Towards Metaprogramming for Parallel Systems on a Chip

Lee Howes[1], Anton Lokhmotov[1], Alastair F. Donaldson[2], and Paul H.J. Kelly[1⋆]

[1] Department of Computing, Imperial College London,
180 Queen's Gate, London, SW7 2AZ, UK
[2] Computing Laboratory, University of Oxford,
Parks Road, Oxford, OX1 3QD, UK

Abstract. We demonstrate that the performance of commodity parallel systems significantly depends on low-level details, such as storage layout and iteration space mapping, which motivates the need for tools and techniques that separate a high-level algorithm description from low-level mapping and tuning. We propose to build a tool based on the concept of decoupled Access/Execute metadata which allow the programmer to specify both execution constraints and memory access pattern of a computation kernel.

1 Introduction

We evaluate several implementations of simple image filters on x86 multicore systems and a GPU-accelerated system. Our experimental results demonstrate that efficiently implementing an algorithm to execute on commodity parallel hardware requires careful tuning to match the hardware characteristics, as the performance depends significantly on low-level details such as iteration space mapping and storage layout. While such manual tuning is possible, it is not practical: the number of versions to write and maintain grows with the number of target architectures. For applications consisting of multiple kernels such development and maintenance becomes infeasible.

We believe that innovative tools and techniques that separate a high-level algorithm description from low-level mapping and tuning will make software engineering for parallel systems more productive and disciplined. We propose to build such a tool based on the concept of Access/Execute (Æcute) metadata which capture both execution constraints and memory access patterns [1].

2 Mean filter

Consider a one-dimensional mean filter, for which the output at t is given by the formula

$$\mathbf{O}_t = \frac{1}{D} \sum_{k=0}^{D-1} \mathbf{I}_{t+k}, \text{ where} \tag{1}$$

⋆ We acknowledge financial support by the EPSRC grants EP/E002412 and EP/G051100.

– $\mathbf{I}$ is an input array of $N + D$ real elements;
– $\mathbf{O}$ is an output array of N real elements;
– D is the *diameter* of the filter, *i.e.* the number of input elements over which the mean is computed (typically, $D \ll N$).

A naïve parallel algorithm can run N threads, each producing a single output element, which requires $\Theta(ND)$ reads and arithmetic operations. A good parallel algorithm, however, must be efficient and scalable [2].

2.1 Scalable algorithm

The algorithm in Listing 1 *strips* the computation, where up to T outputs in the same strip are computed serially in two *phases*. The first phase in lines 2–6 computes $\mathbf{O}_{t0}$ according to (1). The second phase in lines 8–14 computes $\mathbf{O}_t$ for $t \geq t0 + 1$ as $\mathbf{O}_{t-1} + \big(\mathbf{I}_{t+D-1} - \mathbf{I}_{t-1}\big)/D$.

```
for(int t0 = 0; t0 < N; t0 += T) {                          1
  // first phase: convolution                               2
  float sum = 0.0f;                                         3
  for(int k = 0; k < D; ++k)                                4
    sum += I[t0+k];                                         5
  O[t0] = sum / (float)D;                                   6
                                                            7
  // second phase: rolling sum                              8
  for(int dt = 1; dt < min(T,N-t0); ++dt) {                 9
    int t = t0 + dt;                                        10
    sum -= I[t-1];                                          11
    sum += I[t-1+D];                                        12
    O[t] = sum / (float)D;                                  13
  }                                                         14
}                                                           15
```

Listing 1: Scalable mean filter algorithm in C.

This algorithm performs $\Theta(N + ND/T)$ reads and arithmetic operations, considerably reducing memory bandwidth and compute requirements for $T \gg D$. Since the $t0$ loop carries no dependences, up to $\lceil N/T \rceil$ threads can run in parallel. Thus, this algorithm trades off parallelism against work efficienty.

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

2.2 Vertical and horizontal mean image filters

Mean filtering is a simple technique for reducing noise in digital images.

The vertical mean image filter is the one-dimensional mean filter applied to columns of a two-dimensional image of $W \times H$ pixels:

$$\mathbf{O}_{x,y} = \frac{1}{D} \sum_{k=0}^{D-1} \mathbf{I}_{x,y+k}, \text{ where } 0 \leq x < W, 0 \leq y < H - D. \tag{2}$$

Similarly, the horizontal mean image filter is the mean filter applied to rows of an image:

$$\mathbf{O}_{x,y} = \frac{1}{D} \sum_{k=0}^{D-1} \mathbf{I}_{x+k,y}, \text{ where } 0 \leq x < W - D, 0 \leq y < H. \tag{3}$$

Using the algorithm of §2.1, the mean image filters perform $\Theta(N + ND/T)$ reads and arithmetic operations, and can run up to $\lceil N/T \rceil$ parallel threads, where N is the number of output pixels and T is the number of output pixels per strip.

Clearly, the optimal value of T depends on problem parameters W, H and D, and on hardware parameters such as the number of supported threads and memory bandwidth. In our evaluation (§4), we find the optimum by iteratively compiling the kernels for a range of values of T and evaluating the performance.

3 Implementation

We describe efficient implementations of the mean image filter kernels for a GPU, with the NVIDIA Compute Unified Device Architecture (CUDA) [3], and for a multicore CPU, with Intel Streaming SIMD Extensions (SSE) [4]. We assume that the image pixels are represented as single-precision floating-point numbers, and that the images are stored in row-major order.

3.1 Architecture overview

Roughly, a CUDA thread corresponds to an individual SSE vector lane, whilst a CUDA thread block corresponds to a full SSE vector. A GPU core (streaming multiprocessor) executes blocks of multiple threads in SIMD groups of 32 threads (warps) using the 8-lane SIMD unit; a CPU core operates on 4-element vectors using the 4-lane SIMD unit. As a rule of thumb, a GPU runs thousands of threads, whilst a CPU only tens of threads (counting vector lanes).

SSE only supports *coalesced* access to off-chip memory, *e.g.* storing a vector register into a contiguous (and preferably aligned) 128-bit memory region; CUDA supports uncoalesced access to off-chip memory, albeit at a lower memory bandwidth.[3] To reduce off-chip memory access, SSE provides a family of instructions for shuffling data in vector registers, whilst CUDA provides on-chip memory shared between threads in a block. Effectively, these mechanisms enable fast inter-thread cooperation.

3.2 Vectorisation

Vertical mean filter Conceptually, the vertical mean filter has a parallel outer loop iterating over each column and a parallel inner loop iterating over strips of rows:

```
parfor(x = 0; x < W; ++x) // for each column
  parfor(y0 = 0; y0 < H-D; y0 += T) // for each strip of rows
    // two-phase computation here
```

[3] Rules for coalescing vary between different architecture generations.

To enable memory coalescing, threads in a thread block (lanes in a vector) must access a contiguous (and preferably aligned) memory region, which is achieved by assigning adjacent columns to adjacent threads: technically, the loop x is interchanged with the loop $y0$, and then stripmined into vectors.

Horizontal mean filter The horizontal mean filter has a parallel outer loop iterating over each row and a parallel inner loop iterating over strips of columns:

```
parfor(y = 0; y < H; ++y) // for each row
  parfor(x0 = 0; x0 < W-D; x0 += T) // for each strip of columns
    // two-phase computation here
```

Serialisation within strips of columns, however, results in *no* memory coalescing: adjacent threads access adjacent rows with a stride of W. One option, illustrated in Fig. 1, is to transpose the $W \times H$ input image $\mathbf{I}$ to an $H \times W$ intermediate image $\mathbf{T}'$, run the vertical mean filter on $\mathbf{T}'$ to produce an $H \times (W - D)$ intermediate image $\mathbf{T}''$, and then transpose $\mathbf{T}''$ to produce the $(W - D) \times H$ output image $\mathbf{O}$. Another option is to effectively *fuse* transposition and computation into one optimised kernel, by using on-chip memory on a GPU or shuffle instructions on a CPU.

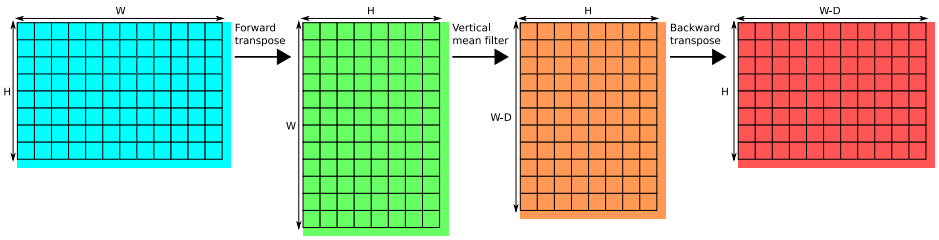
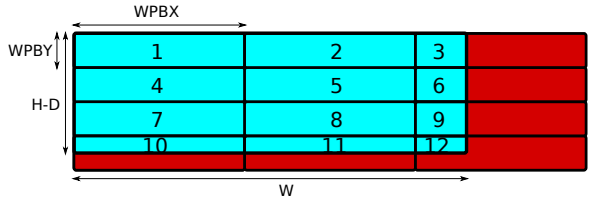


Fig. 1: The horizontal mean filter kernel implemented as a pipeline of the forward transpose, vertical mean filter and the backward transpose kernels. Shadows represent the padding that may be necessary to improve the bandwidth of the transpose kernels (§4.1).

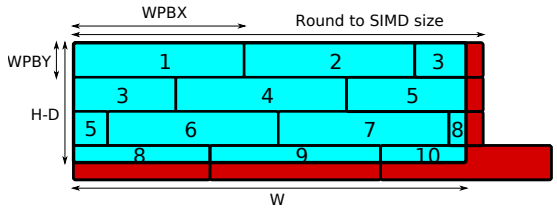
3.3 Parallelisation

Given a low thread count on a CPU, the inner loop computation can be completely serialised, resulting in maximum work efficiency: the CPU out-of-order issue logic can extract adequate instruction level parallelism from the serial instruction stream. In contrast, a GPU exploits limited instruction level parallelism and relies on thread level parallelism to cover memory latency.

On a GPU, thread blocks are located in a grid. For two-dimensional iteration spaces over images, a two-dimensional grid is most natural, with each block producing a rectangular section of the output image. As Fig. 2a illustrates for the vertical mean filter, significant portions of thread blocks covering the right edge of the image may be idle if the image width is not a multiple of the number of columns per thread block.



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



(b) A 1D grid mapping has fewer idle threads by wrapping around the right image edge. Taking into account alignment for efficiency reasons complicates addressing and iteration.

Fig. 2: Different mapping strategies result in different utilisation of threads. Light and dark regions of blocks denote busy and idle threads, respectively. WPBX and WPBY stand for *work per block* in the x and y dimensions, respectively. For 128×1 thread blocks used in our evaluation (§4), WPBX = 128 and WPBY = T.

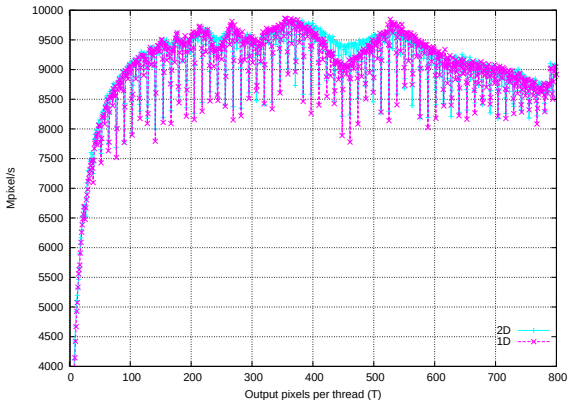
This issue can be alleviated by mapping the iteration space onto a one-dimensional grid that covers the image by wrapping around its right edge, as illustrated by Fig. 2b. As we show in §4.1, a mapping that maximises thread utilisation suffers from misalignment, if the image width is not a multiple of the warp size; a better mapping takes alignment into account by wasting a small number of threads on the right of the image, thus ensuring that the first pixel of each row is handled by the first thread in a warp.

4 Experimental results

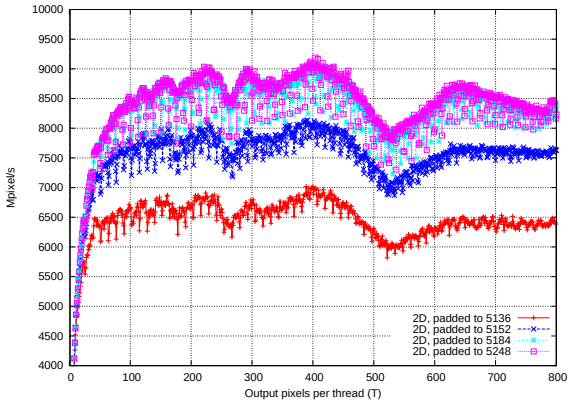
4.1 GPU

We present results obtained on a dual-core 3GHz Intel Core 2 Duo E8400 system with 2 GiB RAM, equipped with an NVIDIA GTX 280 card, running 64-bit Linux Ubuntu 8.04. Code is compiled using CUDA SDK 2.2 and GCC 4.2.4 with the "-O3" optimisation setting. We measure the kernel execution time only and report the best throughput out of 50 runs. In all the experiments, we fix the number of threads per block at 128 (128×1), as we nearly achieve the peak memory efficiency with this setting: $\approx$ 10 Gpixel/s $\times$ 4 bytes/pixel $\times$ (2 reads + 1 write) = 120 GB/s (close to the bandwidth of aligned copy on this card).

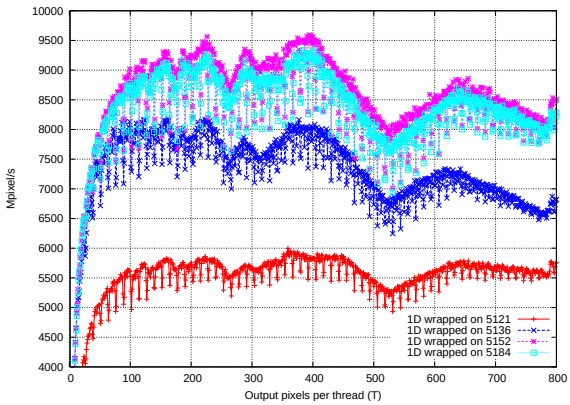
Vertical mean filter Fig. 3a shows that the 1D and 2D grid versions of the vertical mean filter (Fig. 2) are similar in throughput when applied to a 5120×3200 image,



(a) 5120×3200 image. 2D grid; 1D grid.



(b) 5121×3200 image. 2D grid. Data padded to multiples of 16, 32, 64, and 128 pixels.



(c) 5121×3200 image. Data padded to 5184 (a multiple of 64) pixels. 1D grid wrapped on the image width and multiples of 16, 32 and 64 pixels.

Fig. 3: CUDA implementations for the vertical mean filter on a 5120×3200 image (a) and on a 5121×3200 image (b & c), with different iteration space mapping and storage layout.

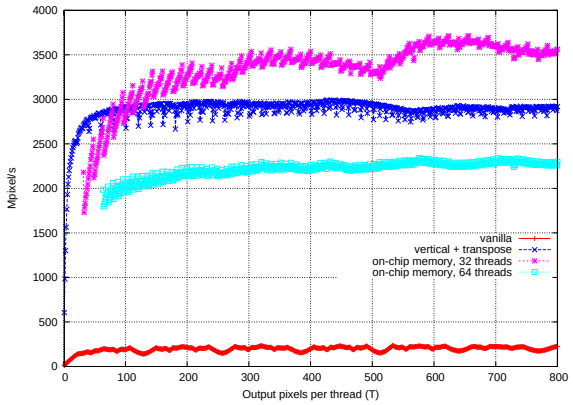


Fig. 4: CUDA implementations for the horizontal mean filter on a 5120×3200 image.

where 5120 is a multiple of 128 pixels. The throughput is below 0.8 Gpixel/s (not shown) when each thread produces a single pixel ($T = 1$), climbs fast with increasing serial efficiency, achieving the peak throughput of 9.9 Gpixel/s at several points, and then declines with decreasing parallelism.

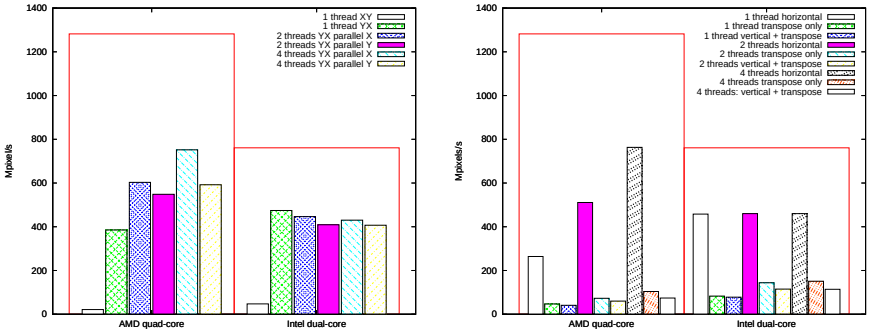
When applied to a 5121×3200 image, however, the 2D grid version only achieves 7.0 Gpixel/s, as shown by the bottom line in Fig. 3b. Whilst we allocate memory using the `cudaMallocPitch` function, which pads the image to a multiple of 16 pixels to enable memory coalescing (5136 pixels in this case), such allocation leads to DRAM partition conflicts. We remedy the conflicts by manually padding the image to a multiple of 32, 64 and 128. Since the results of padding to a multiple of 64 and 128 are very close, we fix the image padding at a multiple of 64 (5184 pixels) for all subsequent experiments.

Fig. 3c shows that the 1D grid mapping that maximises thread utilisation by wrapping on 5121 pixels hardly achieves 6.0 Gpixel/s, whilst wrapping on the image padding of 5184 pixels performs worse than wrapping on the warp size multiple of 5152 pixels.

To summarise, for the misaligned image padded to 5184 pixels, the 1D grid version wrapped on 5152 pixels achieves 9.6 Gpixel/s, whilst the 2D grid version achieves only 9.1 Gpixel/s; thus, the 1D grid version is 6% faster than the 2D grid version.

Horizontal mean filter Fig. 4 shows that the vanilla horizontal mean filter version on a 5120×3200 image achieves only 230 Mpixel/s, for most values of T. The version that uses on-chip memory to effectively fuse transposition and computation into one optimised kernel, achieves 2.4 Gpixel/s and 3.7 Gpixel/s when using 64 and 32 threads.

The composite version that uses separate transpose and vertical mean kernels (Fig. 1) achieves 3.0 Gpixel/s. Note, however, that this performance is only achieved when the intermediate images $\mathbf{T}'$ and $\mathbf{T}''$ are both padded by 64 pixels, resulting in the bandwidth of 80.1 GiB/s and 63.3 GiB/s for the forward and backward transposes, respectively. Without the padding, the bandwidth is only 60.2 GiB/s and 19.1 GiB/s.



(a) The vertical mean filter.

(b) The horizontal mean filter.

Fig. 5: Comparison of different CPU implementations on a 5120×3200 image. The large surrounding boxes represent the peak memory copy throughput for each of the systems, as obtained by running the STREAM benchmark [5].

We estimate that assembling the composite version from the already available components took half a day versus five days for the on-chip memory version. Tuning the kernels and finding the optimal padding parameters to improve the bandwidth, however, took another half a day.

4.2 CPU

We present results on a 2.3 GHz quad-core AMD Phenom 9650 system with 8 GiB RAM (AMD) and on a 3 GHz dual-core Intel Core 2 Duo E8400 system with 2 GiB RAM (Intel), both running 64-bit Linux Ubuntu 8.04. Code is parallelised using OpenMP pragmas and compiled using Intel C Compiler 11.0 with the "-xHost -fast" setting.

Vertical mean filter In the worst performing version in Fig. 5a ("XY"), the loop over columns x and the loop over strips of rows $y0$ have not been interchanged, which results in strided memory accesses. Applying loop interchange ("YX") results in a vast performance increase. Performance on the AMD system increases with enabling more cores, whilst the Intel system achieves the peak performance with a single core, which can be attributed either to the compiler better optimising for Intel's own architecture or to lower performance of a single core on the AMD system. On both systems, it is always more beneficial to parallelise across multiple cores the x loop ("parallel X") than the $y0$ loop ("parallel Y").

Horizontal mean filter The best performing version in Fig. 5b is obtained by the Intel compiler optimising a naïve C implementation that runs through memory sequentially in the horizontal dimension ("horizontal"), thus triggering the CPU cache prefetching mechanism. On the CPUs, the forward and backward transposes are too costly even when using the highly optimised Intel MKL library ("transpose only"), and adding a

best performing version of the vertical mean filter ("vertical + transpose") makes little difference. Indeed, the naïve implementation is so fast that there is nothing to gain from vectorisation but a lot to lose from transposition.

5 Towards Metaprogramming

5.1 Decoupled Access/Execute Metadata

To ease the programmer's burden of mapping and tuning computation kernels to parallel systems on a chip, we propose extending a kernel's description with decoupled Access/Execute (Æcute) metadata [1]. *Execute* metadata for a kernel describe its iteration space ordering and partitioning; *access* metadata describe locations in uniform memory that the kernel may access on each iteration.

```
// Array descriptors (C array wrappers)                          1
Array2D<float> arrayI(&I[0][0], W, H);                           2
Array2D<float> arrayO(&O[0][0], W, H-D);                         3
                                                                 4
// Execute metadata: parallel iteration space                    5
IterationSpace1D x(0,W);                                         6
IterationSpace1D y(0,H-D);                                       7
IterationSpace2D iterXY(x,y);                                    8
                                                                 9
// Access metadata: iteration space -> memory                    10
VerticalStrip2D_R accessI(iterXY, arrayI, D);                    11
Point2D_W accessO(iterXY, arrayO);                               12
```

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

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

5.2 Related work and discussion

Effectively orchestrating data movement in software-managed memory hierarchies is paramount to achieving high performance but is tedious and error-prone. The CUDA-lite [6] tool seeks to automate data movement between on-off chip and on-chip GPU memories by generating appropriate code from ad-hoc source code annotations. We have addressed the problem of generating data movement code between two levels of memory hierarchy on the Cell BE architecture in our previous work [1]. We now aim to address a broader problem of generating code for both data movement across the full memory hierarchy (including the host memory) *and* the iteration space traversal.

Similar to the Sequoia language [7], we 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 a GPU-accelerated system, a hierarchy of iteration space partitions can specify subspaces to be executed: at the lowest level, by individual threads; at the middle level, by blocks of possibly cooperating threads; at the highest level, by possibly cooperating compute devices:

```
iterXY.partitionThreads(1,T);      // 1xT outputs/thread
iterXY.partitionBlocks(128,T);     // 128xT outputs/block
iterXY.partitionDevices(W/2,H-D); // (W/2)x(H-D) outputs/device
```

Ryoo *et al.* [8] also highlight the need for design space exploration, which they call *optimization carving*, but leave out the question of automatically generating different code versions from a high-level representation.

6 Future work

OpenCL [9], a new low-level standard API, aims to provide software portability across heterogeneous systems. Thus, instead of writing, for example, separate CUDA and SSE kernels, the programmer will be able to write an OpenCL kernel and run it on any standard-compliant implementation. However, OpenCL per se does not address the problem of *performance portability*, since low-level code optimised for one device may perform dismally on another, as we have demonstrated in this paper.

We aim to tackle this problem by designing and implementing a framework that will take a device-independent algorithm representation with Æcute metadata and generate efficient device-specific OpenCL code to be processed by vendor compilers.

References

1. Howes, L.W., Lokhmotov, A., Donaldson, A.F., Kelly, P.H.: Deriving efficient data movement from decoupled Access/Execute specifications. In: HiPEAC'09. Volume 5409 of LNCS., Springer (2009) 168–182
2. Lin, C., Snyder, L.: Principles of Parallel Programming. Addison-Wesley, MA, USA (2008)
3. NVIDIA: CUDA. `http://www.nvidia.com/cuda` (2006–2009)
4. Bik, A.J.: The Software Vectorization Handbook. Applying Multimedia Extensions for Maximum Performance. Intel Press (2004)
5. McCalpin, J.D.: STREAM: Sustainable memory bandwidth in high performance computers. `http://www.cs.virginia.edu/stream/` (1990–2009)
6. Ueng, S.Z., Lathara, M., Baghsorkhi, S.S., Hwu, W.m.W.: CUDA-Lite: Reducing GPU programming complexity. In: LCPC'08. Volume 5335 of LNCS., Springer (2008) 1–15
7. Fatahalian, K., Horn, D.R., Knight, T.J., Leem, L., Houston, M., Park, J.Y., Erez, M., Ren, M., Aiken, A., Dally, W.J., Hanrahan, P.: Sequoia: programming the memory hierarchy. In: Supercomputing'06. (2006) 83
8. Ryoo, S., Rodrigues, C.I., Stone, S.S., Stratton, J.A., Ueng, S.Z., Baghsorkhi, S.S., Hwu, W.m.W.: Program optimization carving for GPU computing. J. Parallel Distrib. Comput. **68**(10) (2008) 1389–1401
9. The Khronos Group: OpenCL. `http://www.khronos.org/opencl` (2008–2009)