Decoupled Access/Execute Metaprogramming for GPU-Accelerated Systems

Lee Howes, Anton Lokhmotov, Paul H.J. Kelly
Department of Computing, Imperial College London

Alastair F. Donaldson
Computing Laboratory, University of Oxford

I. INTRODUCTION

We describe the evaluation of several implementations of a simple image processing filter on an NVIDIA GTX 280 card. Our experimental results show that performance depends significantly on low-level details such as data layout and iteration space mapping which complicate code development and maintenance.

We propose extending a CUDA\(^1\) or OpenCL\(^2\) like model with decoupled Access/Execute (“Æcute” \[1\]) metadata, describing its iteration space ordering and partitioning (execute metadata) and its memory access patterns (access metadata).

We believe that using Æcute metadata will make software engineering for accelerated systems more disciplined and productive, by separating algorithm representation from low-level mapping and tuning.

II. MOTIVATING EXAMPLE

We consider a vertical mean image filter, for which the output pixel at position \((x, y)\) is given by the formula

\[
O_{x,y} = \frac{1}{D} \sum_{k=0}^{D-1} I_{x,y+k}, \text{ where} \tag{1}
\]

- \(I\) is a \(W \times H\) grey-scale input image;
- \(O\) is a \(W \times (H - D)\) grey-scale output image;
- \(D\) is the diameter of the filter, \textit{i.e.} the number of input pixels over which the mean is computed (typically, \(D \ll H\));
- \(0 \leq x < W, 0 \leq y < H - D\).

Let \(N\) be the number of output pixels: \(N = W \times (H - D)\). A naïve parallel algorithm can run \(N\) threads, each producing a single output pixel, which requires \(\Theta(ND)\) reads and arithmetic operations, significantly reducing memory bandwidth and compute requirements for \(T \gg D\). Since the \(x\) and \(y\) loops carry no dependences, up to \(\lceil N/T \rceil\) threads can run in parallel.

Note that since the order of arithmetic operations is undefined in (1), both the naïve and scalable algorithms are functionally, if not arithmetically, equivalent.

Clearly, the optimal value of \(T\) depends on problem parameters \((W, H\) and \(D)\), and device parameters \((\text{e.g. the number of cores and memory partitions})\). Thus, in §II-C we use the iterative compilation approach to find the optimum.

A. Scalable algorithm

The algorithm in Listing 1 strips the computation in the vertical dimension, where up to \(T\) outputs in the same strip are computed serially in two phases. The first phase in lines 6–10 computes \(O_{x,y0}\) according to (1). The second phase in lines 12–19 computes \(O_{x,y}\) for \(y \geq y0 + 1\) as \(O_{x,y-1} + (I_{x,y+D-1} - I_{x,y-1})/D\).

```
// for each column
for(int x = 0; x < W; ++x)
{
  // for each strip of rows
  for(int y0 = 0; y0 < H-D; y0 += y)
  {
    // first phase: convolution
    float sum = 0.0f;
    for(int k = 0; k < D; ++k)
      sum += I[(y0+k)*W + x];
    O[y0*W + x] = sum / (float)D;
    // second phase: rolling sum
    for(int dy = 1; dy < min(T,H-D-y0); ++dy)
    {
      int y = y0 + dy;
      sum -= I[(y-1)*W + x];
      sum += I[(y-1+D)*W + x];
      O[y*W + x] = sum / (float)D;
    }
  }
}
```

Listing 1: Vertical mean image filter algorithm in C.

This algorithm performs \(\Theta(N + ND/T)\) reads and arithmetic operations, significantly reducing memory bandwidth and compute requirements for \(T \gg D\). Since the \(x\) and \(y\) loops carry no dependences, up to \(\lceil N/T \rceil\) threads can run in parallel.

B. Efficient implementation

Implementing the vertical mean filter efficiently on a GPU requires mapping the iteration space onto threads, which are grouped into blocks located in a grid.

The most natural iteration space mapping is into thread blocks on a 2D grid, with each block producing a rectangular WPBX × WPBY section of the output image. However, if the image width is not a multiple of WPBX, significant portions of thread blocks covering the right edge of the image may be unused, as illustrated by Figure 1a.

This issue can be alleviated by mapping into thread blocks on a 1D grid that covers the image by wrapping around the right edge, as illustrated by Figure 1b. As we show in §II-C, a mapping that maximises thread utilisation suffers from misalignment, if the image width is not a multiple of the size

\(^{1}\)http://www.nvidia.com/cuda
\(^{2}\)http://www.khronos.org/opencl
Fig. 1: Different mapping strategies result in different utilisation of threads. Light and dark regions of blocks denote used and unused threads, respectively.

(a) A 2D grid mapping loses efficiency from unused threads off the right image edge.

(b) A 1D grid mapping uses its threads more efficiently by wrapping around the right image edge. For efficiency, it must take into account alignment, which complicates both memory access and iteration.

As the results of padding to a multiple of 64 and 128 are barely distinguishable, we fix the image padding at a multiple of 64 (5184 pixels) for all subsequent experiments.

Figure 2c shows that the 1D grid mapping that maximises thread utilisation by wrapping on 5121 pixels only achieves 5998 Mpixel/s, whilst wrapping on the image padding of 5184 pixels performs worse than wrapping on the warp size multiple of 5152 pixels.

Figure 2d summarises the throughput for the misaligned image padded to 5184 pixels: the 1D grid version wrapped on 5152 pixels achieves 9575 Mpixel/s at $T = 396$, whilst the 2D grid version achieves only 9056 Mpixel/s at $T = 409$; thus, the 1D grid version performs 6% better than the 2D grid one.

III. Towards Metaprogramming

To ease the programmer’s burden of mapping and tuning computation kernels to GPU architectures, we propose extending a kernel’s description with decoupled Access/Execute metadata. Execute metadata for a kernel describes its iteration space ordering and partitioning. Access metadata for a kernel describes memory locations the kernel may access on each iteration.

Listing 2: Æcute metadata for the vertical mean image filter.

```c
// Array descriptors (C array wrappers)
Array2D<float> arrayI(I[0][0], W, H);
Array2D<float> arrayO(O[0][0], W, H-D);

// Execute metadata: parallel iteration space
IterationSpace1D x(0,W);
IterationSpace1D y(0,H-D);
IterationSpace2D iterXY(x,y);

// Access metadata: iteration space -> memory
VerticalStrip2D_R accessI(iterXY, arrayI, D);
Point2D_W accessO(iterXY, arrayO);
```

As the results of padding to a multiple of 64 and 128 are barely distinguishable, we fix the image padding at a multiple of 64 (5184 pixels) for all subsequent experiments.

Figure 2c shows that the 1D grid mapping that maximises thread utilisation by wrapping on 5121 pixels only achieves 5998 Mpixel/s, whilst wrapping on the image padding of 5184 pixels performs worse than wrapping on the warp size multiple of 5152 pixels.

Figure 2d summarises the throughput for the misaligned image padded to 5184 pixels: the 1D grid version wrapped on 5152 pixels achieves 9575 Mpixel/s at $T = 396$, whilst the 2D grid version achieves only 9056 Mpixel/s at $T = 409$; thus, the 1D grid version performs 6% better than the 2D grid one.

III. Towards Metaprogramming

To ease the programmer’s burden of mapping and tuning computation kernels to GPU architectures, we propose extending a kernel’s description with decoupled Access/Execute metadata. Execute metadata for a kernel describes its iteration space ordering and partitioning. Access metadata for a kernel describes memory locations the kernel may access on each iteration.

Listing 2: Æcute metadata for the vertical mean image filter.

```c
// Array descriptors (C array wrappers)
Array2D<float> arrayI(I[0][0], W, H);
Array2D<float> arrayO(O[0][0], W, H-D);

// Execute metadata: parallel iteration space
IterationSpace1D x(0,W);
IterationSpace1D y(0,H-D);
IterationSpace2D iterXY(x,y);

// Access metadata: iteration space -> memory
VerticalStrip2D_R accessI(iterXY, arrayI, D);
Point2D_W accessO(iterXY, arrayO);
```

We give an example of Æcute metadata for the vertical mean image kernel in Listing 2. In lines 1–3 we wrap accesses to plain C arrays $I[W][H]$ and $O[W][H-D]$ into Æcute array descriptors $arrayI$ and $arrayO$ to cleanse the kernel of uncontrolled side-effects. In lines 5–8 we construct a 2D iteration space descriptor $iterXY$ from 1D descriptors $x$ and $y$, having the same bounds as the output image dimensions. By default, an iteration space is parallel in every dimension. Finally, in lines 10–12 we specify that on each iteration of the 2D iteration space the kernel reads a vertical strip of $D$ pixels from $arrayI$ and writes a single pixel to $arrayO$.

Similar to Stanford’s Sequoia language [3], we target systems with software-managed memory hierarchies and seek to separate a high-level algorithm representation from a system-specific mapping. Unlike Sequoia, we base our mapping on partitioning (manually or automatically) an iteration space into disjoint subspaces and infer memory access of subspaces from Æcute metadata.

For example, for GPU-accelerated systems, a hierarchy of iteration space partitions can specify subspaces to be executed:

- at the lowest level, by individual threads:
  ```c
  // ixT outputs per thread
  iterXY.partitionThreads(1,T);
  ```
Fig. 2: Comparison of different mappings with various image sizes, data padding and thread wrapping alignment.

- at the middle level, by blocks of possibly cooperating threads:
  
  ```c
  // 128xT outputs per block
  iterXY.partitionBlocks(128,T);
  ```

- at the highest level, by possibly cooperating compute devices:
  
  ```c
  // (W/2)x(H-D) outputs per device
  iterXY.partitionDevices(W/2,H-D)
  ```

IV. WORK IN PROGRESS

We are working on a tool that will take a high-level algorithm representation and generate efficient device-specific OpenCL code. The representation will be kept similar to C++, e.g. as in Listing 1 with accesses to C arrays replaced with accesses to `execute` array descriptors as in Listing 2. Code generation will be particularly oriented towards effectively orchestrating data movement in software-managed memory hierarchies, including automatically handling such low-level details as data alignment and padding.

REFERENCES

