Neon: A Domain-Specific Programming Language for Image Processing

Brian Guenter\(^1\)  
\(^1\)Microsoft Research

Diego Nehab\(^1,\ 2\)  
\(^2\)IMPA

Abstract

Neon is a high-level domain-specific programming language for writing efficient image processing programs which can run on either the CPU or the GPU. End users write Neon programs in a C# programming environment. When the Neon program is executed, our optimizing code generator outputs human-readable source files for either the CPU or GPU. These source files are then added to the user source tree, compiled and executed. When targeting CPUs, the Neon compiler automatically generates highly optimized, multithreaded, vectorized code for processors supporting the SSE instruction set. For GPU applications, the compiler generates HLSL pixel shaders and the required Direct3D 10 setup code to sequence shader passes. In both cases, the performance of Neon programs rivals those of other widely used general purpose image processing systems such as Apple Core Image and Adobe Pixel Bender. In particular, the performance of Neon programs running on the CPU is closer to GPU performance than might be expected from the peak floating-point performance of each architecture.

Keywords: image processing, image processing languages, domain specific languages

1 Introduction

Image processing is becoming increasingly important as digital still and video cameras decline in price and increase in capability. This declining cost has led to an enormous increase in the number of cameras worldwide and to an explosion of video and still content. The value of captured images is greatly enhanced by image processing: computer vision algorithms can identify friends and relatives to simplify video and image indexing, multiple still images can be stitched together into high resolution panoramas, photo manipulation can enhance the color saturation and sharpness of images etc. Often, the processing must be done in real time: a task which may exceed the capabilities of today’s fastest computers. Even for offline tasks, increasing the speed of image processing operations is essential to keep up with the rapidly expanding amount of content.

Modern computers have architectural features which can process images rapidly: vector instructions, multicore CPUs, and dedicated GPUs. On Intel Core i7 processors, for example, multithreading and vectorization can often increase speed by close to 16 times compared to scalar, single-threaded code. Next generation CPUs will double the width of vector operations for an additional \(2 \times\) performance boost [Larsson and Palmer 2009]. Even greater speed improvements are possible by moving image processing tasks to the GPU.

Unfortunately, properly using these features is not easy. For example, the SSE instruction set is available to high-level programming languages in the form of low-level intrinsics that are similar to assembly language. Furthermore, achieving peak performance on CPUs requires careful considerations of data layout and access patterns [Int 2008]. Similarly, programming for GPUs generally requires knowledge of shading languages (such as HLSL or GLSL) as well as complex graphics APIs (such as Direct3D and OpenGL). Programmers must write a substantial amount of setup code to put the GPU in the correct state and to sequence multiple rendering passes properly. In summary, in either type of architecture, writing programs that achieve the highest performance is tedious, error prone, and requires expert knowledge of low-level architectural details. Maintaining a code-base of highly optimized routines can quickly become an intractable problem, especially when many different architectures are targeted.

Neon is a new domain-specific image processing language that allows any programmer to produce image processing routines that achieve the highest levels of image processing performance, without ever worrying about the details of vectorization or multithreading. Neon programs compile natively to the CPU and the GPU, can be embedded in any application, and often outperform hand-optimized code.

2 Related Work

In contrast to hand-coded image processing libraries (such as [Int 2009]), which operate through a series of independent subroutine calls, Neon performs global analysis of a complete image-processing program to generate a single highly optimized procedure. This has several advantages: first, the programmer is not constrained by the coarse granularity of available subroutines. For example, consider a routine that converts from the RGB color space to CIE YIQ. Such a routine may take R, G, and B channels as input and return the resulting Y, I, and Q channels. If the client algorithm only needs the Y channel, then the work performed computing the I and Q channels is wasted. Second, image processing libraries process data in a sequence of independent function calls, each of which reads and writes an intermediate image, or a small block of an image. If many operations are cascaded, this leads to many unnecessary reads and writes, which can be costly even when blocking is used to keep these intermediate values in first or second-level cache. By doing a global analysis over an entire sequence of operations, the Neon compiler can frequently generate code which keeps intermediate results in CPU or GPU registers. This dramatically reduces memory traffic and therefore improves performance.

Another approach to developing image processing applications is to use one of many array programming languages. These include APL as well as the more popular and modern Matlab and IDL. The generality of these programming languages make it difficult to generate efficient multithreaded vectorized code on the CPU, and even harder to target GPUs. Programmers soon realize that high-performance is only obtainable from predefined library functions, which leads problems akin to the use of image processing libraries.

Programming environments that are specific to the domain of image processing have been commercially available for some time. Two of the most widely used are Apple’s Core Image [App 2008] and Adobe’s Pixel Bender [Ado 2009]. Whereas Core Image is tied to the Mac OS X platform, Pixel Bender is embedded in Adobe applications, such as Photoshop, After Effects, or Flash Player. In contrast, the Neon compiler outputs human-readable C++ and HLSL programs that can be embedded in any application and run in any platform\(^1\).

Core Image and Pixel Bender follow the strategy of [Shantzis 1994]: the output image is defined by a graph that specifies sequences of operations (also called filters, kernels, or effects) to be

\(^1\)Porting our Direct3D 10 back end to OpenGL would take a few days.
Determining whether to perform imaging operations on the CPU or the GPU, and how to optimize code for those operations, requires an understanding of the very different architectural characteristics of these two devices. One must also consider the nature of typical image processing operations, which tend to have regular memory access patterns but a low arithmetic to memory-access ratio (we will refer to this as FLOPS/IO).

Applied to a set of inputs. At run-time, rendering is performed on demand. Conceptually, the system traverses the graph backwards and only performs the operations that are necessary for the computation of each output pixel. Both Core Image’s and Pixel Bender’s run-times can analyze the structure of effect graphs and eliminate unnecessary intermediate results, although the extent to which this analysis is performed in practice is unclear from publicly available documents.

The Core Image and Pixel Bender programming languages are ostensibly based on the OpenGL Shading Language, but may fallback to execution on the CPU when needed. In this case, both systems resort to emulation (which may or may not be jitted). This bias towards GPUs leads to significant performance losses when code runs on a CPU. Even the fastest CPU implementations of pixel shading (see WARP, for example [Glaister 2008]) carry too much overhead simulating GPU behavior (rasterization, pixel formats, etc) to be competitive in general image processing tasks.

Neon was designed to satisfy three goals: make the language as general purpose as practical, hide all low-level architectural details from the user, and generate the most efficient code possible. The decision to emphasize efficient code generation, in particular, places strong constraints on generality. The current implementation of Neon requires all images to be of the same size, in the same coordinate frame, with all pixels at integer locations. This simplification has allowed us to generate exceptionally efficient code for both the CPU and GPU but still gives the language enough generality so that it can perform the most common, and important, image processing functions used for picture and video editing.

Neon programs are written in a higher-level language that was not designed for GPUs exclusively. Our CPU and GPU back ends generate native code for each architecture, using the best data layouts in each case. The result is much better performance when running on CPUs, and no compromise when running on GPUs (see tables 5, 6, and 7).

### Table 1: Comparison of memory bandwidth of a Core i7 CPU to low and high end GPUs

The next two columns show the speed of transferring data from the CPU to the GPU and back. Measured with SiSoft Sandra 2010.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Main memory</th>
<th>CPU to GPU</th>
<th>GPU to CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>HD 4850</td>
<td>29.2 GB/s</td>
<td>4.24 GB/s</td>
<td>351 MB/s</td>
</tr>
<tr>
<td>HD 5870</td>
<td>114 GB/s</td>
<td>4.26 GB/s</td>
<td>870 MB/s</td>
</tr>
<tr>
<td>470 GTX</td>
<td>103 GB/s</td>
<td>2.42 GB/s</td>
<td>5.28 GB/s</td>
</tr>
<tr>
<td>Core i7 940</td>
<td>18 GB/s</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

In figure 1 and table 1, we can see how different these two architectures are. For an application with a working set size of 128KB or less, the cache bandwidth of a 2.93GHz Core i7 940 CPU is over 250GB/s. Even for a working set size as large as 1MB, the cache bandwidth is still almost 150GB/s. Memory latency, shown in figure 2, is also very low: 1.4ns (4 clock cycles) for accesses within a 16KB range, and less than 5ns for accesses within a 256KB range.

The Sandra benchmarks only measure main memory bandwidth for graphics cards so we can’t directly compare cache performance between CPUs and GPUs. Referring to table 1 we can see the bandwidth from main memory to the CPU is 18GB/s for the Core i7. The ATI Radeon HD 4850, a low-cost graphics card, has slightly better performance, achieving 29GB/s bandwidth to internal GPU memory. GPU to CPU transfers are especially slow, at about 350MB/s. The far more powerful ATI Radeon HD 5870 has much higher internal memory bandwidth (114GB/s) and faster, but still slow, GPU to CPU bandwidth. The NVIDIA 470 GTX GPU has high internal bandwidth, 103GB/s, and, surprisingly, higher GPU to CPU than CPU to GPU bandwidth. But even with this higher bandwidth, the round-trip overhead makes the GPU less attractive for processing tasks that require image data to be sent to the GPU and returned to the CPU.

From this, we can expect GPUs to perform well on workloads that:

- do not require round-trip data transfers to CPU main memory;
- have a high arithmetic to memory-access ratio;
- have regular memory access patterns, preferably streaming the data through the GPU once or at most a few times.

CPUs have much lower raw floating-point performance compared to GPUs, but have much larger caches, with very high bandwidth; keeping computation in cache as much as possible can yield large performance gains, especially for workloads that have low computation to memory-access ratio or irregular memory access patterns.

### 4 Architecture of the Neon System

Neon is a domain specific programming language embedded in the C# programming language. Neon programs are written in C#: operator overloading and custom class types and constructors are used to define the semantics of the Neon language. We chose C# as the host language for Neon because it is a modern language, with labor...

Figure 1: Cache memory bandwidth of a 2.93 GHz Core i7 940, measured with SiSoft Sandra 2010. For small working sets the bandwidth is significantly higher than the main memory bandwidth on high-end graphics cards.
and complexity reducing features such as automatic garbage collection. In addition, excellent C# programming environments, which make it easier to write and debug Neon programs, are available on both proprietary and open-source platforms.

There are two components to the Neon system. The first is the Neon language itself, along with the Neon compiler. The second component is an interactive test framework, which allows Neon programs to be interactively tested immediately after they are compiled.

The process of getting from a Neon input program to a final executable is illustrated in figure 3.

Neon programs are implicitly pixel parallel; the order of computation of pixels is left unspecified to allow the code generator maximum flexibility to exploit parallelism on the target architecture. There is an implicit current pixel location at which all operations are performed. User wishing to access pixels other than the current pixel location use the offset function, which returns the pixel at position \( (\text{current}_x + \text{offset}_x, \text{current}_y + \text{offset}_y) \). Convolution, rather than being a built-in primitive operator, is performed by using a series of calls to the offset function.

A simple Neon program, which computes a Laplacian of Gaussian high-pass filter, is shown below. It takes one input image and produces one output image. The bit width of input and output channels must be defined; currently the supported bit widths are 8-bit unsigned, 16-bit unsigned, and 32-bit float\(^7\). All intermediate images are 32-bit float, by default. Users can also define scalar parameters as arguments to the Neon function using the ScalarParameterVariable object type. Minimum, maximum, and initial values for scalar parameters can be set and these values will be displayed on the interactive test framework.

```csharp
public static Im laplacian() {
    // Variable Declarations
    In16Bit r = variable16Bit("r");
    ScalarParameterVariable alpha = "alpha";
    alpha.setRange(0.0F, 20.0F);
    alpha.initialValue = 2.0F;
    // End Variable Declarations
    Const[,] colFilter = laplacianFilterKernel();
    Im rl = alpha * convolve(r, colFilter);
    Im result = new Im(out16Bit(rl));
    return result;
}
```

In the Laplacian code example, the function convolve\(^8\) is a Neon library function, but it is not a primitive operator; it is itself written in the Neon language.

4.1 Optimizations

Neon performs two types of optimizations: generic and target specific. Generic optimizations are done in the first optimization pass and include common subexpression elimination, algebraic simplification, and minimization of the number of intermediate images. Target specific optimizations depend on the architectural characteristics of the target processor and include data packing and unpacking to maximize cache coherence, restructuring convolutions to minimize I/O, and precomputing offset indices.

Before an expression is created, its hash code is used to see if it already exists. If it does, the existing expression is used, otherwise a new expression is created. Commutative operators, such as + and *, test both orderings of their arguments.

Algebraic simplification is performed as the expression graph is being created. This approach is more efficient than performing graph rewriting after the entire expression graph has been created. It is also powerful enough to handle the most common and useful algebraic simplifications. For example, the constructor for the * operator\(^*\) performs the following symbolic algebraic simplifications:

\[
\begin{align*}
a \ast 1 & \rightarrow a \\
a \ast -1 & \rightarrow -a \\
a \ast 0 & \rightarrow 0 \\
c_0 \ast c_1 & \rightarrow \text{Const}(c_0 \ast c_1)
\end{align*}
\]

where \( a \) is a variable argument to the * operator, \( c_0, c_1 \) are constant arguments to the * operator, and \( \text{Const}(\cdot) \) is the constructor for the Constant class, which creates a new node representing a constant value. The algebraic simplifications currently performed for all operators are:

\[
\begin{align*}
a \ast 1 & \rightarrow a \\
a \ast -1 & \rightarrow -a \\
a \ast 0 & \rightarrow 0 \\
a / a & \rightarrow 1 \\
a / (-1) & \rightarrow -a \\
a - a & \rightarrow 0 \\
f(c_0) & \rightarrow \text{Const}(f(c_0)) \\
c_0 \ast c_1 & \rightarrow \text{Const}(c_0 \ast c_1) \\
c_0 \pm c_1 & \rightarrow \text{Const}(c_0 \pm c_1) \\
c_0 / c_1 & \rightarrow \text{Const}(c_0 / c_1)
\end{align*}
\]

4.2 Minimizing the Number of Intermediate Images

The most important optimization Neon performs is minimizing the number of intermediate images created. Intermediate images only

---

\(^7\)8,16, and 32-bit for GPU code; 16 and 32-bit for CPU code.

\(^8\)See the supplemental materials for source code.

\(^9\)See supplemental materials for source code.
need to be computed when a node has two or more Offset node ancestors which have different offset values. To see why this is true, look at the following code fragment

```c
Input16Bit r;
Im temp = r*r;
Im result = offset(temp,0,1)+offset(temp,0,2);
```

which might naively be translated to

```c
foreach (pixel in the image) {
    temp[i,j] = r[i]*r[i,j];
    result[i,j] = temp[i+1,j] + temp[i+2,j];
}
```

This code is not correct because the order in which pixel values are computed is not specified in Neon. Suppose pixels are computed from the bottom of the image to the top. Then, at the point where we want to assign a value to result[i,j], the values temp[i+1,j+0], and temp[i+2,j+0] will not have been computed yet. It would be possible to recursively expand the subgraph for each offset, resulting in code like this:

```c
foreach (pixel in the image) {
    result[i,j] = r[i+1,j]*r[i+1,j] + r[i+2,j]*r[i+2,j];
}
```

but this repeats all of the subgraph computation as many times as there are offsets\(^\text{10}\). Even worse, if offsets are nested the computation increases exponentially with the depth of the nesting.

Instead of replicating computation, the Neon graph optimizer computes the minimal number of intermediate images necessary to eliminate all offset conflicts, and inserts Intermediate nodes into the graph where necessary, as in figure 4. Algorithm 1 shows how to find intermediate nodes\(^\text{11}\). The function `findIntermediates` is called with a list of all roots of the graph. In the final code generation stage, each Intermediate node will cause code to be created that writes an intermediate image.

### 4.3 Target Specific Optimizations

The compiler performs many target specific optimizations. We will just mention a few here that each improved performance significantly. For CPU code generation we use OpenMP parallel constructs to parallize computation across scanlines. Image processing is non-blocked, and images are stored in planar, rather than interleaved, form because this maps much better to the SSE instruction set. Intermediate image nodes cause multiple processing loops, called phases, to be generated. Each loop may have operations on an arbitrary number of images so long as they do not have dependencies on intermediate images which have not yet been computed in a previous phase.

The compiler generates code to up convert low bit resolution data on-the-fly as it being read read and to downconvert it on writes. This eliminates the need to create temporary images solely for the purpose of storing data upconverted from low to high bit resolution, for example creating a temporary 32-bit image to store 8-bit input data that has been converted to 32-bit data.

If multiple offsets are present in any one phase, the compiler creates code to do reads and writes of multiple pixels in a batch, unpacking data on the fly.

Small separable convolution kernels are transformed into non-separable kernels; the extra computation is more than outweighed by the reduction in memory data transfer.

In DirectX10 integer indexing is supported for the `Load` method for indices between -7 and 8. The compiler automatically generates precomputed integer indices for offsets that lie within this data range.

### 5 Results

In section 5.1, we describe the benchmarks and the benchmark setup. In section 5.2, we compare Neon code executing on the GPU to Neon code executing on the CPU, and draw conclusions about the FLOPS/IO ratio necessary to get peak performance. In section 5.3, we compare Neon code to three commercial competitive systems: Adobe’s Pixel Bender, Apple’s Core Image, and Microsoft’s WARP. In some sense, this is not an apple to apple com-
Algorithm 1 Finding the minimum number of intermediate images.

```vbnet
findIntermediates(nodeList, intermediates)
offsets: Dictionary<node, Offset>
visited: Dictionary<node, bool>

while nodeList not empty {
    if (node is Offset) offsets[node] = node
    foreach (child of node) {
        if (offsets[child] == NULL)
            offsets[child] = offsets[node]
        else if (offsets[child] != offsets[node])
            intermediates[child] = true
            visited[child] = true
    }
    nodeList = visited.Keys
}
```

The CPU back-end code generator currently only supports 16 and 32-bit per channel for input and output, while the GPU supports 8, 16, and 32-bit. Support for the latest DirectX features.

5.1 Benchmarking Conditions

Six real-world image processing benchmarks were written in Neon, Core Image, and Pixel Bender. These were tested on CPU and GPU processors:

- **unsharp** (figure 5b), uses color space conversion from RGB to YIQ, and separable filtering with a 5x5 Gaussian to increase the apparent sharpness of an image;
- **contrast** (figure 5c) uses the algorithm of [Moroney 2000], involving separable low-pass filtering and conditional operators to selectively brighten shadows or tone-down highlights;
- **emboss** (figure 5d) subtracts an offset version of the image from itself to create a pseudo-3D embossed effect;
- **spectral** (figure 5f) transforms from RGB to YIQ color space, but gives the user control over the weights applied to R, G, and B when computing the Y. It then converts back to RGB using the standard weights.
- **separable** (figure 5g), uses a 23x23 separable low-pass Gaussian filter to blur the image;
- **Laplacian** (figure 5h), uses a 3x3 Laplacian of Gaussian filter kernel to compute the edges of an image;
- **repeated x** (not shown) repeatedly converts from RGB to YIQ and back, where x is the number of repetitions.

All the benchmarks are fairly standard image processing operations, except for repeated x. This synthetic benchmark is used to generate a sequence of computations of increasing FLOPS/IO to measure the relative change in performance between the GPU and CPU as a function of the FLOPS/IO ratio.

As described in section 2, Pixel Bender allows users to create custom image processing effects for several Adobe products. It does not generate code that can be linked into a user’s application. Core Image is an image processing system which is supported only on Mac OS X. Core Image subroutines can be called directly from end user code. WARP is Microsoft’s software implementation of the entire Direct3D pipeline. It includes a jitter that recompiles HLSL code, written for the GPU, into multithreaded SSE code which then runs on the CPU. Its primary use is to allow DirectX programs to run on computers which do not have hardware GPU support for the latest DirectX features.
16, and 32-bit I/O. When comparing the relative performance of the CPU and GPU code, we use 16 bit I/O.

All intermediate images are 32-bit floating-point for both the GPU and CPU codes. GPU benchmarks were run on the ATI Radeon HD 5870, ATI Radeon HD 4850, and NVIDIA GeForce 470 GTX. CPU benchmarks for Core Image comparisons were run on an iMac 27", with a Core i7 860 @ 2.8GHz, 8GB RAM DDR3 @ 1067MHz, with an ATI Radeon HD 4850 with 512MB of RAM. All other CPU benchmarks were run on a Dell Studio XPS workstation with 12GB of RAM DDR3 @ 1067Mhz and a 2.93 GHz Intel Core i7 940. All four cores were used.

Because of differences in the GPU and CPU architectures, the number of floating-point operations per pixel is not exactly equal for each benchmark, as can be seen in table 4. The CPU code has to convert from 16-bit to 32-bit data on input and output; for very small programs this extra overhead causes the CPU programs to have a higher FLOPS/pixel count than the GPU. On the GPU, this conversion is performed automatically by the hardware. This difference in computation is generally quite small relative to the total work done in the benchmark.

GPU timings include only the cost of doing the computation on the GPU, not the significant round-trip cost of transferring the image from CPU main memory to the GPU and back. For the GPU with the fastest transfer rates, the NVIDIA GeForce 470 GTX, the round trip overhead reduces the maximum possible frame rate to no more than 350fps, more than an order of magnitude slower than this processor is capable of computing results.

The benchmarks Laplacian, spectral, and emboss do relatively little computation and have the lowest relative GPU performance. Their framerate on the GPU is nearly identical even though Laplacian does roughly one third the computation of spectral; a strong indication that these benchmarks are limited either by GPU I/O bandwidth or by driver overhead.

The benchmark repeated x, used for the plots in figures 7, 8, and 9, lets us examine how computational speed changes as a function of the FLOPS/I0 ratio. As the number x of repetitions increases, the FLOPS/I0 ratio increases. We ran benchmarks with up to 80 repetitions, which perform about 2200 arithmetic operations per pixel.

Table 4: Comparison between the GPU and CPU: floating point operations per pixel and effective FLOP rate. Because of extra indexing calculations for the GPU code sometimes the FLOPS/pixel is higher than for the CPU. Conversely, because the CPU has to convert from 16-bit to 32-bit some simple programs have more FLOPS/pixel than for the GPU.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Core i7 940</th>
<th>HD 5870</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsharp</td>
<td>70</td>
<td>90</td>
</tr>
<tr>
<td>spectral</td>
<td>31</td>
<td>31</td>
</tr>
<tr>
<td>separable</td>
<td>102</td>
<td>115</td>
</tr>
<tr>
<td>Laplacian</td>
<td>25</td>
<td>13</td>
</tr>
<tr>
<td>contrast</td>
<td>89</td>
<td>114</td>
</tr>
<tr>
<td>emboss</td>
<td>22</td>
<td>22</td>
</tr>
</tbody>
</table>

12Recall from table 1 that main memory bandwidth on the CPU is 18 GB per second.

Figure 6: Ratio of speed of Neon GPU code vs. Neon CPU code: 16bit RGB input and output.

Figure 7: CPU speed, in GFLOPS, measured with the repeated synthetic benchmark. More repetitions increases the FLOPS/I0 ratio. Non-synthetic benchmarks have FLOPS/I0 ratio roughly equal to repeated I.
50% of peak performance

Figure 8: HD 5870 GPU speed, in GFLOPS, measured with the repeated synthetic benchmark. More repetitions increases the FLOPS/IO ratio. Non-synthetic benchmarks have FLOPS/IO ratio roughly equal to repeated 1.

of IO, far higher than any of the standard image processing benchmarks.

On the Core i7 architecture, it is possible, under ideal conditions, to execute one multiply-add per clock cycle. The processor has four cores and there are four 32-bit floating-point words in each SSE word. The peak possible 32-bit floating-point performance therefore is: clock-speed × 2 × 4 × 4. For the benchmark machine, this results in 93.7GFLOPS. As can be seen in figure 7, the CPU floating-point performance increases rapidly between repeated 1 and repeated 5, and then levels off at close to the peak possible performance: repeated 20, 40 and 80 achieve 87%, 90%, and 91% of peak, respectively. Unfortunately, the FLOPS/IO ratio of the more realistic benchmarks are roughly equal to repeated 1, significantly less than required to achieve peak performance.

We show just one graph for the GPUs since the general trend for all three is similar. The peak advertised performance of the Radeon HD 5870 GPU is 2.72TFLOPs; the GPU never reaches this peak in our benchmarks (see figure 8), although it does approach 50% of peak theoretical performance. The slope of the graph is such that performance appears to still be increasing at 80 repetitions. This is already an unrealistically high FLOPS/IO ratio for image processing operations, so we did not test pass this point.

In spite of the much higher peak floating-point performance of the Radeon HD 5870 GPU, the ratio of GPU to CPU speed is relatively low for realistic image processing benchmarks. Even for the repeated 5 benchmark, which has a significantly larger FLOPS/IO ratio than any of the benchmarks typical of real-world applications, the GPU/CPU speed ratio is only 6. For these benchmarks, the GPU/CPU speed ratio appears to be more closely related to the ratio of memory bandwidth than to the ratio of floating-point performance. This is not entirely surprising given the low FLOPS/IO nature of typical image processing operations.

5.3 Comparison between Neon, Pixel Bender, Core Image, and WARP

Neon GPU code was faster than Pixel Bender GPU code for all 6 benchmarks, by a minimum of 2.2x and a maximum of 7.8x. On three of the six benchmarks, Neon code running on the CPU outperformed Pixel Bender running on the fastest GPU. Neon CPU code was roughly two orders of magnitude faster than Pixel Bender CPU code.

On four of the six benchmarks, Neon GPU code was faster than Core Image GPU code, by a minimum of 1.4 and a maximum of 2.5. Neon CPU code is almost one order of magnitude faster than Core Image CPU code. WARP has surprisingly good performance on the CPU, by comparison to Core Image and Pixel Bender, but Neon CPU code was faster than WARP on all 6 benchmarks by a minimum of 3.8 and a maximum of 7.8.

These results are surprising in several ways. First, properly optimized CPU code can be competitive with code running on a GPU, except when the GPU is state-of-the-art. While the fastest GPUs sold today execute our benchmark image processing operations many times faster than similarly priced CPUs, the majority of computers are sold with relatively low-speed GPUs; many have integrated graphics chips with performance far lower than the lowest performance graphics card we tested. Second, the difference in speed between the GPU and CPU appears to be more closely related to the difference in memory bandwidth than to the difference in floating-point performance. For example, the peak floating-point performance of the ATI Radeon HD 5870 and the Intel Core i7 940 differ by a factor of nearly 30. But the measured GPU/CPU speed ratio for real world image processing programs written in Neon is less than 6, which is close to the memory bandwidth ratio of 5.7. Finally, Neon code running on the CPU can, at times, outperform Pixel Bender or Core Image code running on the GPU, possibly due to the overhead of the on-demand rendering paradigm employed by these systems. Neon CPU code dramatically outperforms Pixel Bender, Core Image, and WARP.

Table 5: Comparison of Neon GPU executables vs. Adobe Pixel Bender. All speeds are in fps. Bold numbers indicate fastest times. Bold red numbers indicate Neon CPU code ran faster than Pixel Bender GPU code.

<table>
<thead>
<tr>
<th>Operation</th>
<th>470 GTX</th>
<th>HD 4850</th>
<th>Core i7 940</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Neon</td>
<td>P</td>
<td>B</td>
</tr>
<tr>
<td>unsharp</td>
<td>1575</td>
<td>341</td>
<td>653</td>
</tr>
<tr>
<td>spectral</td>
<td>5875</td>
<td>806</td>
<td>1640</td>
</tr>
<tr>
<td>separable</td>
<td>842</td>
<td>380</td>
<td>383</td>
</tr>
<tr>
<td>Laplacian</td>
<td>4819</td>
<td>652</td>
<td>2462</td>
</tr>
<tr>
<td>contrast</td>
<td>1149</td>
<td>173</td>
<td>489</td>
</tr>
<tr>
<td>emboss</td>
<td>5907</td>
<td>752</td>
<td>2616</td>
</tr>
</tbody>
</table>

*The Pixel Bender tool doesn’t report frame rates below about 5 fps so we had to manually estimate these frame rates.
6 Conclusion

The Neon image processing language strikes a good balance between generality and speed of execution. Neon programs are easy to write, easy to understand, and hide all low level architectural details from the programmer. Many common image processing operations can be expressed entirely in the language and the resulting executables have excellent performance. For GPU executables, Neon outperforms Core Image and Pixel Bender by factors of anywhere from 1.4 to 7.8. For CPU executables, Neon outperforms Core Image by almost an order of magnitude, Pixel Bender by two orders of magnitude, and WARP by a factor of roughly 5.

References


<table>
<thead>
<tr>
<th>Operation</th>
<th>HD 4850</th>
<th>Core i7 860</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsharp</td>
<td>653</td>
<td>475</td>
</tr>
<tr>
<td>spectral</td>
<td>1640</td>
<td>968</td>
</tr>
<tr>
<td>separable</td>
<td>383</td>
<td>517</td>
</tr>
<tr>
<td>Laplacian</td>
<td>2462</td>
<td>978</td>
</tr>
<tr>
<td>contrast</td>
<td>434</td>
<td>707</td>
</tr>
<tr>
<td>emboss</td>
<td>2616</td>
<td>1078</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>Core i7 940</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsharp</td>
<td>479</td>
</tr>
<tr>
<td>spectral</td>
<td>1407</td>
</tr>
<tr>
<td>separable</td>
<td>231</td>
</tr>
<tr>
<td>Laplacian</td>
<td>1038</td>
</tr>
<tr>
<td>contrast</td>
<td>299</td>
</tr>
<tr>
<td>emboss</td>
<td>1127</td>
</tr>
</tbody>
</table>