compute contribution from third row.
lLoad = vload16(0, in + (offset + width * 2 + 0));
rLoad = vload16(0, in + (offset + width * 2 + 2));

lData = convert_short16(lLoad);
mData = convert_short16(
  (uchar16)(lLoad.s12345678, rLoad.s789abcde));
rData = convert_short16(rLoad);

_dx += rData - lData;
_dy -= rData + lData + mData * (short16)2;

// Store the results.
vstore16(convert_char16(_dx >> 3), 0, dx + offset + width + 1);
vstore16(convert_char16(_dy >> 3), 0, dy + offset + width + 1);
```

Listing 1.5. Computing contribution from the third row: `char16_swizzle`.

```
// Compute contribution from third row.
lLoad = vload8(0, in + (offset + width*2 + 0));
mLoad = vload8(0, in + (offset + width*2 + 1));
rLoad = vload8(0, in + (offset + width*2 + 2));

lData = convert_short8(lLoad);
mData = convert_short8(mLoad);
rData = convert_short8(rLoad);

_dx1 += rData - lData;
_dy1 -= rData + lData + mData * (short8)2;
_dx2 += (rData - lData) * (short8)2;
```

```
// Store the results.
vstore8(convert_char8(_dx1 >> 3), 0, dx1 + offset + width + 1);
vstore8(convert_char8(_dy1 >> 3), 0, dy1 + offset + width + 1);
vstore8(convert_char8(_dx2 >> 3), 0, dx2 + offset + width*2 + 1);
vstore8(convert_char8(_dy2 >> 3), 0, dy2 + offset + width*2 + 1);
```

Listing 1.6. Computing contribution from the third row: `2xchar8`.

Eliminating redundant loads The `char16` kernel performed 3 `char16` load operations to read 18 bytes for the first and third rows. The `char16_swizzle` kernel, partially shown in Listing 1.5, performs 2 `char16` load operations for the leftmost and rightmost vectors, and reconstructs the middle vector by swizzle operations.

While the number of instruction words increases in comparison to the `char16` kernel (by nearly 50% for A-words and even slightly for LS-words due to instruction scheduling effects), this kernel can be launched with 128 work-items. This leads to the highest utilisation of the A-pipes of all the versions in this study, as well as the best overall performance.

```
// Compute contribution from third row.
load = vload16(0, in + (offset + width*2 + 0));

lData = convert_short8(load.s01234567);
mData = convert_short8(load.s12345678);
rData = convert_short8(load.s23456789);

_dx1 += rData - lData;
_dy1 -= rData + lData + mData * (short8)2;
_dx2 += (rData - lData) * (short8)2;
```

```
// Store the results.
vstore8(convert_char8(_dx1 >> 3), 0, dx1 + offset + width + 1);
vstore8(convert_char8(_dy1 >> 3), 0, dy1 + offset + width + 1);
vstore8(convert_char8(_dx2 >> 3), 0, dx2 + offset + width*2 + 1);
vstore8(convert_char8(_dy2 >> 3), 0, dy2 + offset + width*2 + 1);
```

Listing 1.7. Computing contribution from the third row: `2xchar8_load16`.

```
// Compute contribution from third row.
lLoad = vload8(0, in + (offset + width*2 + 0));
mLoad = vload8(0, in + (offset + width*2 + 1));
rLoad = vload8(0, in + (offset + width*2 + 2));

lData = convert_short8(lLoad);
mData = convert_short8(mLoad);
rData = convert_short8(rLoad);

_dx1 += rData - lData;
_dy1 -= rData + lData + mData * (short8)2;
_dx2 += (rData - lData) * (short8)2;
_dx3 = rData - lData;
_dy3 = rData + lData + mData * (short8)2;
```

```
// Store the results.
vstore8(convert_char8(_dx1 >> 3), 0, dx1 + offset + width + 1);
vstore8(convert_char8(_dy1 >> 3), 0, dy1 + offset + width + 1);
vstore8(convert_char8(_dx2 >> 3), 0, dx2 + offset + width*2 + 1);
vstore8(convert_char8(_dy2 >> 3), 0, dy2 + offset + width*2 + 1);
vstore8(convert_char8(_dx3 >> 3), 0, dx3 + offset + width*3 + 1);
vstore8(convert_char8(_dy3 >> 3), 0, dy3 + offset + width*3 + 1);
```

Listing 1.8. Computing contribution from the third row: `3xchar8`.

1.3.7 Processing multiple rows

The kernels presented so far have loaded pixels from three input rows to compute pixels in a single output row. In general, to compute pixels in n output rows, $n+2$ input rows are needed. In the extreme case of n being the output image height, the number of memory accesses is reduced by

Version name	Pixels/WI	max(LWS)	A-words/WI	LS-words/WI	A-words/cycle	LS-words/cycle	Pixels/cycle
`char`	1	256	13	7	3.9	2.1	0.3
`char8`	8	128	17	9	3.8	2	1.8
`char16`	16	64	21	7	2.3	0.8	1.8
`char8_load16`	8	128	18	5	5	1.4	2.2
`char16_swizzle`	16	128	32	8	5.1	1.3	2.6
`2xchar8`	16	128	29	14	3.8	1.9	2.1
`2xchar8_load16`	16	128	32	7	4.9	1.1	2.5
`3xchar8`	24	64	39	19	2.2	1.1	1.4

Table 1.1. Performance characteristics of all Sobel versions running on a 512×512 image (see Section 1.3.3 for a description of the columns). Note that our implementations compute two output pixels for each input pixel (*i.e.* double the output of a standard 3×3 convolution filter), but we only count the number of input pixels per work-item (WI).

nearly a factor of 3, which can bring significant improvements for memory bound kernels.

Computing two rows of output The `2xchar8` and `2xchar8_load16` kernels load from four input rows to compute results for two output rows ($n = 2$). They are partially shown in Listing 1.6 and Listing 1.7, and are modifications of the `char8` and `char8_load16` kernels, respectively. Both kernels can be launched with up to 128 work-items and both kernels perform better than the single-row variants. As before, the `load16` version is faster, and indeed achieves the second best performance in this study.

Computing three rows of output The `3xchar8` kernel, partially shown in Listing 1.8, has grown too complex and can only be launched with up to 64 work-items. Therefore, exploiting data reuse by keeping more than two rows in registers is suboptimal on the Mali-T604.

1.3.8 Summary

We have presented several versions of the Sobel filter, and discussed their performance characteristics on the Mali-T604 GPU. Vectorising kernel code and exploiting data reuse are the two principal optimisation techniques explored in this study. The fastest kernel, `char16_swizzle`, is nearly 9 times faster than the slowest kernel, `char`, which reiterates the importance of target-specific optimisations for OpenCL code.

To summarise, we note that although the theoretical peak performance is at the ratio of 2 arithmetic instruction words for every load-store instruction word, the best performance was obtained in the versions with the highest number of arithmetic words executed per cycle. Restructuring the program to trade load-store operations for arithmetic operations has thus been successful, as long as the kernel could still be launched with 128 work-items.

1.4 Optimising the general matrix multiplication

The Sobel filter implementations have hightlighted the importance of using vector instructions and a high number of active work-items. We next study implementations of the general matrix multiply (GEMM) to elucidate the importance of using caches effectively. We first discuss aspects of the caches and how we optimize for them. At the end, we look at the runtimes on an Arndale development board and compare to our discussions.

1.4.1 Algorithm

The General Matrix Multiplication is a function of the Basic Linear Algebra Subprograms (BLAS) API[11] which computes:

$$C = \alpha AB + \beta C$$

where A, B, C are matrices of floating-point numbers and α, β are scalars.

1.4.2 Implementation details

In our implementation, the matrices are $N \times N$ arrays of single-precision floating-point numbers (SGEMM). We consider two common SGEMM variants:

- NN: A is non-transposed, B is non-transposed:

$$C[i,j] = \alpha \sum_{k=0}^{N-1} A[i,k] \times B[k,j] + \beta C[i,j]$$

- NT: A is non-transposed, B is transposed:

$$C[i,j] = \alpha \sum_{k=0}^{N-1} A[i,k] \times B[j,k] + \beta C[i,j]$$

where $i = 0, \ldots, N-1$ and $j = 0, \ldots, N-1$.

CPU implementations of the NN variant often first transpose B and then perform the NT variant, which has a more cache-friendly memory access pattern, as we show in Section 1.4.4.

[11] http://www.netlib.org/blas

```
kernel void
sgemm(global float const *A, global float const *B,
      global float *C, float alpha, float beta, uint n)
{
  uint j = get_global_id(0);
  uint i = get_global_id(1);

  float ABij = 0.0f;
  for (uint k = 0; k < n; ++k)
  {
    ABij += A[i*n + k] * B[k*n + j];
  }
  C[i*n + j] = alpha * ABij + beta * C[i*n + j];
}
```

Listing 1.9. Initial scalar implementation: `scalarNN`.

```
kernel void
sgemm(global float const *A, global float const *B,
      global float *C, float alpha, float beta, uint n)
{
  uint j = get_global_id(0);
  uint i = get_global_id(1);

  float ABij = 0.0f;
  for (uint k = 0; k < n; ++k)
  {
    ABij += A[i*n + k] * B[j*n + k];
  }
  C[i*n + j] = alpha * ABij + beta * C[i*n + j];
}
```

Listing 1.10. Initial scalar implementation: `scalarNT`.

1.4.3 Scalar implementations

We first consider scalar implementations with an $N \times N$ ND-range covering all elements of C. From our experience with optimising the Sobel filter, these versions are clearly suboptimal as they do not use any vector operations. We will (due to their simplicity) use them to introduce the notation for describing memory access patterns of kernels, and we will also use them as examples in some qualitative discussions later.

Non-Transposed Each work-item of the `scalarNN` version in Listing 1.9 produces one element of C by computing the dot-product of a row of A and a column of B.

Transposed Each work-item of the `scalarNT` version in Listing 1.10 produces one element of C by computing the dot-product of a row of A and a column of B^T (or equivalently a row of B).

1.4.4 Memory access patterns of scalar implementations

A single work-item of the `scalarNN` version sequentially reads (within the `k` loop) from pairs of locations $(A[i,0], B[0,j]), (A[i,1], B[1,j]), \ldots, (A[i, N-1], B[N-1, j])$. We will abbreviate this access pattern to:

$$\mathop{\Large ,}\limits_{k=0}^{N-1}(A[i,k], B[k,j])$$

which denotes that the accesses happen sequentially for $0 \leq k < N$.

Similarly, the access pattern of a single work-item of the `scalarNT` version is:

$$\mathop{\Large ,}\limits_{k=0}^{N-1}(A[i,k], B[j,k])$$

With the row-major array layout used in the C language, the `scalarNT` variant reads both A and B with stride 1, while the `scalarNN` variant reads B with stride N.

Let us assume a core executes a single work-group of dimensions (λ_0, λ_1). Since work-items execute in an interleaved order (Section 1.2.3), the actual memory access pattern of the `scalarNN` variant on the core will be:

$$\mathop{\Large ,}\limits_{k=0}^{N-1}\left[\left(\mathop{\Large ,}\limits_{i=0}^{\lambda_1-1}\mathop{\Large ,}\limits_{j=0}^{\lambda_0-1} A[i,k]\right) \mathop{\Large ,} \left(\mathop{\Large ,}\limits_{i=0}^{\lambda_1-1}\mathop{\Large ,}\limits_{j=0}^{\lambda_0-1} B[k,j]\right)\right]$$

which means that on each iteration $0 \leq k < N$ all work-items first read from A and then from B. The $\mathop{\Large ,}$ operator for $0 \leq j < \lambda_0$ is the innermost one, due to the order in which threads are created by the device[12].

1.4.5 Blocking

As a first step towards better implementations, we will introduce *blocking*, a program transformation that will allow us to use vector operations for memory accesses and arithmetic operations. We will later exploit blocking to improve cache usage.

Let us assume that matrix order N is divisible by blocking factors ΔI, ΔJ and ΔK. Imagine that:

- matrix A consists of $\frac{N}{\Delta I} \times \frac{N}{\Delta K}$ submatrices of $\Delta I \times \Delta K$ elements each,

[12]The GPU increments the id associated with λ_0 as the innermost index, and by assigning `get_global_id(0)` to j, we have chosen j as the innermost index in our implementation. This choice will be discussed in Section 1.4.6

- matrix B consists of $\frac{N}{\Delta K} \times \frac{N}{\Delta J}$ submatrices of $\Delta K \times \Delta J$ elements each, and
- matrix C consists of $\frac{N}{\Delta I} \times \frac{N}{\Delta J}$ submatrices of $\Delta I \times \Delta J$ elements each.

Now, instead of writing the SGEMM NN algorithm as:

$$C[i, j] = \alpha \sum_{i,j} A[i, k] \times B[k, j] + \beta C[i, j]$$

where $A[i, k]$, $B[k, j]$ and $C[i, j]$ were individual elements, we can write it as:

$$C[I, J] = \alpha \sum_{I,J} A[I, K] \times B[K, J] + \beta C[I, J]$$

where $A[I, K]$, $B[K, J]$ and $C[I, J]$ are submatrices as above.

Our kernels will still have the same structure, but each work-item will now compute one $\Delta I \times \Delta J$ submatrix of $C[I, J]$. Each iteration of the k loop will multiply a $\Delta I \times \Delta K$ matrix by a $\Delta K \times \Delta J$ matrix.

For the NN variant, where both A and B use normal matrix layout, we implement the matrix multiplication between a 1×4 block from A and the 4×4 block from B as[13]

```
float4 a =  A[i,   k];
float4 b0 = B[k+0, j];
float4 b1 = B[k+1, j];
float4 b2 = B[k+2, j];
float4 b3 = B[k+3, j];
ab += a.s0*b0 + a.s1*b1 + a.s2*b2 + a.s3*b3;
```

where `ab` (of type `float4`) is the accumulator for the 1×4 block of C and all operations are vector operations. The kernel is shown in Listing 1.11.

For the NT variant, we instead select ($\Delta I = 2$, $\Delta J = 2$, $\Delta K = 4$) and implement the multiplication between the 2×4 block of A and the 2×4 block of the transposed B as

```
float4 a0 = A[i  , k];
float4 a1 = A[i+1, k];
float4 b0 = B[j  , k];
float4 b1 = B[j+1, k];
ab.s01 += (float2) (dot(a0, b0), dot(a0, b1));
ab.s23 += (float2) (dot(a1, b0), dot(a1, b1));
```

where `ab` is an accumulator variable of type `float4` for the 2×2 block of the matrix C[14]. The full kernel is shown in Listing 1.12.

[13] In code snippets such as these, we gloss over some details, such as the fact that we need $4k$ instead of k as the offset in the reads from B.

[14] The components `ab.s01` accumulate the top row and the components `ab.s23` accumulate the bottom row of the 2×2 block.

```
kernel void
sgemm(global float4 const *A, global float4 const *B,
      global float4 *C, float alpha, float beta, uint n)
{
  uint j = get_global_id(0);
  uint i = get_global_id(1);
  uint nv4 = n >> 2;

  float4 accum = (float4) 0.0f;
  for (uint k = 0; k < nv4; ++k)
  {
    float4 a = A[i*nv4 + k];

    float4 b0 = B[(4*k+0)*nv4 + j];
    float4 b1 = B[(4*k+1)*nv4 + j];
    float4 b2 = B[(4*k+2)*nv4 + j];
    float4 b3 = B[(4*k+3)*nv4 + j];

    accum += a.s0*b0+a.s1*b1+a.s2*b2+a.s3*b3;
  }
  C[i*nv4 + j] = alpha * accum + beta * C[i*nv4 + j];
}
```

Listing 1.11. Vectorised implementation: `blockedNN`.

We saw the need to introduce blocking to enable the use of vector operations, but register blocking also decreases the number of loads necessary. Our scalar implementations loaded (both the NN and NT variants) N elements of A and N elements of B to compute 1 element of C, so we needed to load $(N + N)N^2 = 2N^3$ elements from A and B. in general, we need to load one $\Delta I \times \Delta K$ block from A and one $\Delta K \times \Delta J$ block from B per iteration, and we need $N/\Delta K$ iterations. We need one work-item for each of the $(N/\Delta I)(N/\Delta J)$ blocks in C, which gives us a total of

$$(\Delta I \Delta K + \Delta K \Delta J)\frac{N}{\Delta K}\frac{N}{\Delta I}\frac{N}{\Delta J} = N^3\left(\frac{1}{\Delta J} + \frac{1}{\Delta I}\right)$$

elements to be loaded into registers.

The above result tells us that we should want to choose ΔI and ΔJ large and similar, while the choice of ΔK is less important. We always set ΔK to 4, as this is the smallest value that allows us to use vector operations.

In NN implementations, we have to also choose ΔJ as a multiple of 4, to allow for the use of vector operations, whereas $\Delta I = \Delta J = 2$ is one option we may choose in the NT case. We can compute the difference in load requirements between the $1 \times 4 \times 4$ and $2 \times 4 \times 2$ implementations[15]

[15]We will sometimes refer to a blocking by writing it $\Delta I \times \Delta K \times \Delta J$.

```
kernel void
sgemm(global float4 * const A, global float4 * const B,
    global float2 *C, float alpha, float beta, uint n)
{
  uint i = get_global_id(0);
  uint j = get_global_id(1);
  uint nv4 = n >> 2;

  float4 ab = (float4)(0.0f);
  for (uint k = 0; k < nv4; ++k)
  {
    float4 a0 = A[ 2*i   *nv4 + k];
    float4 a1 = A[(2*i+1)*nv4 + k];

    float4 b0 = B[ 2*j   *nv4 + k];
    float4 b1 = B[(2*j+1)*nv4 + k];

    ab += (float4)(dot(a0, b0), dot(a0, b1),
                   dot(a1, b0), dot(a1, b1));

  }
  uint ix = 2*i*(n>>1) + j;
  C[ix]           = alpha * ab.s01 + beta * C[ix];
  C[ix + (n>>1)] = alpha * ab.s23 + beta * C[ix + (n>>1)];
}
```

Listing 1.12. Vectorised implementation: `blockedNT`.

by computing $1/\Delta I + 1/\Delta J$ for them, and we find 1.25 and 1, respectively, showing that the second blocking needs fewer load operations. We see that the added flexibility we are given to choose blocking widths in the NT version can help us decrease the number of memory operations, and this is one reason to expect that the NT variants will perform better on the GPU.

In both cases, however, we perform $O(N^3)$ load operations from data of size $O(N^2)$, which means that each datum is reloaded into registers $O(N)$ times. Effective cache usage will clearly be important, as it allows us to access data from caches rather than from main memory.

1.4.6 L1 Cache analysis of the $1 \times 4 \times 4$ blocked NN SGEMM

Overview To estimate the cache usage of our kernels, we have to take into account the order of reads within a work-item, as well as the fact that many work-items are active at the same time. We will first look at one specific choice of program and local work size, and then try to extend the analysis to be able to compare the cache usage of different implementations and local work sizes.

In the program we choose to analyse first, the $1 \times 4 \times 4$ blocked NN implementation, every work-item performs 5 memory operations per iteration in its loop, and we will assume that the memory operations take place in the same order as they appear in the program (*i.e.* the compiler does not change their order),and that, for a given work-item, memory operations never execute in parallel. Another restriction that we will also make in our analysis, and which is important, is that we are able to perfectly predict the order in which work-items execute on the GPU. Finally, this section will only focus on the L1 cache, which allows us to restrict the analysis to a single core, as each core has its own L1 cache.

Thread order A single work-item loops over the variable k, and for every value of k, it performs one memory load from A and 4 memory loads from B. With our assumptions, we know that the work-item[16] will enter the load-store pipeline once for each of those instructions, in the order they appear in the program source. We also know that we schedule one work-group of work-items at a time and that those work-items execute their memory operations in an interleaved fashion one after the other, and that they always do this in the order they were spawned by the GPU. We will now see what how we can use that knowledge to analyse the L1 data cache that we need to use.

Fixed local work size Using our notation introduced previously, work-item (j, i) performs the following reads in loop iteration k

$$A[i, k], B[4k + 0, j], B[4k + 1, j], B[4k + 2, j], B[4k + 3, j]$$

where each memory access now loads a `float4` vector. With many active threads, we will first see all threads performing their first reads from A, and thereafter we will see all threads performing their first read from B, *etc.*. This implies that reads that are executed after each other correspond to different thread executing the same instruction in the program code. With a local work size of (4,32), the GPU initiates the work-items for work-group (m, n) by incrementing the first index first, *i.e.* in the order

$$\begin{aligned}&(4m, 32n), (4m + 1, 32n), (4m + 2, 32n), (4m + 3, 32n),\\&(4m, 32n + 1), (4m + 1, 32n + 1), (4m + 2, 32n + 1), (4m + 3, 32n + 1),\\&(4m, 32n + 2), (4m + 1, 32n + 2), (4m + 2, 32n + 2), (4m + 3, 32n + 2),\\&\dots,\\&(4m, 32n + 31), (4m + 1, 32n + 31), (4m + 2, 32n + 31), (4m + 3, 32n + 31),\end{aligned}$$

where we have again used the comma as a sequencing operation to describe the ordering of `global_id` values of the work-items.

[16]From a hardware point of view, we of course discuss the behaviour and order of threads, but we continue to use the term work-item, remembering that one work-item in OpenCL corresponds to one thread in the GPU.

This means that the memory reads for loop iteration k will execute in the following order

$$\begin{aligned}
&\left(\mathop{\Large ,}\limits_{i=32n}^{32n+31}\mathop{\Large ,}\limits_{j=4m}^{4m+3} A[i,k]\right),\\
&\left(\mathop{\Large ,}\limits_{i=32n}^{32n+31}\mathop{\Large ,}\limits_{j=4m}^{4m+3} B[4k+0,j]\right),\left(\mathop{\Large ,}\limits_{i=32n}^{32n+31}\mathop{\Large ,}\limits_{j=4m}^{4m+3} B[4k+1,j]\right),\\
&\left(\mathop{\Large ,}\limits_{i=32n}^{32n+31}\mathop{\Large ,}\limits_{j=4m}^{4m+3} B[4k+2,j]\right),\left(\mathop{\Large ,}\limits_{i=32n}^{32n+31}\mathop{\Large ,}\limits_{j=4m}^{4m+3} B[4k+3,j]\right)
\end{aligned}$$

where the id variable j is incremented before i as it corresponds to `get_global_id(0)`, and it is therefore written as the innermost $\mathop{\Large ,}$ operator.

We see that the reads from A do not depend on j and are therefore repeated for each group of 4 consecutive work-items, we introduce the $\times$ operation to reflect repetition of the same memory access as in

$$\mathop{\Large ,}\limits_{j=4m}^{4m+3} A[i,k] = A[i,k]\times 4$$

This notation allows us to write a single iteration over k as

$$\begin{aligned}
&\left(\mathop{\Large ,}\limits_{i=32n}^{32n+31} A[i,k]\times 4\right),\left(\mathop{\Large ,}\limits_{j=4m}^{4m+3} B[4k+0,j]\right)\times 32,\left(\mathop{\Large ,}\limits_{j=4m}^{4m+3} B[4k+1,j]\right)\times 32,\\
&\left(\mathop{\Large ,}\limits_{j=4m}^{4m+3} B[4k+2,j]\right)\times 32,\left(\mathop{\Large ,}\limits_{j=4m}^{4m+3} B[4k+3,j]\right)\times 32
\end{aligned}$$

As a cache-line has space for 4 `float4` elements, we see that the reads from A read the first quarter of 32 consecutive cache lines, and the reads from B read 4 full cache lines. To get full cache lines instead, we consider four consecutive iterations in k together, and see that those 4 iterations read 32 full cache lines from A and 16 full cache lines from B. For the moment, we restrict ourselves to considering a single work-group, and we note that these cache lines will never be reused by later operations in the same work-group. We have now arrived at our conclusion for the L1 cache requirements of the loop. If our L1 cache has enough space for 48 cache lines, then we will never read the same value into the L1 cache twice while executing the loop for all work-items in a work-group, as all subsequent uses will be able to reuse the value that is stored in the cache.

After the loop has completed, the work-group additionally has to load and store to C, which needs access to a 32×4 block of 1×4 blocks of C, spanning 32 complete cache lines, meaning that (as long as our L1 cache has at least 32 cache lines large) we will not see any lack of reuse for the elements of C within a work-group.

If we continue to assume that we only have a single work-group at a time, and consider the possibilities for cache reuse between consecutively scheduled work-groups on the same core, we need to consider the state of the L1 cache when work-group (n, m) finishes execution. The L1 cache contains 256 cache lines in total, and the operations on C will have filled 32 of those, so 224 remain for A and B. Each sequence of 4 iteration needs 48 cache lines, so the number of iterations that have their cache lines still in cache at work-group completion is four times $(256 - 32)/48$, or 16, and this lets us see that the work-group size where we may have reuse between work-groups is when we only need 16 sequences of four-iterations each in the loop, or 64 iterations, which corresponds to a matrix size of 256×256 (as we chose $\Delta K = 4$). For larger matrices, no reuse between consecutively scheduled work-groups on the same core is possible.

Arbitrary local work size For an arbitrary local work size, we have two reasons to redo the above analysis. First, we have the obvious reason that we get a different interleaved read pattern between the work-items within a work-group. Second, we can have more than one work-group simultaneously active on the same core, if we choose a smaller local work size.

With a local work size of (λ_0, λ_1), we need to look at all work-groups that are running simultaneously on the same core. If the total number of work-items on the core is 128 (which seemed optimal in the Sobel study), and if $\lambda_0\lambda_1 = 128$, then we have only a single work-group on the core, but we could have chosen $\lambda_0 = \lambda_1 = 4$, which would give us 128/16=8 work-groups executing simultaneously on a core. As before, it will be beneficial to look at the cache usage over four iterations over k, and we can easily generalise the results we had before to see that a single work-group reads λ_1 full cache lines from A and $4\lambda_0$ full cache lines from B for every 4 iterations (provided that λ_0 is a multiple of 4).

If $\lambda_0\lambda_1 <= 64$, we have more than one work-group executing simultaneously on the core. In this case, the work-groups that are simultaneously active on a code will have coonsecutive values of m, and identical values of n. We see that the reads from A read from the same cache lines, so they are reused between the work-groups. We said above that a few work-groups are sent to each core, and we assume that the work-groups we are having active at the same time belong to this set, as we would otherwise not have consecutive group-ids (m, n)[17].

[17] With work-group divergence *i.e.* with a few work-items each from many work-groups partially finished on the same core, we might have work-groups with very different `group_ids` simultaneously active on the same core.

This means that the 128 work-items executing simultaneously on one core use

$$\lambda_1 + 4\lambda_0\,[\text{number of work groups}] = \lambda_1 + 4\lambda_0 \frac{128}{\lambda_0\lambda_1} = \lambda_1 + 512/\lambda_1$$

cache lines from the L1 cache for four consecutive iterations in k. As this expression is independent of λ_0, we can select our λ_0 freely (as long as it is a multiple of 4), and the only effect we see (from our analysis so far) is that a larger λ_0 restricts our possible choices for λ_1. With $\lambda_1 = 1, 2, 4, 8, 16, 32, 64, 128$, we see that we require $513, 258, 132, 72, 48, 48, 72, 132$ cache lines, and with room for 256 lines in the L1 cache of each core, the fraction of L1 we need to use is[18] $2.0, 1.0, 0.52, 0.28, 0.19, 0.19, 0.28, 0.52$, and under our assumptions of fully associative cache and perfect execution order between work-items, we would expect all options with a value below 1 to have the same performance (disregarding effects of L2 and RAM).

As we know that our assumptions are incorrect, though, we need to discuss what happens when executing on a real GPU. First, due to the design of the cache, we will see cache misses before the cache is 100% filled, *i.e.* earlier than our analysis above would have predicted. The more complicated aspect of execution is that the work-items that are spawned in the order we describe here do not keep the order. When one work-item is stalled on a cache miss, other work-items may overtake it, so we will have active work-items that are executing different iterations (different values of k) at the same time. We refer to this as thread divergence (or work-item divergence), and the fraction of L1 we need is a measure of our robustness to keep having good performance in cases of thread divergence. Thread divergence always happens, and is difficult to measure and quantify, but we can qualitatively say that the longer the program runs, the stronger divergence we see. This implies that the benefits of a low L1 cache utilization fraction has a stronger impact on performance for large matrix sizes.

The innermost index Now that we have discussed cache usage, we will return to one aspect of the NN variant that we did not discuss before, namely the choice to use j as the innermost index in the sequencing of operations. We have chosen the order by by using `global_id(0)` for the index j and `global_id(1)` for the index i in our CL C kernels. The argument we will provide here holds only for the NN versions, and is not applicable for the NT versions.

If we again consider the memory read sequence for the scalar NN variant, and only look at a single iteration and a single work-group we see

[18]The number are also shown in Table 1.2.

$$\begin{aligned} & \left(\sum_{i=0}^{\lambda_1-1}\sum_{j=0}^{\lambda_0-1} A[i,k]\right), \left(\sum_{i=0}^{\lambda_1-1}\sum_{j=0}^{\lambda_0-1} B[k,j]\right) \\ = \quad & \left(\sum_{i=0}^{\lambda_1-1} A[i,k] \times \lambda_0\right), \left(\sum_{j=0}^{\lambda_0-1} B[k,j]\right) \times \lambda_1 \end{aligned}$$

but we could instead have swapped the roles of the `global_id`s, and gotten

$$\begin{aligned} & \left(\sum_{j=0}^{\lambda_1-1}\sum_{i=0}^{\lambda_0-1} A[i,k]\right), \left(\sum_{j=0}^{\lambda_1-1}\sum_{i=0}^{\lambda_0-1} B[k,j]\right) \\ = \quad & \left(\sum_{i=0}^{\lambda_0-1} A[i,k]\right) \times \lambda_1, \left(\sum_{j=0}^{\lambda_1-1} B[k,j] \times \lambda_0\right) \end{aligned}$$

We will now look for the number of cache-line switches in the execution of those reads. For the former case, the reads from A are described by $\sum_{i=0}^{\lambda_1-1} A[i,k] \times \lambda_0$ which switches cache line λ_1 times (and between those witches, we read from the same matrix element λ_0 times). For different values of i we read from different rows of the matrix, and separate rows are always on separate cache lines[19]. Staying with the first version, but looking at B, we see that we switch matrix element $\lambda_0\lambda_1$ times, as every read is from a different element than the previous read, and we perform $\lambda_0\lambda_1$ reads. We note, however, that we switch between consecutive elements in memory, which means that we only have $\lambda_0\lambda_1/16$ cache line switches, as we can fit 16 matrix elements into a cache line. For the second version, with the indices in the other order, analogous considerations show that we switch matrix element $\lambda_0\lambda_1$ times for A and λ_1 times for B, but we again step between consecutive memory locations in B, which means that the number of cache-line switches are only $\lambda_1/16$ for B. If we add the cache line switches together for A and B, we have $\lambda_0\lambda_1/16 + \lambda_1$ and $\lambda_0\lambda_1 + \lambda_1/16$ for the two versions, respectively. With λ_0 and λ_1 between 4 and 64, the first version will always need fewer cache-line switches than the latter[20]. In cases where we do have cache misses (*e.g.* due to thread-divergence), this should improve the performance by reducing the concurrent cache needs within an iteration.

[19]Except for very small matrices, of course, but cache analyses are unimportant to those cases anyway.

[20]The roles of λ_0 and λ_1 is interchanged between the two versions, but the first one is still always better

1.4.7 L1 cache analysis of blocked NT kernel

For the blocked NT kernel, we can analyse the L1 cache utilisation in the same way as for the NN kernel. We start by noting that work-item (j, i), in iteration k, performs the memory accesses

$$A[2i, k], A[2i + 1, k], B[2j, k], B[2j + 1, k]$$

and we see that we should again consider 4 iterations over k, to get full cache lines

$$\begin{gathered}
A[2i, k + 0], A[2i + 1, k + 0], B[j, k + 0], B[2j + 1, k + 0]\\
A[2i, k + 1], A[2i + 1, k + 1], B[j, k + 1], B[2j + 1, k + 1]\\
A[2i, k + 2], A[2i + 1, k + 2], B[j, k + 2], B[2j + 1, k + 2]\\
A[2i, k + 3], A[2i + 1, k + 3], B[j, k + 3], B[2j + 1, k + 3]
\end{gathered}$$

During execution of the first four iterations, the first work-group, with its $\lambda_0\lambda_1$ work-items, accesses

$$\sum_{k=0}^{4}\left[\left(\sum_{i=0}^{\lambda_1-1}\sum_{j=0}^{\lambda_0-1} A[2i, k]\right), \left(\sum_{i=0}^{\lambda_1-1}\sum_{j=0}^{\lambda_0-1} A[2i + 1, k]\right),\right.$$
$$\left.\left(\sum_{i=0}^{\lambda_1-1}\sum_{j=0}^{\lambda_0-1} B[2j, k]\right), \left(\sum_{i=0}^{\lambda_1-1}\sum_{j=0}^{\lambda_0-1} B[2j + 1, k]\right)\right]$$

and the first set of M simultaneous work-groups, accessses

$$\sum_{k=0}^{4}\left[\left(\sum_{m=0}^{M-1}\sum_{i=0}^{\lambda_1-1}\sum_{j=m\lambda_0}^{m\lambda_0+\lambda_0-1} A[2i, k]\right), \left(\sum_{m=0}^{M-1}\sum_{i=0}^{\lambda_1-1}\sum_{j=m\lambda_0}^{m\lambda_0+\lambda_0-1} A[2i + 1, k]\right),\right.$$
$$\left.\left(\sum_{m=0}^{M-1}\sum_{i=0}^{\lambda_1-1}\sum_{j=m\lambda_0}^{m\lambda_0+\lambda_0-1} B[2j, k]\right), \left(\sum_{m=0}^{M-1}\sum_{i=0}^{\lambda_1-1}\sum_{j=m\lambda_0}^{m\lambda_0+\lambda_0-1} B[2j + 1, k]\right)\right]$$

where m was incremented as an outer index to both i and j, as we create all work-items in the first work-group, before creating the first work-item in the second work-group. We again share the accesses to A, and these four iterations over k will need $2\lambda_1$ cache lines from A and $2M\lambda_0$ cache lines from B, and as before, we have $M = 128/(\lambda_0\lambda_1)$, giving a total L1 usage of

$$2\lambda_1 + 2\lambda_0\frac{128}{\lambda_0\lambda_1} = 2\lambda_1 + \frac{256}{\lambda_1}$$

If we compare with the result of the NN implementation, we get the L1 utilisation fractions shown in Table 1.2. By comparing with the results with the previous ones, we see that while we had a preference for $\lambda_1 = 16$ or $\lambda_1 = 32$ for the $1 \times 4 \times 4$ blocked (NN) version, the $2 \times 4 \times 2$ blocked (NT) implementation works better with smaller work-groups.

λ_1	1	2	4	8	16	32	64	128
Cache lines (NN)	513	258	132	72	48	48	72	132
L1 fraction (NN)	2.0	1.0	0.52	0.28	0.19	0.19	0.28	0.52
Cache lines (NT)	258	132	72	48	48	72	132	258
L1 fraction (NT)	1.0	0.52	0.28	0.19	0.19	0.28	0.52	1.0

Table 1.2. L1 cache utilisation for the $1 \times 4 \times 4$ blocked NN and the $2 \times 4 \times 2$ blocked NT kernels. We note that if we want to choose $\lambda_0 = 4$, we are restricted to $\lambda_1 \leq 32$.

1.4.8 L1 Cache blocking

We saw above that the L1 cache utilisation determined our robustness against thread divergence, but every program will, if we do not interfere with thread scheduling in any way, experience thread divergence. For large enough matrices, this will always lead to performance degradations in the kernels we have discussed so far. Our strategy to get around this issue is to introduce yet another level of blocking, and to rewrite the algorithm with this additional level of block-matrix multiplication.

As a means of relating this level of blocking to the discussion about register blocking, we now introduce a much larger ΔK, so that we have two: the $\Delta K_{\text{reg.}}$ introduced previously, and the new ΔK_{cache}. After each set of ΔK_{cache} iterations in the loop, we reassemble all work-items in the work-group, to ensure that no thread divergence appears within the work-group.

Relating the change to the actual code, we insert a barrier operation between every ΔK_{cache} iterations in the loop. As this only limits thread divergence within a work-group, we will still have to take into account the divergence between work-groups that will limit L1 cache sharing between different work-groups. It therefore seems as if we should expect that this method will work best when there is only one simultaneously active work-group on each core. The full kernel is shown in Listing 1.13.

The benefit of the barrier is that we can get the same L1 cache sharing for large matrices as we had for small matrices. The cost of executing the barrier is due to the fact that we have to reassemble all work-items of the work-group at regular intervals. If no thread divergence occurs, this means that all work-items need to enter into the barrier, which takes time proportional to the number of work-items, and then all work-items need to exit from the barrier, which again takes a number of cycles proportional to the number of work-items involved. This means that the effective cost of a barrier is the number of cycles it takes, times the number of work-items that are taking part in the barrier, or at least[21] $2\lambda_0^2\,\lambda_1^2$ It is therefore benefi-

[21]Executing the `barrier` instruction takes $2\lambda_0\lambda_1$ cycles, and in this time the $\lambda_0\lambda_1$

```
#define di ((uint)2)
#define dj ((uint)2)
#define dk ((uint)32)

kernel void
sgemm(global float4 const *A, global float4 const *B,
  global float2 *C, float alpha, float beta, uint n)
{
  uint j = get_global_id(0);
  uint i = get_global_id(1);
  uint nv4 = n >> 2;

  float4 ab = (float4) 0.0f;
  for (uint k = 0; k < nv4; k += dk)
  {
    for (uint kk = 0; kk < dk; ++kk)
    {
      float4 a0 = A[ 2*i    *nv4 + kk+k];
      float4 a1 = A[(2*i+1)*nv4 + kk+k];

      float4 b0 = B[ 2*j    *nv4 + kk+k];
      float4 b1 = B[(2*j+1)*nv4 + kk+k];

      ab += (float4)(dot(a0, b0), dot(a0, b1),
                     dot(a1, b0), dot(a1, b1));
    }
    barrier(CLK_GLOBAL_MEM_FENCE);
  }

  uint ix = 2*i*(n>>1) + j;
  C[ix]           = alpha * ab.s01 + beta * C[ix];
  C[ix + (n>>1)] = alpha * ab.s23 + beta * C[ix + (n>>1)];
}
```

Listing 1.13. cache-blocked implementation: `cacheblockedNT`. The constants `di`, `dj` and `dk` correspond to our ΔI, ΔJ and ΔK_{cache}, respectively.

cial for the actual execution time of the barrier to have small work-groups. As the work-group divergence that is a consequence of small work-groups will hurt cache-sharing between work-groups, we have two competing behaviours, and we need to find a good compromise. We may therefore want to select a small work-group size to ensure that barriers are cheap (and an NT algorithm, as we say that it goes better with small work-groups). On the other hand we will want to choose a large work-group size, as we want to keep work-group divergence small, and therefore want to have only a

work-items involved are prevented from performing other work, which means that the work lost due to the `barrier` instruction is $2\lambda_0^2\lambda_1^2$. Again, the description is simplified but sufficient for our needs (*e.g.* while the last work-item exits from the barrier, the first work-item is already performing work again).

small number of active work-groups.

We also have to find a good compromise for the value of the parameter ΔK_{cache}. It must be small enough to keep thread divergence low, but we also need to ensure that we do not execute the barrier too often (to keep the cost of executing the barriers low).

The barrier is executed $N/(\Delta K_{\text{reg.}}\Delta K_{\text{cache}})$ times, with a cost of $2\lambda_0^2\lambda_1^2$ cycles each time, and we want to ensure that this cost is is small compared to the number of memory operations. The number of memory operations performed by the work-group between two consecutive barriers is $\Delta K_{\text{cache}}(\Delta I \Delta K_{\text{reg.}} + \Delta K_{\text{reg.}}\Delta J)\lambda_0\lambda_1$ and our condition for ΔK_{cache} becomes

$$\Delta K_{\text{cache}}(\Delta I \Delta K_{\text{reg.}} + \Delta K_{\text{reg.}}\Delta J)\lambda_0\lambda_1 > 2\lambda_0^2\lambda_1^2$$

which we rewrite as

$$\Delta K_{\text{cache}} > \frac{2\lambda_0\lambda_1}{(\Delta I \Delta K_{\text{reg.}} + \Delta K_{\text{reg.}}\Delta J)}$$

and for the $2 \times 4 \times 2$ blocking with a large work-group size we find that we need

$$\Delta K_{\text{cache}} > \frac{2[\#\text{active threads}]}{8+8} = \frac{256}{16} = 16$$

For our kernel, we have chosen $\Delta K_{\text{cache}} = 32$.

1.4.9 Page table look-ups

One aspect that we have not yet discussed but that is important for the performance of the NN and NT versions, is the way they affect the memory management unit. The physical memory is partitioned into areas called pages, and the abstract pointer space is also partitioned into pages in a similar way. Every page that we use in the pointer space must correspond to a page in physical memory, and information about this mapping is stored in page tables. When we try to access memory using a pointer, we need to find out which page it points to in pointer space, and which page that corresponds to in the physical memory space. The process of looking up a page in the page tables is costly, and we want to perform as few page table look-ups as possible when executing our kernel[22]. Going back to the scalar kernels, to keep the discussion simple, the memory accesses of a work-group of the scalar NN kernel looks like

$$\mathop{,}\limits_{k=0}^{N}\left[\left(\mathop{,}\limits_{i=0}^{\lambda_1-1}\mathop{,}\limits_{j=0}^{\lambda_0-1} A[i,k]\right), \left(\mathop{,}\limits_{i=0}^{\lambda_1-1}\mathop{,}\limits_{j=0}^{\lambda_0-1} B[k,j]\right)\right]$$

[22]This description is simplified, but sufficient for our current purposes.

and we see that it needs to access data from all rows of B, but only from a single row of A. The scalar NT kernel instead reads from

$$\mathop{,}\limits_{k=0}^{N}\left[\left(\mathop{,}\limits_{i=0}^{\lambda_1-1}\mathop{,}\limits_{j=0}^{\lambda_0-1} A[i,k]\right),\left(\mathop{,}\limits_{i=0}^{\lambda_1-1}\mathop{,}\limits_{j=0}^{\lambda_0-1} B[j,k]\right)\right]$$

which accesses only one row each of A and B. This means that, if the matrices are much larger than the size of a page, the work-group in the NN variant needs to access memory from many more pages than the NT variant, although the access the same number of bytes. We must expect that the need to access more pages implies a need to perform a larger number of page table look-ups, which decreases performance.

We should look at the memory accesses for the work-groups that are active at the same time, and not just a single work-group, but the conclusion will be the same, that the NN version needs to perform a larger number of page table look-ups than the NT version. The conclusion also remains the same when we look at blocked versions, as we are still in the situation that all work-items access the entire column from B in the NN versions.

If we turn to a cache-blocked NN version, we have a barrier that we introduced to prevent thread divergence, and although we still need a larger number of look-ups in the NN version than in the NT version, we do not need as many at the same time as in the scalar and blocked NN versions, as the barrier ensures that the work-items that are simultaneously active only want to access a limited number of pages at once.

To summarise the discussion of page tables, cache-blocking should improve the situation slightly over the scalar and blocked NN versions, but for large matrices, we must expect the NT versions to have a performance advantage over the NN versions.

1.4.10 Performance

Performance metrics The primary performance metric is the number of floating-points operations (nflops) per second (FLOPS). For each output element $C[i,j]$, $(2N-1)+1=2N$ flops are required to compute the dot-product of $A[i,*]\cdot B[*,j]$ and scale it by α, 1 multiplication to scale $C[i,j]$ by β and 1 addition to obtain the final result. Thus, nflops $= 2N^2(N+1)$. Dividing nflops by execution time in nanoseconds (10^{-9} seconds) gives GFLOPS (10^9 FLOPS).

Overall trends In Table 1.3, we show performance numbers for different kinds of implementations and different matrix sizes. Variations within one kind is not shown, and different rows in the same column can contain results for different variants of the same kind (*e.g.* different blocking parameters or work-group sizes).

size	sNN	sNT	bNN	bNT	cbNN	cbNT
96	2.0	2.0	8.0	11.2	5.1	7.4
192	2.0	2.1	8.4	13.0	5.1	8.1
384	2.0	2.1	7.6	13.2	6.4	10.1
768	2.0	2.1	7.5	13.1	6.6	10.0
1440	2.0	2.1	7.4	13.1	6.5	9.3
2880	1.6	2.0	6.8	10.8	6.1	9.2

Table 1.3. Performance in GFlops for some variants and matrix sizes. For each matrix size and kind of implementation, we have only selected one number, and it may, *e.g.*, use different blocking parameters at different sizes. The columns show the best performance we see in GFlops for scalar (s), blocked (b) and cache-blocked (cb) variants of non-transposed (NN) and transposed (NT) sgemm implementations.

size	blocked		cache-blocked	
	median	high	median	high
96	9.3	9.4	7.0	7.4
192	10.5	10.6	8.0	8.1
384	9.9	10.0	9.9	10.0
768	9.7	9.9	9.7	10.0
1440	9.6	9.9	9.1	9.3
2880	1.1	9.4	9.2	9.2

Table 1.4. Performance in GFlops for $1 \times 4 \times 4$ blocked and cache-blocked NT implementations. Each implementation was run 3 times for each matrix size and each local work-group size. The median and best results from the 9 runs is listed in the table.

We see that the NT versions perform better than NN versions, and that blocked versions are better than scalar versions, as expected. While already the smallest shown matrix size appears sufficient for the scalar versions, the blocked and (more strongly) the cache-blocked variants need larger amounts of work to reach their best performance.

Our experiments with larger matrices, where performance is heavily influenced by system effects like thread-divergence, are not shown in the table, due to variations in results.

Cache-blocking One surprising result in the table is the large difference between the blocked and cache-blocked variants, where the introduction of cache-blocking seems to come at a large cost. This impression is misleading, and due to our selection of data. We only display the best performance achieved for each kind of implementation, and we saw in the Sobel results that the number of registers play a crucial role in determining performance.

For the best blocked NT implementation we have, the corresponding cache-blocked versions uses more registers which prevents us from keeping 128 simultaneous threads per core. Due to the limited number of threads, this version is not our best cache-blocked implementation, as we have other variants that use fewer registers. The columns bNT and cbNT therefore display results for different implementations. If we instead consider (nearly) identical implementations of $1\times4\times4$ blocked and cache-blocked NT variants (shown in Table 1.4) we see much larger similarities in the top results, and we also see the large difference in the median result for large matrices.

Work-group divergence For large workloads, work-group divergence will start to appear, and this will affect performance. The occasional very good performance results with blocking versions probably appear in runs where we see no or very little work-group divergence, whereas the much lower median result shows that the typical case has lower performance.

Cache-blocking, introduced to stifle thread-divergence, also has an effect in preventing work-group divergence, thereby decreasing the variability.

The cost of a barrier In our previous estimate of the cost of executing a barrier instruction, we assumed that we would have one work-item per cycle entering into the barrier, and then one work-item per cycle exiting. In reality, with thread-divergence, we do not have the ideal case of one work-item entering the barrier every cycle, as the first work-item will be a few instructions ahead of the last one. Instead, all work-items will have to wait for the last one to arrive at the barrier. The actual cost of the barrier can therefore be significantly higher than our estimate, if a few work-items of the work-group are far behind in executing the program.

1.4.11 Summary

We started from scalar versions of a generic matrix multiplication, and then transformed the initial kernels, successively arriving at more elaborate implementations. We discussed the reason behind the transformations, and we discussed how the transformations were took advantage of aspects of the hardware. We started by introducing vector operations, and then focused on the memory system and in particular on the L1 cache. We also discussed execution times, as measured on an Arndale development board, and we found that the qualitative results were as we expected, although a quantitative comparison shows that our simplifying assumptions are not always satisfied.

1.5 Conclusion

Acknowledgments

The authors gratefully acknowledge the work of their colleagues at ARM's Media Processing Division.

Bibliography

[ARM Limited 13] ARM Limited. “Mali OpenCL SDK v1.1.”, 2013. http://malideveloper.arm.com/develop-for-mali/sdks/mali-opencl-sdk.

[Howes et al. 10] Lee Howes, Anton Lokhmotov, Alastair F. Donaldson, and Paul H. J. Kelly. “Towards metaprogramming for parallel systems on a chip.” In *Workshop on Highly Parallel Processing on a Chip (HPPC)*, LNCS, 6043, LNCS, 6043, pp. 36–45. Berlin, Heidelberg: Springer-Verlag, 2010. Available online (http://dl.acm.org/citation.cfm?id=1884795.1884803).

[Ryoo et al. 08] Shane Ryoo, Christopher I. Rodrigues, Sam S. Stone, John A. Stratton, Sain-Zee Ueng, Sara S. Baghsorkhi, and Wen-mei W. Hwu. “Program optimization carving for GPU computing.” *J. Parallel Distrib. Comput.* 68:10 (2008), 1389–1401. Available online (http://dx.doi.org/10.1016/j.jpdc.2008.05.011).