Abstract—In recent years, biologically-inspired visual object recognition algorithms — those that aim to mirror the computations performed by the brain's visual system — have emerged as exceptionally promising candidates in object and face recognition research, achieving impressive performance on a range of object and face recognition tasks. While these algorithms typically require a large number of operations per image analyzed, recent advances in many-core parallel computing hardware have enabled practical consideration of relatively large biologically-inspired systems. Here, we focus on a key operation found in this class of models, parallel convolution, and explore the design-space of implementations. Specifically, we compare CPU, GPU and FPGA implementations, spanning SISD, SIMD, and MISD parallelization regimes. We find exceptional speed-ups are possible with GPUs; however, we also find that FPGAs — and MISD configurations in general — remain competitive in a number of domains, particularly when power and space considerations outweigh price.

I. INTRODUCTION

Humans are able to recognize objects in the visual environment in a manner that is effortless, fast, and robust. However, implementing this ability in machines has proven to be a daunting task, and remains an active, open area of research in computer vision and machine learning [1, 2]. One promising line of research has been to look for inspiration in building vision algorithms that borrow computational ideas and structure from the brain. These so-called “biologically-inspired” vision approaches have fielded impressive results in recent years, yielding state-of-the-art or near state-of-the-art recognition performance on a variety of object and face recognition tasks [2, 6].

One practical hurdle in the use of biologically-inspired algorithms is that they, like the natural systems they emulate, employ highly parallel architectures of millions or billions of simple processing units. A consequence of this fact is that a large number of operations are required per scene to be identified, which has historically made it cumbersome to work with even modestly sized systems using traditional serial computing hardware. However, the recent emergence of many-core parallel computing hardware has led to a disruptive shift in the speed with which large-scale biologically-inspired systems can be evaluated, and this has in turn led to discovery of new, highly effective biologically-inspired algorithms using GPU hardware [3, 4].

In this paper, we specifically consider biologically-inspired algorithms that roughly belong to the sub-class known as “convolutional” neural networks [5,7]. These architectures consist of a hierarchy of multiple stages of processing, with each stage being comprised of a set of parallel convolutions with a bank of filter kernels, followed by a series of nonlinear operations on the result. The particular family of models that we consider here is described by Pinto et al. [5] and have been shown to yield good results in a variety of object and face recognition tasks. Because the convolution usually occupies the largest fraction of computation in these models [5], we will focus our analysis on this particular sub-operation. In contrast to “regular” 2D convolution operations where a single kernel is convolved with an input image, here a large number (16-256) of filter kernels are applied to the same image, enabling extra degrees of freedom for developing algorithms that can make optimal use of the compute resources of a given parallel architecture. We also focus our attention on fixed-point arithmetic, since natural neuronal processes have a fundamentally limited dynamic range, and because the algorithms belonging to this family can generally function in a fixed-point mode.

While previous work has demonstrated the success of single-instruction, multiple data (SIMD) implementations of this class of algorithms on Graphics Processing Unit (GPU) hardware [5], here we take a step back and consider the design space of possible parallel implementations for this algorithm, spanning single-instruction, single data (SISD), SIMD, and multiple-instruction, single data (MISD) parallelization regimes, on CPU, GPU, and field programmable gate array (FPGA) hardware, respectively. In particular, we weigh the relative advantages and disadvantages of each architecture, both in terms of actual performance achieved in implementations on each kind of hardware, and in terms of what could be theoretically possible, if one were given access to an arbitrary custom computing machine (e.g. an ASIC implementation). Consistent with previous reports, we find that large speed-ups and excellent hardware resource utilization are possible with GPU implementations. However, we also show that a pipelined, MISD FPGA implementation remains competitive along a number of dimensions, particularly where power and size are important, enabling good FPGA resource utilization while avoiding memory bandwidth walls. Finally, we discuss how each of these implementations and parallelization regimes could fit into a larger space of design requirements, ranging from low-power embedded systems to high-performance computing clusters.
A. Parallelization Regimes

The four types of parallelization regimes that make up the Flynn taxonomy are SISD, SIMD, MISD and multiple instruction multiple data (MIMD) [8]. In Figure 1a, we present SISD, also known as the Von Neumann architecture. SISD is the basis of modern sequential CPU architecture. SIMD is integrated in a single compute node (CN) through local system bus. There are three different types of PEs in a single CN: an Intel Xeon quad-core CPU 2GHz, a NVIDIA Tesla C1060 card and a Xilinx Virtex 5 LX330 FPGA hosted on an ADM-XRC-5T2 card. SISD implementation executed on the CPU is written using the C programming language, while SIMD implementation is written using CUDA version 2.3 [10] for execution on the GPU. The MISD FPGA implementation is written using the VHDL programming language. FPGA devices are configured and controlled through the Alpha Data ADMXRC driver APIs [11]. The communication between tasks across different PEs in the same CN is based on shared memory Inter Process Communication (IPC) framework. A resource management (RM) program is created for coordinating task activities, data movement and hardware allocation.

C. Algorithm

As discussed above, we chose to focus our efforts on the parallel convolution sub-operation, as this operation accounts for up to 89% of the computation in relevant models [5,7]. The direct technique for convolution was used, which incorporates a kernel $h$ and uses linear operations to calculate the resulting value in the output image pixel $g$ as a linear combination of brightness in a local neighborhood of the pixel $f(i,j)$ in the input image.

$$g(i; j) = f(i; j) \otimes h(x; y) = \sum_{i-m}^{m} \sum_{j-n}^{n} f(x - i)(y - j) \times h(i, j)$$

For the purpose of this investigation, filter bank of 16 convolution kernels are used, where each kernel is 3x3 pixels. In practice, larger kernel sizes are sometimes used in convolutional networks [5,7]; however, a 3x3 kernel size is used here due to limitations on the number of multipliers in FPGA chips available in our development system. Larger kernel sizes could be similarly implemented using available FPGA chips with more hardware multipliers. While it is possible to implement a convolution as a multiplication in the Fourier domain, all of the implementations considered here used the “direct” approach, which is usually faster for small kernel sizes and large images, as is the case for this application.

III. RESULTS

A. SISD implementation on a CPU

Our reference CPU implementation of 2D convolution is written using the C programming language. It is compiled using the GCC 4.2.4 compiler. We implement it on an Intel 2GHz dual quad core CPU running the Ubuntu 9.04 operating system. Our best implementation achieves 0.65 GOPS for all scene sizes. As this implementation is a reference SISD implementation, no effort was made to take advantage of the CPU’s SIMD capabilities.

B. SIMD implementation on a GPU

To explore SIMD implementation on GPU hardware, we develop a series of three implementations, starting with a naïve implementation, and moving to successively more optimized implementations that achieve better utilization of the GPU’s compute resources.

Our first implementation of SIMD on a GPU is a naïve implementation of 2D convolution that does not use shared memory. It achieves up to 257 GOPS performance on a 1000x1000 frame. Our second implementation makes better utilization of the data level parallelism and faster memory bandwidth by using shared memory to store pixels; this implementation achieves 4.62 GOPS for a 100x100 frame, and up to 401 GOPS for a 1000 x 1000 frame. For small frame sizes we do not achieve proper utilization of SMs. Thus, in our third implementation, we extended our SIMD implementation by executing multiple 2D convolution kernels in parallel. In doing so we achieve 256 GOPS for a 100x100 frame and over 473 GOPS for a 1000x1000 frame. In each of our shared memory implementations, almost 55% of the time is spent in transferring data to and from shared memory.
C. MISD implementation on a FPGA

A common paradigm for instruction-level parallelism is pipelining. In this approach, pixels are loaded into a pipeline sequentially, such that multiple operations are effectively applied to the same piece of data (in this case, a pixel value) at once.

The flow of the proposed algorithm is illustrated in Figure 2. We perform 16 parallel 3x3 2D convolutions on each input image. To achieve this in a pipeline, kernel coefficients for all 16 kernels are pre-loaded into registers, while pixels are loaded in from the left. Each clock cycle, a new pixel is loaded in, all the pixels in the pipeline are shifted right and the rightmost pixel is discarded. Every clock cycle the pixels are multiplied with the corresponding kernel coefficients and the results of the convolved pixel.

In our FPGA implementation, we use external memory to store each incoming input image. The memory controller for accessing the external memory is part of the Alpha Data API [11]. Storing image data in external RAM allows for the handling of frames greater than 1024x1024 in size. We explicitly avoid using BRAM to store image data because its results in frame data being distributed across the FPGA chip, requiring long routing wires that, in turn, reduce the maximum possible clock frequency. We use the inbuilt DSP48E slices to carry out the multiplication and addition. In our implementation, once the pipeline is filled, we read one pixel and output 16 pixels per clock cycle. However, we only have four banks of memory and each bank can supply 128-bit in one clock cycle. We can read/write 8 pixels from each bank and so we need at least two banks for output. We however use only one bank for input.

Our implementation is flexible in that we can change the input frame size and kernel values at runtime. The kernel size, however, must remain constant. The input frame is first stored in external memory. The address generation logic then gathers the pixels from external memory and feeds the DSP48E slices. Similar address generation logic stores the final pixels in external memory. Currently each individual instance of our 2D convolution core is capable of operating at 330MHz. However, our initial prototype of the integrated solution, which consists of 16 2D convolution cores working in parallel, operates at just 200MHz, due to as-of-yet unresolved practical generation issues. However, we expect to operate the complete solution at, or near, 330MHz and are currently working towards this solution. Our implementation uses 24,428 Xilinx Slices, 22,281 Xilinx LUTs and 144 DSP48Es.

D. Comparisons of GOPS, power and silicon area efficiency

In Figure 3 we present the achieved GOPS for each our implementations, along with the theoretical maximum performance (which is typically unattainable), for each platform that we considered. Not surprisingly, our reference SISD CPU implementation achieved the lowest performance, 0.65 GOPS, approximately an order of magnitude below the chip’s theoretical maximum (5.3 GOPS) [5]. Our best GPU implementation achieved over 470 GOPS, approximately half of the maximum theoretical performance of 933 GOPS [12]. Finally, our FPGA implementation on the Virtex 5 LX330 achieved a substantially lower performance of 95 GOPS. It should be noted, however, that the Virtex 5 chip used here is one generation old, and belongs to the less DSP-heavy “LX” subfamily. The newer Virtex 6 SX240 FPGA offers over 2000 DSP slices (as compared 192 in the LX330), and theoretically higher clock speeds [13]. Given that the processing of a given image can be easily decomposed into multiple parallel pipelines to take advantage of more multipliers if they are available, we predict that an FPGA implementation on DSP-oriented Virtex 6 could potentially achieve a greater than 10x speed-up relative to the Virtex 5 LX330 used here. Interestingly, this would place the raw performance of the FPGA-based design on par with, or perhaps even exceeding that of the GPU.

Finally we consider performance with respect to the various costs associated with each implementation. The first category of cost we consider is price of the parallel computing machines. In the case of our GPU and FPGA systems, we also include the costs of the host CPU that hosts these processors. The price of the GPU system is US$1,130 [14] while the price of the GPU system excluding the host system is US$ 1,220 [12]. The Alpha Data FPGA board used is priced at US$ 5,130 including the communication with the manufacturer). In Figure 3a we plot frame size versus GOPS per unit of price. We note that for all frame sizes, the GPU achieves substantially better price / performance ratio as compared to the other options considered here.

Other potentially important considerations, particularly in embedded applications, are the power requirements and silicon area used in each design. Thus, we next consider GOPS achieved with respect to processor power usage. The Intel E5405 uses 80W of power [15] while the NVIDIA Tesla C1060 uses 187.8W of power [16] and the Alpha Data FPGA board uses 25W of power [11]. In Figure 3b, we plot the frame size Vs GOPS per unit of power. We note that for every level frame size, the FPGA offers significantly higher GOPS per unit of power. The FPGA maintains a constant 3.8 GOPS per watt. The GPU offers a maximum of 2.5 GOPS per watt for frame size 1000x1000. The CPU on the other hand offers a constant 0.008 GOPS per watt, which is the lowest among all the platforms.
Finally, in Figure 3c, we plot GOPS per unit area of silicon used for all our implementations. The die size of Intel E5405 is 214 mm² [17] while the die size of Tesla C1060 862 is mm² [16]. The die area of the FPGA solution was calculated as per the methodology detailed in [18]. Again, the FPGA implementation stands out where size is important.

IV. DISCUSSION

We have described a preliminary design-space exploration for a key component of a useful class of biologically-inspired object recognition systems. Because the computations performed by such systems are highly parallel in nature, they are ideally suited to acceleration via the growing armamentarium of parallel computing tools that have become available in the last decade.

In recent years, GPUs have taken the world of parallel computing by storm, in many cases displacing FPGA-based accelerators in high-performance computing context. Our results do not argue against this trend – our GPU-based implementation yields excellent performance, at an exceptional price to performance ratio. It is worth noting, however, that our FPGA posts a respectable performance figure, especially considering that the FPGA we used occupies the low to mid end of the Xilinx range with respect to DSP slices, which are by far the rate-limiting resource in our implementation. Using the latest-generation, DSP-oriented FPGAs, there is a good chance that it might be possible to match the performance of current generation GPUs with respect to raw performance, if not price/performance ratio. Our FPGA implementation also stands out where power consumption and/or silicon area are important, achieving the highest performance per watt and square millimeter of silicon. While these factors are likely to be less important than price in a server rack setting, they argue strongly for viability of FPGA-based design in embedded settings.

REFERENCES


