DESIGN AND OPTIMIZATION OF A STENCIL ENGINE

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

John S. Brunhaver II
June 2015
© 2015 by John S. Brunhaver, II. All Rights Reserved.
Re-distributed by Stanford University under license with the author.

This work is licensed under a Creative Commons Attribution-Noncommercial 3.0 United States License.
http://creativecommons.org/licenses/by-nc/3.0/us/

This dissertation is online at: http://purl.stanford.edu/zd923qy7249
I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Mark Horowitz, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Christos Kozyrakis

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Oyekunle Olukotun

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost for Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format. An original signed hard copy of the signature page is on file in University Archives.
Abstract

Application specific processors exploit the structure of algorithms to reduce energy costs and increase performance. These kinds of optimizations have become more and more important as the historical trends in technology scaling and energy scaling have slowed or stopped. Image processing and computer image understanding algorithms contain the kinds of embarrassingly parallel structures that application specific processors can exploit. Further, these algorithms have very high compute demands, which makes efficient computation critical. So these specialized processors are found on many SoCs today. Yet, these image processors are hard to design and program, which slows architectural innovation.

To address this issue we leverage the fact that most image applications can be composed as a set of “stencil” kernels and then provide a virtual machine model for stencil computation onto which many applications in the domains of image signal processing, computational photography, and computer vision can be mapped.

Stencil kernels are a class of function (e.g. convolution) in which a given pixel within an output image is calculated from a fixed-size sliding window of pixels in its corresponding input image. This fixed window in the input data, where each data element is reused between concurrent computations, allows for a significant reduction in memory traffic through buffering and provides much of the efficiency in specialized image processors. Additionally, the predictable data flow for stencil kernels, allows for the producer consumer relationships between stencil kernels in large applications to be statically determined and exploited, further reducing memory traffic. Finally, the functional nature of the computation and the significant number of times it is invoked allows for the implementation of the computation to be highly optimized.

Stencil kernels play a recurring role in image signal processing, computer vision, and computational photography. Any process that creates a filter, constructs low level image features, evaluates relationships of nearby pixels or features, etc. is implementable as a
stencil kernel. Many applications in the domain image processing and understanding are built by cascading these operations (e.g. filtering noise, looking for local features and local segments, then localizing regions and objects from those segments and features). These applications also play a significant role in society, whether it is to automate the home, car, or factory or to improve the capabilities of our mobile devices in capturing and understanding the world around us. While the computation model may seem restrictive and domain specific any improvement in the efficiency of this computation for this domain would permeate many fields and society increasing the capability and decrease the cost of innovation and progress.

When applications are written in a domain specific language restricted to stencil computation, it can be compiled to the stencil virtual machine model proposed in this thesis. This model allows for an application’s behavior to be specified without knowledge of the underlying system implementation. Conversely, such a model allows for a great degree of flexibility in the implementation of that underlying system, which provides opportunity for optimization. The input to this virtual machine model is an intermediate language called Data Path Description Assembler (DPDA), which represents a compiler target for high level languages.

While many hardware-software systems implement the virtual machine and execute DPDA, this thesis presents a method to generate fixed function hardware from DPDA code. The resulting hardware is two orders of magnitude more efficient than a comparable CPU or GPU implementation. This hardware generator greatly reduces cost of designing customized engine for new imaging applications, and also serves as a critical reference for research exploring the overheads of more flexible compute engines.
Acknowledgments

While at Stanford I have had the benefit of many mentors, who have offered me guidance while I grew as a researcher. Specifically, I really appreciate the time (and patience) of Mark Horowitz and Ofer Shacham that not only provided me significant support in the development of this work but provided a mirror to my actions so that I could grow. Additionally, I am grateful for all of the opportunities that Mark has given me and the example he has provided that has helped me to begin my own research program. I also significantly valued all of the effort that Mark put into making his research group feel like a community.

I also appreciate the help and review that my committee, Christos Kozyrakis, Kunle Olukotun, Pat Hanrahan, and Virginia Williams have provided. Additionally, I am grateful for the encouragement that Pat Hanrahan offered me earlier in my graduate career and the great feedback he has provided in this work. I am also thankful for the support that Kayvon Fatahalion provided in introducing me to real-time graphics research. Finally, Subhasish Mitra provided me support and understanding when I first arrived at Stanford, and I significantly value the lessons I learned during that time.

This work was made possible by the financial support of Qualcomm through the Qualcomm Innovation Fellowship and the DARPA Perfect program through the Specialized Extremely Efficient Computing (SEEC) grant.

In addition to the faculty at Stanford, members of industry like Bill Mark, Kurt Akely, David Luebke, and Steve Keckler took time out of their schedules to provide critical feedback of my work in progress and encouraging words of support. Additionally, John Owens at UC Davis and Dejan Markovic at UCLA have provided me professional mentorship that has helped many of my recent challenges seem surmountable.

Day to day I have been surrounded by great folks who shared in the trials and tribulations of graduate work. Specifically, Artem, Andrew, Steven, Sameh, Jing, Xuan, Noy, James, Zack, Nicole, Daniel, Christina, Glen, and Mario were brothers and sisters in this
endeavor.

During my time at Stanford I was lucky enough to be able to marry the love of my life, Samantha. She has been there almost every step of the way sharing in the joy, pain, anguish, depression, and elation that is research. I am also grateful for the support of my parents. I am not sure that everyone is lucky enough to have a therapist and a Colonel to provide psychological/martial motivation.

I would also like to apologize to the coffee shops that I have terrorized in the pursuit of putting words to page. Specifically, I will think very fondly of my time at Coupa, Bytes, and Royal.

Finally I will leave you with three pieces of advice that have been indelibly inked into my cranial vault:

- “Fail fast”
- “Wrap it up with a ribbon”
- “Just get it done”
Contents

Abstract v

Acknowledgments vii

1 The Case for Domain Specific Hardware 1
  1.1 Modern microprocessors are power constrained . . . . . . . . . . . . . . . . 1
  1.2 Transistor technology energy scaling has slowed . . . . . . . . . . . . . . . . 5
  1.3 The need to change the computation . . . . . . . . . . . . . . . . . . . . . 7
  1.4 This dissertation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11

2 An Image Processing Stencil Abstraction 13
  2.1 Image signal processing hardware . . . . . . . . . . . . . . . . . . . . . . . 13
    2.1.1 ISP applications are extreme locality compute . . . . . . . . . . . . . . . 14
    2.1.2 ISP implementation of extreme locality computation hardware . . . . . 15
    2.1.3 Fixed function image co-processors reduce CPU overheads . . . . . . . . 18
  2.2 ISP computation can be formally specified . . . . . . . . . . . . . . . . . . 19
    2.2.1 Image processing domain specific languages specify image functions 20
    2.2.2 Spiral generates hardware from extreme locality DSL . . . . . . . . . . 23
    2.2.3 Convolution engine specifies convolutional image functions . . . . . . 24
    2.2.4 Stencil virtual machine model specifies ISP applications . . . . . . . 25
  2.3 Stencil compute is broad and socially significant . . . . . . . . . . . . . . . 30
    2.3.1 Computer vision . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
    2.3.2 Computational photography . . . . . . . . . . . . . . . . . . . . . . . 31
  2.4 Generate hardware for stencil compute . . . . . . . . . . . . . . . . . . . . 32
    2.4.1 High level synthesis of a domain specific language . . . . . . . . . . 32
3 Stencil Engine Generator

3.1 Stencil computation intermediate representation
   3.1.1 Data path description assembler
   3.1.2 Stencil kernel parameter
   3.1.3 Stencil application topology
   3.1.4 Specifying higher dimensional stencil applications
   3.1.5 Intermediate representations for hardware generation
   3.1.6 Intermediate representations of stencil computation

3.2 Generating fixed function hardware
   3.2.1 Stencil engine microarchitecture
   3.2.2 Methodology

3.3 Results
   3.3.1 Hardware cost
   3.3.2 ASIC quality design
   3.3.3 Understanding the (in)efficiency of ASIC design
   3.3.4 Amortizing global memory energy

3.4 Stencil engine contributions and related work

4 Optimization of a Stencil Engine

4.1 Relative marginal cost analysis
4.2 Working set partitioning
4.3 Optimal working set partition size
4.4 Optimal application partition size
4.5 Contribution and related work

5 Conclusion

Bibliography
List of Tables

1.1 Bandwidth to compute ratios of various algorithms .......................... 9
2.1 Image signal processor kernels and descriptions ............................ 16
3.1 Data path description assembler register types .............................. 38
3.2 Data path description assembler register dimensionality ................. 39
3.3 Data path description assembler operators ................................. 39
3.4 Data path description assembler operation specification legend ........ 44
3.5 Data path description assembler arithmetic operation specification .... 45
3.6 Data path description assembler comparison operation specification ... 45
3.7 Data path description assembler predication operation specification ... 46
3.8 Data path description assembler reduction operation specification ... 46
3.9 Data path description assembler bit string operation specification .... 47
3.10 Data path description assembler hardware operation specification .... 47
3.11 Common boundary conditions in stencil kernels ............................ 49
3.12 Stencil kernel parameters .................................................. 49
3.13 Operation count required to amortize DRAM energy ....................... 76
4.1 Marginal cost analysis for optimal application strip width ............... 86
4.2 Marginal cost analysis for optimal application size ......................... 93
# List of Figures

1.1 CPU-DB microprocessor total die power vs year of market release .......................... 2
1.2 FPGEN single precision floating point multiply accumulate energy vs performance ................................................. 3
1.3 FPGEN single precision floating point multiply accumulate energy cost per operation vs area cost per performance ................................................. 4
1.4 CPU-DB feature scaling in time .................................................................................. 5
1.5 CPU-DB domain voltage vs feature size ...................................................................... 6
1.6 Energy per bit of various operations ........................................................................... 7

2.1 Abstract demosaic operation .................................................................................... 14
2.2 A simple camera pipeline ......................................................................................... 16
2.3 Canonical image signal processor kernel architecture ................................................. 17
2.4 Image function examples .......................................................................................... 22
2.5 Stencil operation ...................................................................................................... 27
2.6 Stencil down-sample operation ................................................................................ 27
2.7 Stencil up-sample operation .................................................................................... 28
2.8 Cascade of stencil operations .................................................................................. 28
2.9 Fan in and fan out of stencil operations ................................................................... 30

3.1 Data path description assembler 9 × 9 convolution example ........................................ 41
3.2 Data path description assembler sum of accumulated differences example ................. 42
3.3 Data path description assembler bilinear demosaic example ........................................ 43
3.4 Data path description topology definition for simple camera pipeline ......................... 50
3.5 Data path description assembler one-dimensional convolution example ..................... 51
3.6 Data path description assembler five-dimensional convolution example .................... 52
3.7 Stencil engine kernel microarchitecture ..................................................................... 55
Chapter 1

The Case for Domain Specific Hardware

While computer performance has been growing at an amazing rate for the last few decades the power it dissipates has been growing as well. As a result power now constrains the performance of modern microprocessors (Sec. 1.1). This limitation is made more serious because technology scaling has slowed and energy is no longer scaling rapidly (Sec. 1.2). This slowing means that improvements in manufacturing technology will not increase performance at the historical rate. So to continue to scale performance we need to increase the energy efficiency of our microprocessors by orders of magnitude. Given that we have been working on this for over a decade it is unlikely to be feasible, unless we make large restrictions on the type of computation we target (Sec. 1.3). Fortunately, very efficient implementations are possible for important classes of computation. Specifically, this dissertation proposes a virtual machine model for a subset of the domain of Image Signal Processing, Computer Vision, and Computational Photography that can be used to generate energy efficient domain specific hardware (Sec. 1.4).

1.1 Modern microprocessors are power constrained

Total die power (TDP) is limited by the amount of heat that can be extracted from the system for a reasonable cost. For a data center or desktop component, the rate of heat transfer to the environment practically limits components to around 100W or 1W/mm² using forced air techniques[122]. While there are expensive cooling techniques (e.g. water
CHAPTER 1. THE CASE FOR DOMAIN SPECIFIC HARDWARE

Figure 1.1: The total die power (TDP) listed in specification of major microprocessor designs from the Stanford CPU DB [59] is plotted against the time of their market release. Note that specified total die power was a reasonable indicator of active device power dissipation prior to 2004, it is currently only an indicator of maximum average power dissipation. The trend line indicates the median TDP in a three year window.

cooling [46]) that allow for increases in TDP, these cooling techniques are only used when the total cost of ownership is not the most significant design constraint [211]. Laptop cooling, while still using forced air is reduced in efficiency as the number-of and space-for moving components is reduced [122] yielding power constraints of 10 – 40W. Additionally, hand-held devices are constrained by the energy that can be comfortably dissipated into a user’s hand limiting smart phones and tablets to 3W and 10W respectively [10, 11].

Figure 1.1, shows the trend of desktop and server power with year. In the early 2000’s, total die power reached a limit near 100W, where die power stopped growing with time. Many referred to this as the Power Wall [47, 210] and described the changes in architecture to accommodate this new power limit as a Right Hand Turn [167].

To understand this Right Hand Turn, remember that, in a power constrained design, in order for operation throughput to increase, the energy per operation\(^1\) (pJ/op) must

\(^1\)this metric will often be referred to as energy efficiency or energy cost in this thesis
increase as shown in Equation 1.1.

\[
\text{Power} = \frac{\text{Energy}}{\text{Op}} = \frac{\text{Performance} \cdot \text{Energy/Op}}{\text{Operations} \cdot \text{Second}} \cdot \frac{\text{Joules}}{\text{Operation}} \tag{1.1}
\]

This causes a problem with conventional processor design, which tend to follow the law of diminishing returns in its approach to increasing single thread performance.2

For example, to increase the throughput of a single floating point multiply accumulate unit (FMA), the energy per operation of that unit must similarly increase. As seen in Figure 1.2, this cost grows asymptotically making even marginal increases in serial performance or energy efficiency at the extremes incredibly expensive.

---

2 Out of order execution [203], wide instruction issue [149, 48, 201], and branch speculation [68, 184] disproportionately increase energy and area relative to the increase in performance [25]. Modern graphics processors, on the other hand, eschew many of these features [137].
The **Right Hand Turn** solution to increasing performance without increasing power was to redefine performance to focus on throughput and then to increase parallelism [154]. Naturally there are a few issues with this approach, the most important is finding parallelism in real world applications [21]. Luckily, there are many applications with significant amounts of parallelism. For example, computer graphics [40] and the scientific computations popular in the exoscale community [120, 36] including finite element analysis used to predict fluid dynamics [65] or chemistry [28] have provided a practically infinite supply of parallel workloads [83].

![Figure 1.3: Taking the floating point multiply accumulate design [75] already plotted in Figure 1.2, the energy per operation is plotted against the area cost of throughput performance. Note that the Pareto designs in Figure 1.2 are not necessarily the Pareto designs in this figure.](image)

When optimizing a throughput application with parallelism, performance is no longer a good metric, as adding hardware will always improve performance. Instead the trade-off is between the silicon cost of the performance\(^3\) \((mm^2/(op/ns))\) and the energy required to execute the operation. Figure 1.3 plots these two parameters for the floating point multiply

---

\(^3\)this metric will often be referred to as **area efficiency** or **area cost** in this thesis
1.2. TRANSISTOR TECHNOLOGY ENERGY SCALING HAS SLOWED

accumulate unit [75] discussed earlier.

Historically, each successive fabrication technology generation lowered both the energy per operation cost and the area cost of performance. If these trends continued with parallel machines, it would be easy to increase throughput by $8 \times$ at constant area and constant power every five years (Fig. 1.4)[59, 8, 61]$	extsuperscript{4}$

![Figure 1.4: The feature size of each microprocessor from CPU-DB[59] is plotted against the year that microprocessor was released. The trend line shows the semilog change in feature size with time as a reduction in feature size by $2 \times$ every 5 years.](image)

1.2 Transistor technology energy scaling has slowed

Historically, using Dennard’s constant electric field scaling of MOS transistors[61] a technology scaling factor $s$ of $2 \times$ would reduce energy per operation by a factor of $8 \times$ (Eq. 1.3) and area per performance by a factor of $8 \times$ (Eq. 1.4).

---

$	extsuperscript{4}$Assuming constant field scaling[61] and $\frac{1}{2} \times$ feature scaling every five years (Fig. 1.4)
\[ \hat{E} \propto \frac{1}{s^3} E_{old} \propto \left( \frac{1}{s} C_{old} \right) \left( \frac{1}{s} V_{old} \right)^2 \] (1.3)

\[ \frac{A}{f} \propto \frac{1}{s^2} A_{old} / f_{old} \propto \left( \frac{1}{s^2} A_{old} \right) \cdot \left( \frac{1}{s} \frac{C_{old} V_{old}}{I_{old}} \right) \] (1.4)

Figure 1.5: The specified domain voltage of server and desktop microprocessors is plotted against the manufacturing technology feature size for that microprocessors [59].

Unfortunately, voltage scaling has slowed (Fig. 1.5), and thus most of the future gains are derived solely from feature reduction. In this regime a scaling factor of 2\times would reduce energy per operation by a factor of 2\times (Eq. 1.5) and area per performance by a factor of 4\times (Eq. 1.6)[52, 19].

\[ \hat{E} \propto \frac{1}{s^3} E_{old} \propto \left( \frac{1}{s} C_{old} \right) \left( \frac{1}{s} V_{old} \right)^2 \] (1.5)

\[ \frac{A}{f} \propto \frac{1}{s^2} A_{old} / f_{old} \propto \left( \frac{1}{s^2} A_{old} \right) \cdot \left( \frac{1}{s} \frac{C_{old} V_{old}}{I_{old}} \right) \] (1.6)

So today’s power constrained throughput design is limited by the energy efficiency of its
operations, with the result that performance scales linearly with feature size. The difference in energy scaling and area scaling has left many to consider what can be done with the proportionally growing unused portions of the die area [71, 84, 190, 210].

Further there are many indications that the current feature scaling trend where 2× reduction in feature size every fives year (Fig 1.4), is running out. The 2010 ITRS predicted a slowing in feature scaling starting in 2013 [8]. While Intel’s 14nm slip in 2014[54, 95] isn’t a confirmation of these predictions, it could be a new trend in semiconductor manufacturing.

With these scaling trends, we need to do something other than just wait for better transistors, because we could be waiting for a while.

1.3 The need to change the computation

![Figure 1.6: The extremely low cost of short bitwidth operations in 45nm relative to the cost of reading data from DRAM underscores the importance of locality in allowing us to execute tens of thousands of operations per global memory operation. Note that the y-axis is displayed in energy per bit and thus a 6 orders of magnitude (2^{21}×) difference in energy exists between the lower left corner and upper right corner of the plot. The SIMD and RISC costs are based on Tensilica cores.19](image-url)
Since we cannot depend on current manufacturing trends to provide us significant increases in energy efficiency, we need to change the computation to emphasize lower cost operations\(^5\) (Fig. 1.6). Specifically, the goal is to reduce an application’s energy cost to the energy cost of its fundamental work, its arithmetic operations.\(^6\)

When we change the computation to emphasize lower cost operations, we should do four things. First, we should favor computation with a high compute to bandwidth ratio to minimize the number of expensive global memory operations (less than 1 in 4000) of the operation mix. Second, we should favor computation with significant locality and minimal working set to reduce the cost and frequency (less than 1 in 100) of local memory operations. Third, we should minimize the overhead of instruction fetch and decode by unrolling computation in space rather than time. Fourth, we should tailor the arithmetic precision to the computation, hopefully minimizing the use of expensive high precision operations (less than 1 in 100).

The reason we need applications with large number of operations per global memory operations\(^7\) is simple: memory operations are so incredibly expensive that reducing all other energy to zero would have a marginal impact on total energy\([21]\), unless an algorithms performs 1000’s of arithmetic operations per global memory operation. These algorithms are said to have a high compute to memory bandwidth ratio \([37, 111, 159]\). Table 1.1, lists some common kernels and their naive implementation’s average compute-bandwidth ratio complexity.

Few kernels require this large number of operations, so efficient execution engines need to pipeline multiple kernels together to take advantage of producer consumer relationships\([57, 187, 173]\) and increase the aggregate compute to bandwidth ratio. For example, algorithms like rasterization\([34]\) have a relatively modest compute to bandwidth ratio. So in real-time rendering systems it is cascaded in a pipeline which will: displace triangle vertices\(^8\), conservatively reject covered triangles\(^9\), rasterize triangles into pixels\(^10\), calculate the final depth of the pixel\(^11\), eliminate trivially covered pixels\(^12\), calculate the final color of the

---

\(^5\) **Operation**: an atom of work in an application, algorithm, or computation

\(^6\) **Arithmetic Operation**: An atom of work that performs calculation, such as an add, divide, multiply, or comparison

\(^7\) **Global Memory Operations**: an operation that reads or writes to off-chip memory addresses

\(^8\) vertex shading

\(^9\) z-cull

\(^10\) rasterization

\(^11\) interpolation

\(^12\) early-z
## 1.3. THE NEED TO CHANGE THE COMPUTATION

Table 1.1: Bandwidth to compute complexity ratios of various algorithms of problem size $N^d$ where $N$ is some whole number and $d$ is the dimensionality of the problem. Each compute, bandwidth, and ratio is given as a value of average complexity. Additionally, the working set size complexity is listed as well as the complexity of immediate re-use. A value of 1 indicates constant complexity, a value of $N$ indicates a complexity that grows linearly with problem size, and a value of $M$ indicates a complexity that grows with size of the inner loop.

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Compute</th>
<th>Bandwidth</th>
<th>Ratio</th>
<th>Working Set</th>
<th>Imm. Reuse</th>
</tr>
</thead>
<tbody>
<tr>
<td>Axpy[107]</td>
<td>$N$</td>
<td>$N$</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Inner Product[107]</td>
<td>$N$</td>
<td>$N$</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Rasterization[34]</td>
<td>$N$</td>
<td>$logN$</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Ray Tracing[209]</td>
<td>$logN$</td>
<td>$logN$</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>LFIR-1D[147]</td>
<td>$NM$</td>
<td>$N$</td>
<td>$M$</td>
<td>$M$</td>
<td>$M$</td>
</tr>
<tr>
<td>LFIR-2D[147]</td>
<td>$N^2M^2$</td>
<td>$N^2$</td>
<td>$M^2$</td>
<td>$MN$</td>
<td>$M^2$</td>
</tr>
<tr>
<td>LFIR-3D[147]</td>
<td>$N^3M^3$</td>
<td>$N^3$</td>
<td>$M^3$</td>
<td>$MN^2$</td>
<td>$M^3$</td>
</tr>
<tr>
<td>Quick Sort[176]</td>
<td>$N\log N$</td>
<td>$N$</td>
<td>$\log N$</td>
<td>$N$</td>
<td>1</td>
</tr>
<tr>
<td>Bubble Sort[22]</td>
<td>$N^2$</td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
<td>1</td>
</tr>
<tr>
<td>Matrix Mul.[107]</td>
<td>$N^3$</td>
<td>$N^2$</td>
<td>$N$</td>
<td>$N$</td>
<td>1</td>
</tr>
</tbody>
</table>

On the other hand, algorithms whose complexity increases as a function of the problem size $N$ (e.g. sort and matrix multiply) exhibit a potential for incredible compute bandwidth ratios for very large problems sizes without pipelining.

---

13 pixel shading  
14 z-raster-operation  
15 color-raster-operation  
16 also known as convolution
The problem, unfortunately, is the size of the working set\(^{17}\) of a kernel (Tbl. 1.1). When this size exceeds the limited amount of on-die storage, values must be re-read from global memory and intermediate values be stored in global memory. One of the difference between naive implementations of these kernels and their production worthy counterparts are the choices made to trade-off computation-bandwidth ratio for an achievable on-die working set, a process generally referred to as blocking\(^{18}\)[155, 221, 177, 60].

Once the kernel’s working set fits on-die and the global memory operations are minimized, the next challenge is to minimize accesses to local memory as it is two orders of magnitude more expensive than arithmetic operations. Values that are immediately re-used by a kernel, can be stored in local registers amortizing the local memory operation across a larger number of arithmetic operations. Table 1.1 lists the complexity of the immediate re-use for the naive implementation of these kernels. Though, more efficient implementations may make a trade-offs that increase the locality in exchange for an increased working set\(^{72}\).

Having minimized the data memory energy overhead our next goal is to reduce instruction overhead. Given the high cost of an instructions overhead relative to the cost of an arithmetic operations, we should execute many arithmetic operations per instruction. As noted in Figure 1.6, there is a 2000× overhead to executing a single low precision integer op in a RISC instruction. This instruction overhead energy cost is derived from the effort to increment the program counter, predict the branch, predict the branch target, fetch the instruction, decode the instruction, set pipeline control bits, read operands from the bus or register file, place the result on the bus or register file, etc. While it would be reasonable to assume that those overheads for a scalar machine could be evenly divided across each of 16 parallel and identical operations resulting in a 16× increase in the energy efficiency relative to the scalar design, some overhead components increase with that parallelism\(^{102}\) such that there is a 250× instruction overhead for a SIMD-16 architectures.\(^{19}\)

In fact, prior work has pointed out that we will need to do significantly better than fusing a few operations into an instruction and vectorizing those instructions, as reasonable vector widths would require significant operation fusing in order to amortize the SIMD instruction

\(^{17}\) **working set** the number of live memory addresses. A memory address is “live” if the application has used the address and may use the address in the future.

\(^{18}\) This thesis will discuss these issues for stencil kernels in Chapter 4

\(^{19}\) Note that the relative efficiencies for RISC and SIMD are based on values Hameed and Qadeer made available for their 32-bit implementation in the Tensilica framework[82, 162].
overheads\[82, 162\]. The limit of this operation fusing would be to create, in hardware, the entire flow graph for the inner loop of the computation and then stream the data through it. Thus we significantly unroll programs into space rather than time \[104, 62\]. Instead of executing a single instruction with a single operation on a single piece of data, a single “instruction” should invoke a large number of operations repeatedly on a stream of input data\[155, 202, 92, 152\]. There has been some success with these techniques in the green droid work \[205, 190\], but their efficiency gain was limited by the energy costs of memory and the ability to stream computation over the unrolled hardware.

Finally, we should favor cheaper arithmetic operations. A change from large bitwidth operations to short bitwidth operations can provide one to two orders of magnitude improvement in arithmetic operation cost as show in Figure 1.6.

Surprisingly there are a large class of important applications with these characteristics. This thesis first explores this class of application and then describes how these algorithms can be converted into energy efficient hardware implementations.

1.4 This dissertation

Most of these highly local algorithms can be expressed as stencil kernels and play a significant role in the domains of image signal processing, computer vision, and computational photography. Chapter 2 presents a formal definition for stencil kernels, which forms the basis for building our energy efficient compute abstraction. Practically, the work in this thesis builds on top of many ideas in hardware generation and domain specific languages, and Chapter 2 provides an introduction to those concepts.

Chapter 3 shows how an application composed of stencil kernels and defined in a domain specific language could be used as a basis to procedurally generate fixed function hardware. We use an intermediate language called Data Path Description Assembler (DPDA) to arbitrate that front-end to back-end relationship by formally defining the interface to the compute abstraction or virtual machine model. DPDA represents a general intermediate which could be executed in any system that implements the virtual machine model. The generated fixed function represents not only a functional validation of that virtual machine model but a demonstration of its energy efficiency.

As we hinted at earlier (Sec. 1.3), there is a trade-off between the amount of locality and the size of the working set. Chapter 4 presents a general cost model and optimization
framework for selecting the working set size of an application and determining the optimal amount of work in a pipeline of stencil kernels. While, we demonstrate this optimization in the context of fixed function hardware, the framework is general enough that it would apply equally to any implementation of the stencil virtual machine model.

Finally chapter 5 concludes by summarizing the contributions of this thesis work and suggesting future research directions.
Chapter 2

An Image Processing Stencil Abstraction

If the computation must change, we can look to image signal processing (ISP) hardware, common in modern SoCs, as an example for how extreme locality computation achieves incredible energy efficiency (Sec. 2.1). Unfortunately, this hardware has been difficult to design and program, which has slowed innovation in this space. Recent work has explored how algorithms like those found in an ISP can be specified (Sec. 2.2), from which we propose a stencil virtual machine model (Sec. 2.2.4) as the basis for restricting image processing and image understanding applications to its energy efficient subset. That restriction on computation is useful because that efficient subset of image signal processing, computer vision, and computational photography is pervasive (Sec. 2.3). If we have a description for stencil computation, it would be great then, if hardware could be procedurally generated from that description (Sec. 2.4). This process of generating hardware (Sec. 3.2) from these stencil descriptions (Sec. 3.1) is exactly what is described in this thesis.

2.1 Image signal processing hardware

The hardware we generate greatly resembles the recurring designs patterns found in image signal processing hardware. This hardware is principally responsible for translating raw camera sensor data into an accurate and aesthetically pleasing picture in modern SoCs like the applications processors found in hand-held devices (Sec. 2.1.1). The computation, which resembles two-dimensional convolution, contains all of the properties of an extreme
locality algorithm allowing the hardware to achieve high energy efficiency (Sec. 2.1.2). Recent work has examined how this implementation differs from that of a scalar processor and has examined how general computation containing some of the properties of extreme locality computation could be used to procedurally generate efficient hardware (Sec. 2.1.3). This prior work identifies the gaps in both design efficiency and energy cost that must be addressed by any work that seeks to procedurally generate energy efficient hardware from algorithm descriptions.

2.1.1 ISP applications are extreme locality compute

Image signal processing is concerned with computing images from raw sensor data in three major steps. The first set of stages in an image signal processor pipeline deal with correcting the physical non-idealities of the sensor system (eg. lens aberration, silhouetting, shot noise, dead pixels, etc). The second set of stages interpolate the color values to generate a full RGB image from a Bayer mosaic image (Fig. 2.1)[126]. The final set of stages improve many of the aesthetic considerations and resolve any remaining noise issues (eg. chroma noise, color temperature, contrast enhancement).

Each kernel calculates an output pixel based only on a small region of the input image and some constant coefficients. These output pixels are calculated in row major order to match the read pattern of the imaging sensor. Therefore successive output pixels have overlapping regions allowing for the kind of immediate re-use that we referred to in LFIR-2D kernels (Tbl. 1.1). Additionally, successive rows of pixels share lines of pixels in the input image making the total window of re-use or total working set finite and limited.

![Figure 2.1: Abstract demosaic operation](image-url)
For example, Figure 2.1 shows an abstract implementation of the demosaic kernel. A normal sRGB image contains a red green and blue color intensity channels in each pixel. However an image sensor only captures red green and blue intensities at the pixel resolutions. Each pixel’s missing color data must be interpolated from surrounding pixels. This process of converting from Bayer mosaic space to the RGB color space is called demosaic.

Every output pixel in the demosaic is calculated with the interpolation function \( f_w \). The function operates on windows of pixels visualized as a \( 5 \times 5 \) grid of pixels. This pixel window represents the data required to re-sample into the new color space. The total working set for this kernel, determined by the height of the window, is shown to the left as the colored subset of pixels that must be stored. Maintaining this buffer of image lines prevents any redundant access to memory in order to re-fetch pixel values that have already been seen. These input pixels are usually represented in a low precision format like unsigned 8 or 12 bit integers.

While the details may change (eg. function, window size, pixel format), all of the kernels in an image signal processing pipeline conform to this design pattern. From this perspective, a general image signal processing kernel has the extreme locality characteristics of: significant immediate re-use, finite and small working set, functional in execution such that the computation can be unrolled in space, and low precision in operation.

Image signal processors are built by deeply cascading these extreme locality operations, the last extreme locality requirement for energy efficient compute. For example, the Frankencamera\[18, 70, 158, 194\] prototype image signal processing pipeline implements the stages in the order described in Table 2.1.

The ISP’s implementation takes advantage of these extreme locality properties to achieve high energy efficiency by fixing the execution in the hardware.

### 2.1.2 ISP implementation of extreme locality computation hardware

Modern image signal processor hardware is implemented as a deeply pipelined fixed function unit in the un-core\(^1\) of an application processor unit\[74, 198, 197, 88\]. Each component in that fixed hardware serves a specific property of extreme locality computation. Looking at a simple pipeline (Fig. 2.2), each of the three operations - black level correct, demosaic, color correction matrix - are represented as the cascade of functions reading from and writing into local working sets.

\(^1\)Intel jargon for not the core processor where application code runs
### Table 2.1: An ordered description of the processing stages in the prototype image signal processing pipeline designed in this framework for the Frankencamera\[18\]

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>blc</td>
<td>black level correction</td>
</tr>
<tr>
<td>lsc</td>
<td>lens shade correction</td>
</tr>
<tr>
<td>dpc</td>
<td>dead pixel correction</td>
</tr>
<tr>
<td>xtk</td>
<td>cross-talk</td>
</tr>
<tr>
<td>den</td>
<td>denoise</td>
</tr>
<tr>
<td>wbg</td>
<td>white balance gain</td>
</tr>
<tr>
<td>dem</td>
<td>demosaic</td>
</tr>
<tr>
<td>srp</td>
<td>sharpening</td>
</tr>
<tr>
<td>ccm</td>
<td>color correct matrix</td>
</tr>
<tr>
<td>gcc</td>
<td>gamma color correction</td>
</tr>
<tr>
<td>cnr</td>
<td>chroma noise reduction</td>
</tr>
</tbody>
</table>

Figure 2.2: A simple camera pipeline that implements black level correction, demosaic, and a color correction matrix.

The working subset of the images buffered between each kernel is stored in static random access memory arrays. The lines of the image stored in these buffers are the lines that are re-used between each row. Traditionally these buffers are referred to as line buffers\[172, 219, 96\]. The write side interface to the line buffer receives pixels in row major order. The read side interface of the line buffer provides the incremental updates to the window of pixels. This incremental update of pixels is a column of pixels whose height matches the height of the input window.

This incremental data can be combined with the prior overlapping window already in the local registers to build the pixel window needed for the current function. Maintaining these immediately reused value saves significant energy and bandwidth over reading the entire pixel window for each iteration. This local register structure, implemented as a shift register, provides all of the immediate reuse that occurs between subsequent function invocations.
The function itself is implemented as fixed function datapath. This involves unrolling the corresponding instructions in space and allocating an arithmetic unit for each operation (eg. an adder for an addition, a multiplier for a multiplication, a mux for select, etc.). Given its functional nature it can be implemented as a combinatorial logic path and then optimally and deeply pipelined using automated re-timing techniques\cite{112,113}. This pipelining enables subsequent function invocations each cycle resulting in a throughput of a pixel per cycle. Similar to a digital signal processing pipelines, the operations and precisions in the data path can be highly tailored to the computation\cite{109,127}.

Applications, which can be conceptualized as a cascade of kernels, are constructed by similarly cascading these hardware components. The communication between kernels are arbitrated through FIFO’s allowing for back pressure to force small changes in pixel rate. Recall that this cascade of kernels is important for increasing the total amount of computation done per global memory operation.

Figure 2.3 summarizes these concepts as a single unit. An individual kernel is built by binding the attributes of kernel computation - function, window size, pixel formats, etc - to the canonical image signal processor kernel architecture. The canonical image signal processing pipeline then would be the linked list interconnection of the appropriate number of these canonical kernels.

![Diagram](image)

Figure 2.3: The canonical image signal processor captures immediate reuse in local registers, captures the full window of reuse in the line buffers, unrolls the computation in space, and uses FIFO’s to implement producer consumer relationships.

Not only are image signal processing pipelines built this way, a number of other image processing accelerators - like H.264 encoders, H.264 decoders \cite{42,45,213} - leverage this line-buffered interconnection of functional data paths. Commodity image signal processor
ASICS are often a package of commonly used pipelines and kernels packaged together to allow for the acceleration of a number of applications[198].

Traditionally, building these architectures has proved difficult. Specifically, the issue here is the gap between application experts and hardware experts, often referred to as the software-hardware co-design problem[183, 196]. As an application expert makes changes or improvements in the algorithm design, those changes requires a significant hardware iteration through the logic design engineer. Conversely, the logic designer may observe arbitrage opportunities that increase energy efficiency at some image quality cost. Without a concept of either the execution model or the algorithms structure, it is difficult for the hardware engineer to make or suggest improvements.

Recent work has attempted to address this design problem by more closely coupling image processing accelerators to highly flexible scalar cores.

2.1.3 Fixed function image co-processors reduce CPU overheads

The idea of adding specialized co-processors as special instructions is certainly not new (eg. x87 floating point extensions[148, 31, 174, 29]). Recent work has explored the extension of microprocessor execution engines through special instructions that allow for access to deeply pipelined fixed function units local too the core. Much of this work was done in the context of “Understanding sources of inefficiency in general-purpose chips”[82]. The inefficiencies identified by Hameed and Qadeer underpin the motivation for changing the computation (Sec. 1.3). This work extends those insights by identifying the patterns in computation which are likely to minimize those overheads and building a formal representation for the image processing subset.

The Tensilica microprocessor[4] and its Xtensa toolset[78] allows for the Verilog-like description[5] of instruction extensions to be synthesized into the microprocessor core. This allows for a reduction of the design effort of the logic designer and also provides for a reduction in implementation risk due to the availability of a scalar processor. Hameed and Qadeer used this framework to build instruction extensions that support H.264 encoding kernels and capitalize on a number of its extreme locality characteristics.

Unfortunately, this approach is still limited by the software co-design problem as the algorithm and hardware still need to be developed. Further, this model for computation does not explicitly capture notions like working set locality or producer-consumer locality and instead depends on the memory hierarchy and cache coherence to achieve these
2.2 ISP computation can be formally specified

Venkatesh and Taylor provide an improvement on the Hameed and Qadeer work by creating a process which not only identifies the computation that should be accelerated, but proceduralizes the generation of hardware for mature code [205]. This conservation core is used in the Greendroid processor to accelerate hot codes in mature Android operating system libraries [190]. The idea here is that die-area is cheap relative to power [71, 84, 210] and so a large number of infrequently used, easily generated generators would be a relative cost improvement. Interestingly, much of the power savings found in the mature code were for the kind of extreme locality computation that is traditionally implemented as fixed function hardware in the un-core like our image signal processor. Unfortunately, the conservation cores, by focusing on kernels rather than pipelines, is limited in its ability to optimize memory hierarchy energy without a change in the computation. Further this process was dependent on dynamic code analysis and benchmarking in order to determine which codes were important.

The approach taken in this work is to reduce design effort with proceduralization but to restrict the computation to algorithm specifications where we can use static analysis to take advantage of all aspects of an extreme locality computation, especially the memory energy.

2.2 ISP computation can be formally specified

Luckily, there are a multitude of ways to specify this subset of extreme locality computation that resembles the canonical image signal processor. In fact there are a number of domain specific languages built for image processing and image understanding that could be used as the front-end for such a system (Sec. 2.2.1). Milder’s work on Spiral [132, 130] shows how a front-end could be translated into an intermediate representation and eventually translated into hardware (Sec. 2.2.2). Spiral, though, focuses on digital signal processing, a similar and overlapping domain of computation. Qadeer and Hameed [162] propose an intermediate for the convolutional subset of image signal processing and an architecture on which it can be executed (Sec. 2.2.3). We extend their compute model by proposing a virtual machine model for stencil computation, a subset of image processing that describes all computation that can be executed on a canonical image signal processor (Sec. 2.2.4). This virtual machine model forms the basis for an intermediate representation (Sec. 3.1)
that we generate into fixed function hardware (Sec. 3.2).

2.2.1 Image processing domain specific languages specify image functions

While domain specific languages are not new (eg. SQL, OpenGL[40]), there has been a renewal of interest in DSL’s to better understand the trade-off between the performance of the program, the productivity of the programmer, and the expressiveness of the programming language[110, 32, 41]. API’s, like BLAS[107], greatly simplify the expression while limiting the semantic to well implemented routines. OpenCV[146] is a popular interface for computer vision applications. While such a framework is practically useful in constructing pipelines from known components, it makes the optimization of pipelines and the creation of novel kernels difficult or impossible (Sec. 2.2.1). Halide[163, 164] addresses both issues by focusing the expression of arbitrary image functions (Sec. 2.2.1). Because of this relaxed programming model, efficient task schedules based on the line buffered interconnection can be difficult or impossible to determine. Naturally, restricting the computation of Halide to programs that could be run on the canonical image signal processing pipeline architecture would reduce this difficulty. This restricted programming model is exactly the approach that Darkroom[87] took (Sec. 2.2.1). In fact, we use Darkroom as the front-end for our system to generate hardware.

OpenCV

OpenCV[146, 133, 218, 189, 73] contains many of the commonly recurring routines in computer vision and machine learning. Many of these routines are highly optimized implementations with back end targets including x86, CUDA[36], and OpenCL[157, 100, 216]. OpenCV has eliminated some of the variance in computer vision application performance results due to differences in underlying implementations and has also reduced the barriers and effort to build a working prototype.

OpenCV as a first mover in the space, acts as a sanity check for the inclusion of the appropriate kernels and pipelines in a new or restrictive programming model. For example, in this work a number of the benchmarks included have direct corollaries to OpenCV invocations (Sec. 2.3). As OpenCV grows to include machine learning routines, whose implementations resemble a sequence of BLAS calls, the ability for a single compute model, like the canonical image signal processor, to span the entire space becomes difficult. Future work might consider the canonical architecture for BLAS functions in the pursuit of a
similar and composable model for extreme locality computation (Ch. 5).

Fundamentally, an API abstraction is a composition model rather than a tool for new expression. While it is possible to create pipelines of kernels using this framework, it is difficult to create novel kernels using the API. This has a lot to do with the problem that any given user’s expression could not leverage the implementation tricks used to achieve high efficiency. However, the implementation being an API can be easily composed with an high level language implementation of the desired novel kernel.

The more pressing issue is that an API is based the sequential invocation of routines. This sequencing restricts the schedule to one in which a complete image result is computed before invoking the new kernel. This is significantly different than the canonical image signal processing pipeline where all kernels run simultaneously and only fractions of the intermediates exist at any given time.

Future standards, like OpenVx, that seek to expand or build on the OpenCV standard have built in components to allow for novel expression of kernels and optimal analysis of the pipeline[165]. Though, for large and complicated pipelines with arbitrary interconnection of novel kernels, determining the optimal schedule can be fairly complicated. Halide addresses this issue by partitioning the expression of the kernels, the pipelines, and the schedule that they run on.

**Halide**

Halide[163, 164] allows for the arbitrary expression of image functions. This is a function whose input is the pixel address of the output image is defined in terms of the composition of other image functions. For example, an image “in” can be blurred by simply adding and then dividing the components to produce the image “blur” (Fig. 2.4).

This matches our model for a canonical image signal processor in that we have a single function which can be invoked to generate a single output pixel. However this makes two significant departures from that computation model.

First subsequent invocations of the function can be dependent on each-other, for example the summed area table image “sat” is generated in terms of prior calculated values of “sat” significantly restricting the order of invocation and requiring that a portion of the result is always available as input. In the canonical version of the image signal processor there was no requirement on the ordering of the processing on the inputs due to a cyclical dependence.
blur(x,y) = ( in(x-1,y-1) + in(x+0,y-1) + in(x+1,y-1) \\
+ in(x-1,y+0) + in(x+0,y+0) + in(x+1,y+0) \\
+ in(x-1,y+1) + in(x+0,y+1) + in(x+1,y+1) ) / 9. ;

sat(x,y) = ( sat(x-1,y) + sat(x,y-1) + sat(x-1,y-1) + in(x,y) ) ;

sample(x,y) = ( in(s(x,y)) + in(s(x,y)) + in(s(x,y)) + in(s(x,y)) ) / 4. ;

Figure 2.4: Image function examples

However, it would be simple to extend that model to include this specific feedback case$^2$.

Second, the support in the input image is unknown at compile time. It is possible in arbitrary image functions to dynamically calculate input image addresses in an unbounded way. For example, the image “sample” is created by sampling the image “in” at the locations specified in the image “s”, which is an image whose pixels contain “x,y” coordinates. If those sample locations are unbounded then it is difficult to calculate the window size required for the function. In cases where the bound is much larger then the amount of computation, this would resemble random accesses into memory rather than dense arithmetic performed on local registers.

Both of these issues result in computation model where the schedule cannot be simply and statically determined$^{[87]}$. The innovation provided to combat this complexity is the ability to parametrically define the schedule executed for each kernel. With some intuition users can arrive at reasonable task schedules$^{[164]}$. Because the design space has been parameterized, it is also possible to use genetic programming to explore the design space for optimal task schedules$^{[163]}$.

While Halide could be restricted to only support the computation of a canonical image signal processor, this work utilizes Darkroom which already has these restriction.

Darkroom

Darkroom$^{[87]}$ adds these two restrictions missing from Halide. First, the support in the input image must be determined either by a hard bound or an affine transformation of

  We have some reservations about generally supporting cycles as they create a constraint limiting the optimization of the fixed function datapath.
output pixel addresses. Second, the cascade of kernels must be cycle free. These two restrictions are sufficient to guarantee that computation matches the canonical image signal processor pipeline such that optimal task schedules and transformations can be statically determined.

The perfect match with our needs isn’t an accident as Darkroom was co-developed with this work to be a front end for these kinds of extreme locality computation. Darkroom produces a human readable intermediate DPDA, that we describe in this work and use to generate fixed function hardware (Sec. 3.1).

We are not the first to observe that a sufficiently restricted front end that produces an intermediate representation matched to an extreme locality computation model could be used to generate high performance x86 code and energy efficient fixed function hardware. SPIRAL\cite{160, 131, 132, 130} has done exactly this for the domain of digital signal processing, a close cousin of image processing and computer understanding.

### 2.2.2 Spiral generates hardware from extreme locality DSL

SPIRAL is a domain specific language based on the algebraic expression of linear operators for digital signal processing\cite{160, 134}. SPIRAL implicitly identifies an ideal form of computation conforming to a canonical digital signal processor, which has also historically been a source of significant efficiency in SoC designs.

SPIRAL therefore requires a similar set of restrictions on the computation: functional, statically determined data dependences, finite working sets, etc. This naturally allows for the compiler to more easily determine the properties of the computation and make transformations like vectorization, loop unrolling, etc.

Milder’s work explored how this can be used to directly generate hardware\cite{131, 132, 134, 129, 130}. His work focused on the implementation of discrete Fourier transforms\cite{131} and sorting networks\cite{222}. Milder identified the formal representation for the underlying implementation of algorithms to be run on their canonical digital signal processor\cite{132}. This intermediate representation, existing between the front-end algebraic description of the algorithm and the back-end resulting hardware, bares a significant semantic resemblance to the representation we use. Though Milder utilized an algebraic description while we focused on a parameterizable assembler (Sec. 3.1.1). Additionally, Milder proposed how this formal representation could be procedurally transformed\cite{130} using a set of familiar digital signal processor transformations\cite{127}. Much of that work is relevant here as many of
these transforms work for any datapath that implements a single static assignment routine.

Most of this hardware work, though, focused on individual kernels (e.g., DFT, sorting) rather than pipelines\textsuperscript{130}. In that sense, our work, building an abstraction around the canonical image signal processing pipeline, extends the work done in Milder by examining image processing and image understanding pipelines of kernels. Naturally we could have extended Spiral directly as the baseline for this work. As we identify later (Sec. 3.1.1), there were a number of semantics missing from Milder’s data path representation including precision annotations and non-linear operations.

Fundamentally, though we needed an abstraction that includes images, which is exactly what the Qadeer and Hameed explored in their implementation of the Convolution Engine\textsuperscript{162}.

\subsection*{2.2.3 Convolution engine specifies convolutional image functions}

As a follow on to their earlier work describing the inefficiency of general purpose processors\textsuperscript{82}, Hameed and Qadeer looked at the cost flexibility by exploring an architecture restricted to a single domain of computation, convolutional operators\textsuperscript{162}. The Convolution Engine focused on the convolutional subset of image processing. The convolutional subset can be described as a restriction on our canonical image signal processing pipeline such that each pixel in window is computed by the single map function whose result is then reduced by a commutative operation (Eq. 2.1).

\begin{equation}
    y_{i,j} = F_{Reduce}(\text{for } F_{Map}(x_{(i+k,j+l)}, \beta_{k,l}) \text{ over all } k,l) \\
    | F_{Reduce}(\sum, min, max, ...) \\
    | F_{Map}(*, |p - q|, ...)
\end{equation} \tag{2.1}

The architecture on which this is based matches this model directly by creating a set of functional units whose selection is determined by the mode of operation. For example, convolution would involve selecting a multiplication for map and summation for the reduction, while a patch comparison routine would select an absolute difference for the map and a summation for the reduction. Hameed and Qadeer point out that this convolutional subset is a relatively significant portion of image signal processing\textsuperscript{162}, which we echo when justifying the pervasiveness of our canonical image signal processor pipeline (Sec. 2.3).
2.2. **ISP computation can be formally specified**

In place of the local shift registers in the canonical ISP, which holds the window provided to the function, this unit uses a shift register modified to include arbitrary access to sub-windows. In this way, their image functions are not restricted to support a fixed window size. Instead, there is an upper bound for window sizes supported by the architecture that must be virtualized when exceeding the threshold.

While there is an intermediate representation for this architecture that can be hand generated from high level code, this assembly language is actually an instruction set meant to explicitly manage the accelerator. Practically any kernel described in this format is fixed to this architecture, and generating code for that architecture (eg. from Halide or Darkroom) is difficult.

Further, the convolution engine framework doesn’t explicitly support a line buffered interconnection of image processing kernels. In order to take advantage of such an optimization where intermediate images are optimally buffered in the cache hierarchy a front-end like Halide[163, 164] or Darkroom[87] would need to provide the implementation of software line buffers. Non-convolutional image functions, supported by the canonical image signal processor, could then be supported on the vector processor co-located with the Convolution Engine.

In this work we extend this convolutional subset to more closely match the canonical image signal processing pipeline. We refer to this broader set as stencil computation.

### 2.2.4 Stencil virtual machine model specifies ISP applications

Dense stencil computation describes a set of data independent sliding window image functions of the form:

\[
y_{(i,j)} = f_w(x_{((k,k+n,l;l+m)})} \quad | \quad (k,l) = \alpha \circ (i,j) - \delta
\]

\[
\alpha, \delta \in \mathbb{Q}^2
\]

\[
(k,l), (i,j), (n,m) \in \mathbb{N}^2
\]

where:

- \( Y \) and \( X \) are 2-dimensional images
- \( y_{(i,j)} \) is the \((i,j)\) pixel in the image \( Y \).
• $f_w$ is a function on a window.

• $x(k:k+n,l:l+m)$ is the input window whose contents are determined by slice $(k : k + n, l : l + m)$. This window is used as the input to $f_w$ from $X$ to calculate $y(i,j)$.

• $(k, l)$ describes the beginning of the slice and is calculated from the element wise product (⊙) of the desired index $(i, j)$ by a scaling factor $\alpha$ plus an offset vector $\delta$.

• $(n, m)$ parameterized the extent of the slice, where the actual extent has the dimensions $(n + 1, m + 1)$.

• $\alpha$, $\delta$ are 2-element vectors of rational numbers.

• $(k, l), (i, j), (n, m)$ are 2-element vectors of natural numbers.

Importantly any individual value $y(i,j)$ can be calculated from a limited subset of the input image $x$ defined by the rectangular region, or window, $(k : k + n, l : l + m)$. The rectangular region is statically determined by a scale $\alpha$, a shift $\delta$, and an extent $(n, m, ...)$. In most cases this guarantees that subsequent output in $Y$ (eg. $y(i,j)$, $y(i+1,j)$, $y(i+2,j)$) share a majority of their inputs.

As an example of stencil function, we can write an image convolution in the form:

$$y(i,j) = \sum_{k=0,l=0}^{P,Q} \beta(k,l) \cdot x(i+k-p/2,j+l-q/2) \quad | \quad \beta \in \mathbb{R}^{P,Q}$$  \hspace{1cm} (2.3)

If you think of this convolution example in Equation 2.3 as a specific instance of Equation 2.2 then $\alpha$ would be $(1, 1)$ and $\delta$ would be a shift to center the support. Another way to think about $\delta$ is that it identifies the center or centroid of the window. The centroid can be thought of as the window relative address at which the window is graphically centered in the input image. In Figure 2.5 the centroid address $\delta$ would be equivalent to $(2, 2)$.

The function $f_w$ translates directly to the image functions described as a part of the canonical image signal processing architecture and the image functions that drive both Halide and Darkroom.

You can interpret values of $\alpha$ that are both integer and greater than 1 to indicate a down-sampling operation. For example, Figure 2.6 shows a down-sample rate of 2 in each dimension. Operationally, this downsampling would be equivalent to striding over the input.
2.2. ISP COMPUTATION CAN BE FORMALLY SPECIFIED

Figure 2.5: In this example a $5 \times 5$ window is used from the input image to calculate the output pixel using the function $f_w$. The centroid is shown in the input with a small red dot. The centroid graphically defines the support around a pixel while also relating input pixel address to output pixel addresses. In this case the centroid address would be $(2,2)$.

Figure 2.6: In this example a $5 \times 5$ window is used from the input image to calculate the output pixel using the function $f_w$ at a down sampled rate where $\alpha = (2,2)$. Graphically this is shown as skipping every other row and column in the output.

domain at a faster rate. For example, in Figure 2.6 every other column and every other row is skipped.

Values of $\alpha$ that are rational and smaller than 1 pose a problem as they result in fractional values for $k, l$ at an arbitrary $i, j$ which have no direct meaning. We interpret these values of $\alpha$ (eg. $\frac{1}{2}, \frac{1}{2}$) as an $f_w$ which generates multiple output pixels per input window (eg. $2 \times 2$). A stride of one in the input image, would then match a stride of $1/\alpha$ in the output image. This requires that the $f_w$ specification would include the function for each output (eg. 4 outputs). This can be visualized as generating multiple pixels per
CHAPTER 2. AN IMAGE PROCESSING STENCIL ABSTRACTION

Input: x

Figure 2.7: In this example a $5 \times 5$ window is used from the input image to calculate the output pixel using the function $f_w$ at an up sampled rate where $\alpha = (1/2, 1/2)$. Graphically this is shown as producing multiple pixels per window.

iteration. For example Figure 2.7 shows an up-sample rate of $2 \times 2$ with an $\alpha = (1/2, 1/2)$.

Input: x

Figure 2.8: In this example a $3 \times 3$ stencil operation $f_w$ is cascaded with a $3 \times 4$ stencil operation $g_w$ to calculate $z = h_w(x)$. The intermediate image $y$ shows the dependency through the leading pixel of the input of $g_w$ to the output of $f_w$.

Stencil operations can be cascaded, as in Figure 2.8). For example the results of $f_w$ can be used as the input of $g_w$ to calculate the result $z$: 
2.2. ISP COMPUTATION CAN BE FORMALLY SPECIFIED

\[ y(i,j) = f_w(x(k:k+n,l:l+m)) \quad | \quad (k,l) = \alpha \odot (i,j) - \delta \]
\[ z(d,e) = g_w(y(i:i+o,j:j+p)) \quad | \quad (i,j) = \beta \odot (k,l) - \gamma \]
\[ | \quad \alpha, \delta, \gamma \epsilon \mathbb{Q}^2 \]
\[ | \quad (n,m), (o,p) \epsilon \mathbb{N}^2 \] (2.4)

which can be shorthanded\(^3\) to:

\[ z = g_w(f_w(x)) = h_w(x) \] (2.5)
\[ z = h_w(x_{k:k+q,l:l+r}) \quad | \quad (k,l) = \beta \odot \alpha \odot (i,j) - \beta \odot \delta - \gamma \]
\[ | \quad (q,r) = (n,m) + (o,p) \] (2.6)

where the combined or cascaded operations is also a stencil operation.

The new window for the fused or composed functions has an extent \((q+1, r+1)\) and is equivalent to adding up the window extent parameters of each stage \((n, m) + (o, p) + 1\). This combined window is referred to as an effective stencil.\(^4\) Generally, the effective stencil \(R\) of a pipeline of kernels \(p = 0..P-1\) where the extent parameter of each stage \(p\) on any dimension \(d\) is \(M_d^p\) can be calculated as:

\[ R_d = 1 + \sum_{p=0}^{P-1} (M_d^p - 1) \] (2.7)

To reduce confusion, this thesis will refer to stencil operations which cannot be fissioned further into smaller operations as a stencil kernels\(^5\) (eg. \(f_w, g_w\)) and will refer to cascades or combinations of these stencil kernels as stencil applications (eg. \(h_w\)). In cases where comments are equally applicable to either kernels or applications this thesis will refer to stencil operations.

---

\(^3\)which follows from algebraic function composition

\(^4\)effective stencil is used to describe the window of the stencil operation \(h_w\) representing the fusion of all stencil operations in the cascade (e.g. \(f_w, g_w\))

\(^5\)the term kernel is used here to allude to the idea that this window function forms the inner loop kernel that traditional microprocessor architects would optimize given the large number of invocations
Figure 2.9: More complicated stencil applications can be constructed by building complex directed acyclic graphs of operations. In this case the kernels $f, g, h, d$ combine to calculate the application $x \rightarrow z$.

In the cases where there is fan-in and fan-out of stencil operations, as in Figure 2.9, the effective stencil can be calculated backwards from the output in breadth-first order. At any fan-out the relevant effective stencil is the larger across each dimension. Further, the graph of stencil operations cannot contain cycles as this would break the functional composition assumption.

Stencil kernels are operators that can be defined by: a function that calculates a pixel from a window of values, by the size of the window, the rate of sampling, and the centroid of the window. These stencil kernels are composed into directed acyclic graphs to build stencil applications. This stencil compute formalism matches the computation model described by the canonical image signal processing pipeline (sec. 2.1.2) and is a restriction on computation to express the subset of extreme locality high energy efficiency programs (Sec. 1.3). The next section will show, that while the compute model is restrictive, this subset of image signal processing and image understanding is broad and socially significant.

### 2.3 Stencil compute is broad and socially significant

This virtual machine model makes significant restrictions on the computation. For the restriction to be worthwhile, the abstraction must be both pervasive and important. Image signal processing, as we have already discussed, is concerned with the veracious capture of images and plays a role in all digital camera systems (Sec. 2.1.1). Computer vision focuses on the creation of computer understanding of those captured images (Sec. 2.3.1), which is required for any form of automation or recognition based on visual cues. Computational
photography utilizes the understanding of the captured images to simulate cameras otherwise not available to capture images (e.g. pan-tilt, high dynamic range, vintage) or enable esoteric sensor configurations to generate human readable images (Sec. 2.3.2), which is a rising trend in mobile photography.

Already a lot of consumers have expressed interest in the strengths of these visual computing applications. Consumer demand has driven an arms race in both sensor resolution and quality. Consumers have flocked to applications like Instagram in order to simulate the nostalgic feelings provided by some of the non-idealities of the analog cameras from prior decades. On top of this consumer demand, the drive to automate the home[128] and the automobile[200] rely heavily on computer vision. So, any improvement in both the resource efficiency or capability of consumer systems to execute stencil applications would have broad and significant social impact.

### 2.3.1 Computer vision

Beyond image capture, stencil kernels are an important recurring design pattern in computer vision[151] used in the pre-processing of image data[145, 144, 141], extracting features[85, 170, 56, 118], performing low level detection tasks[206], performing low level tracking tasks[140, 195, 119, 212], and performing low level segmentation tasks[204, 17].

In addition to the ISP, this thesis focuses on six computer vision algorithms (Harris, FAST, Canny, SLIC, Optical Flow, and Lukas-Kanade-Flow). The Harris corner detector[85] is used in computer vision[118, 143] to detect interesting locations in an image. The FAST corner detector[170] is an alternative to the Harris corner detector and is used[114, 142] to find points of interest in an image. The Canny edge detector[39] is used[123, 139] to find contours of interest in an image. SLIC[17] is a super pixel segmenter often used in image segmentation[193] to reduce the problem size of the global segmentation problem. Optical flow (eq. 2.8) is a component of image tracking or dense motion estimation[108, 33, 191] used to calculate the vectors that best correspond to matched patches in neighboring images, $x$ and $x'$.

\[
y(i,j) = \arg \min_{k,l} \sum_{m,n} \left| x(i+k+m,j+l+n) - x'(i+m,j+n) \right|
\]  

(2.8)

We chose these algorithms as they represent some of the foundational algorithms[193]
CHAPTER 2. AN IMAGE PROCESSING STENCIL ABSTRACTION

in computer vision that feature heavily in OpenCV (Sec. 2.2.1).

2.3.2 Computational photography

While image signal processing is concerned with the veracious capture of images and computer vision is concerned with the understanding of those images, computational photography is concerned with utilizing the understanding of the captured images to simulate cameras otherwise not available to capture images (e.g. pan-tilt, high dynamic range, vintage) or enable esoteric sensor configurations to generate human readable images[115, 135, 136]. While much of computational photography has its origins in early computer vision[193], stencil kernels play an important role in local operations like high dynamic range[69] and bilateral filtering[43].

This thesis will look at one computational photography applications (Richardson-Lucy Deconvolution). RL deconvolution[169] removes the blur from an input image by estimating an removing a point scatter function. We selected this algorithm as it represented a case where significant arithmetic was computed for a small amount of memory communication.

2.4 Generate hardware for stencil compute

If we have a description for stencil computation, it would be great if a hardware could be procedurally generated from that description. We have already described SPIRAL (Sec. 2.2.2) as a potential road-map for such a system. In SPIRAL, Milder converted the formal algorithmic description of the data-paths directly into Verilog. Another approach might layer high level synthesis tools between the intermediate representation for the computation and the eventual generation of a register transfer level hardware description. There has already been a significant amount of work that extends high level synthesis to allow for a domain specific language to be translated into a ASIC hardware or FPGA design (Sec. 2.4.1). Specifically in the domain of stencil computation, there has been a significant focus on leveraging polyhedral compilation techniques to generate efficient FPGA configurations for scientific computation (Sec. 2.4.2). In this work we utilize Genesis2 to implement our procedural generation of SystemVerilog for the stencil hardware from a specification to be executed on the virtual machine model (Sec. 2.4.3). This procedural generation is based on building template hardware which can generate instances from the specification of its parameters (Sec. 2.4.4).
2.4. GENERATE HARDWARE FOR STENCIL COMPUTE

2.4.1 High level synthesis of a domain specific language

George et al.\textsuperscript{[76]} shows how OptiML\textsuperscript{[188]}, a domain specific language created with DeLite\textsuperscript{[110, 32]}, can be translated into OpenCL\textsuperscript{[157, 100, 216]} and then compiled into an efficient FPGA configuration using Altera’s OpenCL\textsuperscript{[9]} compiler. George’s work focused on the execution speedup and energy savings of four kernels in the space of machine learning. While, they did not necessarily achieve the orders of magnitude energy efficiency that we are chasing, they show an important design decision in the construction of a domain specific language to hardware synthesis flow. This choice fundamentally differs from the approach taken here (Sec. 3.2) and the choice Milder made in translating SPIRAL to hardware (Sec. 2.2.2).

The distinction is that George leveraged as much of the HLS infrastructure in the implementation of the back-end system as possible. So while SPIRAL’s\textsuperscript{[160, 131, 132, 130]} hardware generation needs to re-implement\textsuperscript{[130]} many of the optimizations that already exist in these high level synthesis frameworks\textsuperscript{[127]} when generating Verilog, George’s work only needs to format their intermediate to make those optimizations possible. Further, there are many compiler optimizations that are already trivially included in an HLS framework to optimize data-paths like constant propagation, dead-code elimination, peep-hole optimization, and precision optimization\textsuperscript{[49]}.

2.4.2 Polyhedral hardware generation

One approach to optimizing stencil kernels is to write each kernel as a nested loops, as would naively be done to implement them in a general language like C, and allow the compiler to reason about the loops in order to find optimizations in the design space. Recently, this compiler work has been used to generate stencil CUDA binaries \textsuperscript{[91, 90, 101]}. This differs significantly from the perspectives provided by Halide, OptiML, Darkroom, and SPIRAL where the expression was closely tied to the intended behavior rather than the coupling of implementation and behavior.

Polyhedral compilation techniques\textsuperscript{[215, 79]}, which have their origin in compiling nested loop programs for systolic arrays, work by parameterizing the loops in a program and exploring the trade-offs between working set size, locality, and parallelism that we alluded to in Section 1.3. By unrolling and reordering loops different performance characteristics can be tuned out. It is natural to think then, that such an approach would be helpful in mapping these algorithms to FPGA.
Cong’s work\cite{50, 51, 156} explores this trade-off for scientific stencil computation. This work focused on kernels with significant floating point code and thus didn’t explore optimizations related to the producer consumer relationships between kernels.

In fact, the polyhedral parameterization does not include the required parameters to see line buffered interconnection of kernels\cite{163, 164}. This is due to the fact that parameterizations don’t travel between nested loop groups and none of the parameterizations include the ability to recompute results. Though it wouldn’t be difficult to include those parameters, as was done in the Halide intermediate representation, this would have the effect of significantly increasing the parameter space for an application.

That parameterization is the biggest issue with using a polyhedral analysis on these kinds of programs. The space of reasonable designs is actually very small, and this parameterization hides what are statically determinable and reasonable schedules\cite{87}. Though, we have only examined this for 2-dimensional images, so it may be the case that polyhedral techniques are required for hi-dimensional problems.

### 2.4.3 Genesis2 hardware generation language

The fixed function hardware generator presented in Chapter 3 is implemented using Genesis2\cite{180, 178}. Our generator contains components similar to the hardware generators used in Spiral\cite{132, 130}. It takes specifications (Sec. 3.1) that can be executed on the virtual machine model (Sec. 2.2.4) and produce fixed function hardware (Sec. 3.2).

Genesis2\cite{179} is a single pass meta-programming language where the metalanguage Perl\cite{15} is used to control the elaboration of the object language SystemVerilog\cite{7}. Genesis2 allows for structural modules to be constructed at compile time allowing for a greater re-use of child components beyond SystemVerilog’s constructors and allows for the tracking and introspection of those module parameters. It is distinct from generation frameworks like BlueSpec\cite{1, 12} or templated SystemC\cite{3} in that it allows for structural generation and distinct from frameworks like Chisel\cite{27} or MyHDL\cite{2} in that it can easily incorporate older designs and design methodologies. Most significantly, it makes a hard distinction between the language that describes the generation, Perl, and the language that implements that generation, SystemVerilog. This distinction is important for distinguishing between code that will result in hardware and code that will not result in hardware.

With respect to this work, it was important that the generation language allowed for near arbitrary proceduralization of the hardware and a deep introspection of arbitrarily
constructed parameters. The issue here is that the proceduralization needed to be complex enough to build data paths as in Spiral[132, 130] and that the parameters needed to capture all of the concepts contained in the virtual machine model (Sec. 2.2.4, Sec. 3.1).

2.4.4 Generation from hardware templates

The hardware template is procedural encapsulation of an abstract design. For example, the floating point multiply accumulate unit generator[75] we discussed earlier (Sec. 1.1) represents an abstract floating point unit and contains the procedures to generate specific designs. Template based design is not new, nor is it specific to hardware. For example, template based meta-programming[207, 103] is a significant portion of modern C++ design productivity[16]. These C++ templates are usually based on the abstraction of a type which is propagated into the interface of the object or function and the operations contained within. In this way a function whose implementation only requires standard operations to be implemented, can be described once for all possible types. With respect to hardware templates we talk about two kinds of parameterization: permutation and generation.

Most designs that we talk about are specified with permutation parameters. A permutation parameter is specified with a number or a member from a list. These permuted parameters, then, define a large but finite number of possibilities. For example, a FIFO might have a parameter defining the word-length of an entry and a parameter defining the number of possible entries in the FIFO. A third parameter might list the possibility to implement the FIFO either as a circular buffer or a bi-directional shift register. We can enumerate all of the possible designs as the cross product of these parameters, and their implementation is usually direct (eg. a for-loop’s iteration count, a signal declaration, or case-switch on possible implementations). Examples of these permutation generators include Galal’s floating point generator[75], Richardson’s FFT generator[168], Zhu’s application specific logic in memory block generator[221], Danowitz’s interface generator[58], and Tensilica’s Xtensa core[4].

Another type of parameter is generative in that it processes a data structure using a rule based system to create hardware. While a permutation describes a closed design region with an enumerable number of instances, a generative rule describes a space of designs. The rule based elaboration reduces the amount of effort in building components whose specification can be encapsulated concisely in simple human readable formats.
For example, an on-chip network could be generated[150] with parameters for the number of nodes in the network, the interconnectivity of the network, and a specific enumerated topology type. Similar to our reduction tree, this would be a permutation generator capable of expressing known designs. On the other hand, as in Papamichaeis work[150], this could be extended with a specification for describing the graph of interconnection between network nodes. As long as those interconnection specifications conform to the structure and rules of the elaboration system, novel topologies outside the enumerated list can be expressed. Further examples include Wachs’s cache coherence protocol generator[208], Gibb’s network parser generator[77], Shacham’s JTAG generator[178], and Tensilica’s instruction set extension framework[5].

A template based architecture, then, is the composition of these flexible widgets such that the interdependent parameters are determined from the template fields. Some of these interdependent parameter values are a direct result of interpreting the generative parameter data structures. In Chapter 3, we describe the input (Sec. 3.1) to our template based hardware generator, which represents programs to be run on the virtual machine model, and the architecture of the template on which the generator is based (Sec. 3.2), which is capable of generating canonical image signal processing pipeline instances (Sec. 2.1.2).
Chapter 3

Stencil Engine Generator

This chapter presents a data path description assembler (DPDA) (Sec. 3.1) to fulfill the role of an intermediate representation for those high level languages which conforms to the stencil compute model and is one of the main contributions of this thesis. DPDA represents a general virtual machine model for a stencil engine which can be used as a compilation target for high level languages. We can generate fixed function hardware directly from this intermediate representation (Sec. 3.2), which is the second contribution of this chapter. This fixed function hardware not only represents a validation of the virtual machine model but acts as a demonstration of the efficiency of stencil computation, providing a practical lower bound for the energy cost of this computation (Sec. 3.3).

3.1 Stencil computation intermediate representation

Given the virtual machine model (Sec. 2.2.4) and the formulation of a stencil application (Eq. 2.2), a stencil application can be specified in three parts. First, the functional part of the stencil kernel expressions (eg. \( f_w \) from Equation 2.2) can be defined using an assembler language (Sec. 3.1.1). Second, the properties that define the sampling rate, window size, and pixel format can be defined in terms of a set of a set of parameters (Sec. 3.1.2). Third, an application is defined in terms of how these kernels are interconnected (Sec. 3.1.3). We designed this stencil application specification such that it would naturally extend from functions on two-dimensional images to functions on arbitrarily dimensioned tensors (Sec. 3.1.4). This intermediate representation is only useful, though, if it is possible to generate that specification from higher level languages. In this work we use Darkroom[87], but there
are a number of other high level languages that have an overlapping computation model (Sec. 2.2.1). Naturally, our intermediate representation derives many of its design choices from other specification languages meant for hardware generation (Sec. 3.1.5). Though, with respect to prior work in stencil application intermediate expressions (Sec. 3.1.6), this is the first work to define a formal specification and compute model that is not specific to a computer architecture.

### 3.1.1 Data path description assembler

Data path description assembler (DPDA) is a language for defining functional expressions using assembly like operators on registers of specified precision. The goal of the language is to facilitate the construction, expression, and optimization of arithmetic data paths.

DPDA has three components. The first component are registers that hold and define intermediates, inputs, and outputs. The second component is the operations which read from registers and write to registers. The third component is the interconnection graph of registers and operations.

<table>
<thead>
<tr>
<th>Register Type</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed Point</td>
<td>fix_8_4 archer</td>
<td>a signed value with a fixed radix. In this case 8 bits of integer and 4 bits of fraction</td>
</tr>
<tr>
<td>Unsigned Fixed Point</td>
<td>ufix_3_9 lana</td>
<td>an unsigned value with a fixed radix. In this case 3 bits of integer and 9 bits of fraction</td>
</tr>
<tr>
<td>Floating Point</td>
<td>float_23_8 cyril</td>
<td>a signed floating point number. In this case 23 bits of mantissa, 8 bits of exponent, and 1 bit for sign</td>
</tr>
<tr>
<td>Boolean</td>
<td>bool duchess</td>
<td>a single bit Boolean value</td>
</tr>
</tbody>
</table>

Table 3.1: The four base types in DPDA are enumerated along with a short description of how precision is specified.

Registers represent all of the intermediate values, inputs, and outputs of the function. Table 3.1 lists the possible intermediate base types. Each type is also provided additional fields to define the precision in terms of allocated bits. This is to allow the reduced precision expressions that were suggested in Section 1.3.

In addition to base type, more complex types can be built by adding dimensionality
3.1. **STENCIL COMPUTATION INTERMEDIATE REPRESENTATION**

<table>
<thead>
<tr>
<th>Array Type</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalar</td>
<td>fix_8_4 malory</td>
<td>a single value that is unindexable</td>
</tr>
<tr>
<td>Vector</td>
<td>ufix_3_9 cheryl[3:0]</td>
<td>an array of 4 elements</td>
</tr>
<tr>
<td>Matrix</td>
<td>float_23_8 barry[2:0][3:0]</td>
<td>a $3 \times 4$ array of arrays</td>
</tr>
<tr>
<td>Tensor</td>
<td>float_23_8 pam[2:0][3:0][4:0]</td>
<td>a $3 \times 4 \times 5$ array of arrays of arrays</td>
</tr>
<tr>
<td>Tensor</td>
<td>bool ray[1:0][1:0][0:0]</td>
<td>a $2 \times 2 \times 1$ array of arrays of arrays</td>
</tr>
</tbody>
</table>

Table 3.2: The base register types can be modified with dimensionality tokens which allow for a register definition to be extended to a vector, matrix, or tensor tokens to the register definition. This allows for the concise expressions of operations that occur across many elements and for the suggestion of commutative reductions. Table 3.2 enumerates some possibilities.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Example</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>add z a b</td>
<td>$z = a + b$</td>
</tr>
<tr>
<td>mul</td>
<td>mul z a b</td>
<td>$z = a \cdot b$</td>
</tr>
<tr>
<td>mul</td>
<td>mul Z A B</td>
<td>$Z = A[N:0] \cdot B[N:0]$</td>
</tr>
<tr>
<td>mux</td>
<td>mux z s a b</td>
<td>$z = s?a : b$</td>
</tr>
<tr>
<td>sum</td>
<td>sum z A</td>
<td>$z = \sum A$</td>
</tr>
<tr>
<td>min</td>
<td>min z A</td>
<td>$z = \min A$</td>
</tr>
</tbody>
</table>

Table 3.3: DPDA operators mimic the format of other assembler languages placing left hand side registers in the left most fields while placing right hand side registers in the right most fields. Operations like add, mul and mux operate on scalars or element wise on vectors. For example, in an element wise execution of mux, the $i$ index in $S$ determines whether the $i$ index of $A$ or $B$ is placed at the $i$ index of $Z$. Reduction operators like sum and min require a dimensional right hand side to generate a scalar left hand side.

Registers are read and written by operators. All operators lack state and the output is solely a function of the inputs. The precision of these operators can be determined from the precision of the input and output registers. Operators can be divided into two categories: scalar operators which operate on scalar values to produce a scalar value (eg. add, sub, mul, div) and reduce operators which read dimensional registers to produce a scalar register (eg. sum, min, max, mux, mean, median). An important operation is “mux”, which allows for the implementation of branch like behavior through predication.

Operations and registers are organized into a directed acyclic graph (DAG) with the
additional requirement that a register is write once\(^1\). Naturally, registers can be read many times. The property that all DPDA programs are a write once DAG of operators\(^2\) guarantees that the entire expression is itself a function.

The implication is that a DPDA program can be implemented in combinatorial logic and trivially re-timed[113] and can be implemented as an instruction sequence that can be statically optimized[98, 106, 105]. Naturally this formulation also benefits from many of the other observations about functional programming[55, 94, 93, 67](eg. facilitated verification, optimization, and automation).

The inputs, outputs, and constant programmable coefficients are defined in the kernel parameters “input”, “output”, and “coeff” respectively. These interfaces are specified using register definitions. The input pixel format is combined with the “Window” parameter to build the full input. For example, “pix_in[2:0]” is combined with “[9,9]” to specify that the full input window register as “pix_in[2:0][8:0][8:0]” (Fig. 3.1). In this case the inner dimension of length represents the channels per pixel, the middle dimension of length 9 represents the horizontal or dominant traversal dimension, and the outer dimension of length 9 represents the vertical or secondary traversal dimension. Generally, our system uses the raster based coordinate positioning where the origin is in the upper left of the image. Similarly any parameter in this system will be organized from most dominant dimension to least dominant dimension (eg. “x” then “y”).

The constant programmable coefficients “coeff” are a set of registers that can be reconfigured with low frequency. For example, a slightly configurable convolution unit (Sec. 3.1) would allow for the filter coefficients to be defined once at the beginning of execution and then held constant during the full invocation. In OpenGL these are often referred to as Uniforms because they are uniform over the execution[40], while in digital signal processing they would be referred to as programmable taps.

As it is occasionally necessary to know the current address of the centroid there is a special register “centroid_pos” that contains the current centroid address as an integer. This is useful, for example, when the position in the image determines the contents of the pixels as in a mosaic image.

As pixels can sometimes be complicated in format - the input, output, and coefficients

\(^1\)a scalar register may only have one incoming directed edge, while a register with dimensionality may only have an incoming directed edge per each element in the tensor.
\(^2\)also called single static assignment (SSA)
can each be specified as a list of DPDA registers. Fundamentally, this allows for struct\textsuperscript{3} like definitions of pixels. Allowing for complex values for inputs and outputs achieves two goals. First it allows for a natural way to express stencil kernels that share inputs and intermediate values. Second it allows for the expression of algorithms which operate on and produce complex values. For example, many color channel formats allocate more bits to the color channels which carry more information. Figure 3.3, shows an example of how this struct format could be used to allocate a different number of bits for red, green, blue.

A \(9 \times 9\) convolution of 3 channel pixels is shown in Figure 3.1. The scalar operation multiply is distributed element wise for the three tensors such that the \(i, j, k\) term of \text{pixel\_in} and \text{coeff} is used to calculate the \(i, j, k\) term of \text{partial\_res}. The summation reduction is broken into three components one for each of the channels\textsuperscript{4}.

```plaintext
conv:
    Name: convolution
    Window: [9,9] # Window size
    StrideIn: [1,1] # Down sample rate
    StrideOut: [1,1] # Up sample rate
    Centroid: [4,4] # Window offset
    PixelIn: ufix_8_0 pix_in[2:0] # Pixel Format
    PixelOut: ufix_12_0 pix_out[2:0] # Pixel Format
    Coeff: ufix_0_8 coeff[2:0][8:0][8:0] # Prog. Const.
    Boundary: ZeroFill
    Function: conv_fun

conv_fun: #9x9 convolution on 3 channel pixels
    ufix_9_8 partial_res[2:0][8:0][8:0] # Partial results

mul partial_res pix_in  coeff # 9x9x3 multiply
sum pix_out[0] partial_res[0][8:0][8:0] # 9x9 reduction
sum pix_out[1] partial_res[1][8:0][8:0] # 9x9 reduction
sum pix_out[2] partial_res[2][8:0][8:0] # 9x9 reduction
```

Figure 3.1: Data path description assembler \(9 \times 9\) convolution of a 3 channel image. The entry conv_func defines the function while the entry conv lists the kernel parameters and points to the window function conv_fun.

Additionally, a \(7 \times 7\) sum of accumulated differences (SAD) program (Fig. 3.2) would replace the multiply with a subtract and absolute operators. This time the summation is

\textsuperscript{3}as in C struct

\textsuperscript{4}Note that supplying a tensor type in place of a scalar type in a reduction operation implies a partial reduction along the right most unspecified dimensions. Figure 3.1 could take advantage of this as the reduction is across pixels in the left most dimensions which are the channels.
represented as one instruction to take advantage of the syntax where supplying a tensor
type in place of a scalar type in a reduction operation implies a partial reduction along the
right most unspecified dimensions.

sad:
Name: sumOfAbsoluteDifferences
Window: [7,7] # Window size
StrideIn: [1,1] # Down sample rate
StrideOut: [1,1] # Up sample rate
Centroid: [3,3] # Window offset
PixelIn: fix_9_0 pix_in[2:0] # Pixel Format
PixelOut: fix_12_0 pix_out[2:0] # Pixel Format
Coeff: fix_9_0 coeff[2:0][6:0][6:0] # Prog. Const.
Boundary: ZeroFill
Function: sad_fun
sad_fun: #7x7 sad on 3 channel pixels
fix_9_0 partial_res[2:0][6:0][6:0] # Partial results
fix_9_0 partial_res2[2:0][6:0][6:0] # Partial results

sub partial_res pix_in coeff # 7x7x3 subtract
abs partial_res2 partial_res # 7x7x3 absolute value
sum pix_out partial_res # 3 sets of 7x7 to 1 sum

Figure 3.2: Data path description assembler 7 × 7 sum of absolute difference of a 3 channel
image.

Further, a bilinear demosaic using a 3 × 3 window (Fig. 3.3) makes use of a special
register to determine its position in the Bayer mosaic. The mux operation allows for the
predication\(^5\) of the function as the computation is pixel address dependent.

Data path description supports a number of operations. The full list of operations
along with their syntax can be found online[35]. We provide a list of the operations in
Tables 3.5, 3.6, 3.7, 3.8, 3.9, 3.10. They are specified in terms of the attributes of their
registers and the function they execute. Table 3.4 provides a legend for that specification.
The register specification for the register interface is defined in terms of a dimensionality
component and a type component. Scalar indicates that a given register may be scalar or
may be a tensor as long as all other scalar argument match in dimension. For example,
the earlier element wise binary operators like add would have a scalar specification for each
entry. Tensor indicates that a given registers dimensionality is independent of the other
registers and greater than all scalar registers. Operations with a scalar left hand side and

\(^5\)_mux4 is a 4 to 1 select using a one hot selection
3.1. STENCIL COMPUTATION INTERMEDIATE REPRESENTATION

demo:
Name: demosaic
Window: \([3,3]\) # Window size
StrideIn: \([1,1]\) # Down sample rate
StrideOut: \([1,1]\) # Up sample rate
Centroid: \([1,1]\) # Window offset
PixelIn: \(\text{ufix}_8_2\ \text{pix}_\text{in}\) # Pixel Format
PixelOut: # Struct Pixel Format
  - \(\text{ufix}_6_0\) red
  - \(\text{ufix}_8_1\) green
  - \(\text{ufix}_6_0\) blue
Coeff: bool \(p[3:0]\) # Pixel Phase from address
Boundary: ZeroFill
Function: demo_fun
demo_fun: #demosaic
\(\text{ufix}_8_2\) NSEW_R[3:0]
\(\text{ufix}_8_2\) CORN_R[3:0]
\(\text{ufix}_8_2\) pass_R
mv NSEW_R[0] pix_in[1][0]
mv NSEW_R[1] pix_in[1][2]
mv NSEW_R[2] pix_in[0][1]
mv NSEW_R[3] pix_in[2][1]
mv CORN_R[0] pix_in[2][2]
mv CORN_R[1] pix_in[0][0]
mv CORN_R[2] pix_in[0][2]
mv CORN_R[3] pix_in[2][0]
mv pass_R pix_in[1][1]
\(\text{ufix}_12_2\) x_sum[3:0]
sum x_sum[0] NSEW_R
sum x_sum[1] NSEW_R[1:0]
sum x_sum[3] CORN_R
\(\text{ufix}_11_2\) NSEW_mean[3:0]
rshift x_mean[0] x_sum[0] 0x02
rshift x_mean[2:1] x_sum[2:1] 0x01
rshift x_mean[3] x_sum[3] 0x02

Figure 3.3: Data path description assembler bilinear demosaic example. Note that we have simplified this example, by providing the phase as a pixel address dependent coefficient. The full implementation would calculate \(p\) from \(\text{cent}roid\_pos\) a special register containing the current centroid address.

a tensor right hand side represent a reduction of that tensor type. Supplying a tensor type
in place of a scalar type in a reduction operation implies a partial reduction along the right most unspecified dimensions to the match the dimensionality of the destination. A tuple register is a vector whose length matches the dimensions of the tensor register. The types of registers listed in the arguments indicate which matching types are allowed. Sometimes a specific register must be a specific type and so its base type will not match the other registers, for example a comparison operation will always result in a boolean left hand side register regardless of the types of the compared. If the specific register may hold any type without matching the other registers it will be marked with an X. For example, a move operation can perform register type conversions and so its left hand side and right hand side do not match.

<table>
<thead>
<tr>
<th>Dimensionality:</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalar</td>
<td>s</td>
</tr>
<tr>
<td>Generalized Tensor</td>
<td>t</td>
</tr>
<tr>
<td>Tuple</td>
<td>p</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type:</th>
<th>Symbol</th>
<th>Forced</th>
</tr>
</thead>
<tbody>
<tr>
<td>fix</td>
<td>f</td>
<td>F</td>
</tr>
<tr>
<td>ufix</td>
<td>u</td>
<td>U</td>
</tr>
<tr>
<td>float</td>
<td>r</td>
<td>R</td>
</tr>
<tr>
<td>bool</td>
<td>b</td>
<td>B</td>
</tr>
<tr>
<td>any</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.4: The operation specification includes the kinds of registers they operate on. This includes a dimensionality component and a type component. For example, a binary scalar operation (eg. add) would have the register specification \{sufr, sufir, sufr\} to indicate that the dimensionality and type of all registers must match and that the only legal types are ufix, fix, and float.

Arithmetic operations (Tbl 3.5) are all of the form that they have matching register type, matching register dimensions, and a single left hand side register. They may be used to express singular operations on a set of scalar registers or to express element-wise operations on a set of tensors with matching dimensions. Some elements are restricted in the types they operate on as operations on that type have no implementation. For example, the absolute value of a ufix will always be the same value so the operation is not allowed.

Comparison operations (Tbl. 3.6) are all of the form that they have matching right hand side register types, matching register dimensions, and single boolean left hand side register type. They may be used to express singular operations on a set of scalar registers or to
### 3.1. STENCIL COMPUTATION INTERMEDIATE REPRESENTATION

<table>
<thead>
<tr>
<th>Op</th>
<th>Registers</th>
<th>Expression</th>
<th>Register format</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mv</code></td>
<td><code>z a</code></td>
<td><code>z = a</code></td>
<td><code>sX , sX</code></td>
</tr>
<tr>
<td><code>add</code></td>
<td><code>z a b</code></td>
<td><code>z = a + b</code></td>
<td><code>sufr , sufr , sufr</code></td>
</tr>
<tr>
<td><code>sub</code></td>
<td><code>z a b</code></td>
<td><code>z = a - b</code></td>
<td><code>sufr , sufr , sufr</code></td>
</tr>
<tr>
<td><code>mult</code></td>
<td><code>z a b</code></td>
<td><code>z = a * b</code></td>
<td><code>sufr , sufr , sufr</code></td>
</tr>
<tr>
<td><code>mac</code></td>
<td><code>z a b c</code></td>
<td><code>z = a * b + c</code></td>
<td><code>sufr , sufr , sufr , sufr</code></td>
</tr>
<tr>
<td><code>mod</code></td>
<td><code>z a b</code></td>
<td><code>z = a%b</code></td>
<td><code>sufr , sufr , suf</code></td>
</tr>
<tr>
<td><code>div</code></td>
<td><code>z a b</code></td>
<td><code>z = a/b</code></td>
<td><code>sufr , sufr , sufr</code></td>
</tr>
<tr>
<td><code>abs</code></td>
<td><code>z a</code></td>
<td>`z =</td>
<td>a</td>
</tr>
<tr>
<td><code>neg</code></td>
<td><code>z a</code></td>
<td><code>z = -a</code></td>
<td><code>sfr , sfr</code></td>
</tr>
<tr>
<td><code>exp2</code></td>
<td><code>z a</code></td>
<td><code>z = 2^a</code></td>
<td><code>sufr , sufr</code></td>
</tr>
<tr>
<td><code>exp</code></td>
<td><code>z a</code></td>
<td><code>z = e^a</code></td>
<td><code>sufr , sufr</code></td>
</tr>
<tr>
<td><code>and</code></td>
<td><code>z a b</code></td>
<td><code>z = a&amp;b</code></td>
<td><code>sufb , sufb , sufb</code></td>
</tr>
<tr>
<td><code>or</code></td>
<td><code>z a b</code></td>
<td>`z = a</td>
<td>b`</td>
</tr>
<tr>
<td><code>nand</code></td>
<td><code>z a b</code></td>
<td><code>z = (a&amp;b)</code></td>
<td><code>sufb , sufb , sufb</code></td>
</tr>
<tr>
<td><code>nor</code></td>
<td><code>z a b</code></td>
<td>`z = (a</td>
<td>b)`</td>
</tr>
<tr>
<td><code>xor</code></td>
<td><code>z a b</code></td>
<td><code>z = a \&amp; b</code></td>
<td><code>sufb , sufb , sufb</code></td>
</tr>
<tr>
<td><code>xnor</code></td>
<td><code>z a b</code></td>
<td><code>z = (a \&amp; b)</code></td>
<td><code>sufb , sufb , sufb</code></td>
</tr>
<tr>
<td><code>not</code></td>
<td><code>z a</code></td>
<td><code>z = a</code></td>
<td><code>sufb , sufb</code></td>
</tr>
<tr>
<td><code>buf</code></td>
<td><code>z a</code></td>
<td><code>z = a</code></td>
<td><code>sufb , sufb</code></td>
</tr>
</tbody>
</table>

Table 3.5: Data path description assembler arithmetic operation specification

Express element-wise operations on a set of tensors with matching dimensions.

<table>
<thead>
<tr>
<th>Op</th>
<th>Registers</th>
<th>Expression</th>
<th>Register format</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>gt</code></td>
<td><code>z a b</code></td>
<td><code>z = a &gt; b</code></td>
<td><code>sB , sufr , sufr</code></td>
</tr>
<tr>
<td><code>eq</code></td>
<td><code>z a b</code></td>
<td><code>z = a == b</code></td>
<td><code>sB , sufr , sufr</code></td>
</tr>
<tr>
<td><code>ne</code></td>
<td><code>z a b</code></td>
<td><code>z = a! = b</code></td>
<td><code>sB , sufr , sufr</code></td>
</tr>
<tr>
<td><code>lt</code></td>
<td><code>z a b</code></td>
<td><code>z = a &lt; b</code></td>
<td><code>sB , sufr , sufr</code></td>
</tr>
<tr>
<td><code>lte</code></td>
<td><code>z a b</code></td>
<td><code>z = a &lt;= b</code></td>
<td><code>sB , sufr , sufr</code></td>
</tr>
<tr>
<td><code>gte</code></td>
<td><code>z a b</code></td>
<td><code>z = a &gt;= b</code></td>
<td><code>sB , sufr , sufr</code></td>
</tr>
<tr>
<td><code>eqz</code></td>
<td><code>z a b</code></td>
<td><code>z = (a + b) == 0</code></td>
<td><code>sB , sufr , sufr</code></td>
</tr>
<tr>
<td><code>nez</code></td>
<td><code>z a b</code></td>
<td><code>z = (a + b)! = 0</code></td>
<td><code>sB , sufr , sufr</code></td>
</tr>
<tr>
<td><code>gtz</code></td>
<td><code>z a b</code></td>
<td><code>z = (a + b) &gt; 0</code></td>
<td><code>sB , sufr , sufr</code></td>
</tr>
<tr>
<td><code>ltz</code></td>
<td><code>z a b</code></td>
<td><code>z = (a + b) &lt; 0</code></td>
<td><code>sB , sfr , sfr</code></td>
</tr>
<tr>
<td><code>gtez</code></td>
<td><code>z a b</code></td>
<td><code>z = (a + b) &gt;= 0</code></td>
<td><code>sB , sfr , sfr</code></td>
</tr>
<tr>
<td><code>ltez</code></td>
<td><code>z a b</code></td>
<td><code>z = (a + b) &lt;= 0</code></td>
<td><code>sB , sfr , sfr</code></td>
</tr>
</tbody>
</table>

Table 3.6: Data path description assembler comparison operation specification

Predication operations (Tbl. 3.7) are all of the form that they have a single left hand side
scalar register whose type matches half of the right hand side registers. The other half of the registers must be boolean and represent the predication mask. The mux2, mux3, muxN operations are one hot select operations where one and only one of the boolean signals may be asserted. When select operations are used with tensors all of the fields dimensionality must match. For example, the c1 register of mux would represent a predication mask.

<table>
<thead>
<tr>
<th>Op</th>
<th>Registers</th>
<th>Expression</th>
<th>Register format</th>
</tr>
</thead>
<tbody>
<tr>
<td>mux</td>
<td>z c1 a b</td>
<td>$z = c1?a : b$</td>
<td>sufrB,sB,sufrB,sufrB</td>
</tr>
<tr>
<td>mux2</td>
<td>z c1 a c2 b</td>
<td>$z = c1?a : (c2?b : ...)$</td>
<td>sufrB,sB,sufrB,sB,sufrB</td>
</tr>
<tr>
<td>mux3</td>
<td>z c1 a c2 b c3 c</td>
<td>$z = c1?a : (c2?b : (c3?c : ...))$</td>
<td>sufrB,sB,sufrB,sB,sufrB,sB,sufrB</td>
</tr>
</tbody>
</table>

Table 3.7: Data path description assembler predication operation specification

Reduction operations (Tbl. 3.8) are all of the form that they have a single tensor right hand side register, resulting in a matching scalar type or an unsigned tuple type. When a tensor is provided in place of a scalar it implies a reduction across the right most elements of A. However this requires that the left most dimensions of A match the dimensions of the tensor provided for z.

<table>
<thead>
<tr>
<th>Op</th>
<th>Registers</th>
<th>Expression</th>
<th>Register format</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum</td>
<td>z A</td>
<td>$\sum A$</td>
<td>sufr, tufr</td>
</tr>
<tr>
<td>min</td>
<td>z A</td>
<td>$\text{min}A$</td>
<td>sufr, tufr</td>
</tr>
<tr>
<td>max</td>
<td>z A</td>
<td>$\text{max}A$</td>
<td>sufr, tufr</td>
</tr>
<tr>
<td>argmin</td>
<td>z A</td>
<td>$\text{argmin}A$</td>
<td>pU, tufr</td>
</tr>
<tr>
<td>argmax</td>
<td>z A</td>
<td>$\text{argmax}A$</td>
<td>pU, tufr</td>
</tr>
<tr>
<td>argvalmin</td>
<td>z y A</td>
<td>$\text{min} \text{, argmin}$</td>
<td>sufr, pU, tufr</td>
</tr>
<tr>
<td>argvalmax</td>
<td>z y A</td>
<td>$\text{max} \text{, argmax}$</td>
<td>sufr, pU, tufr</td>
</tr>
</tbody>
</table>

Table 3.8: Data path description assembler reduction operation specification

Bit string operations (Tbl. 3.9) act on fix and ufix types where the dimensionality of all registers must match. The register corresponding to the shift amount or population count is not required to match the type register types of the other fields. Note that while operations like rshit and lshift have a corresponding C operation, operations like rotate and popcount are usually implemented in C by invoking their assembly opcode directly through ASM.

Hardware operations (Tbl. 3.10) vary in form but are thematically related in that they are tied to the ability to cheaply implement them in a specific architecture. Operations like csa3 are usually only implemented in hardware in the course of optimizing or implementing
3.1. STENCIL COMPUTATION INTERMEDIATE REPRESENTATION

<table>
<thead>
<tr>
<th>Op</th>
<th>Registers</th>
<th>Expression</th>
<th>Register format</th>
</tr>
</thead>
<tbody>
<tr>
<td>rshift</td>
<td>w a b</td>
<td>( w = a &gt;&gt; b )</td>
<td>suf, suf, sUF</td>
</tr>
<tr>
<td>lshift</td>
<td>w a b</td>
<td>( w = a &lt;&lt; b )</td>
<td>suf, suf, sUF</td>
</tr>
<tr>
<td>rotate</td>
<td>w a b</td>
<td>right rotate bits</td>
<td>suf, suf, sUF</td>
</tr>
<tr>
<td>lrotate</td>
<td>w a b</td>
<td>left rotate bits</td>
<td>suf, suf, sUF</td>
</tr>
<tr>
<td>popcount</td>
<td>w a</td>
<td>population count</td>
<td>sU, suf</td>
</tr>
</tbody>
</table>

Table 3.9: Data path description assembler bit string operation specification

Table 3.10: Data path description assembler hardware operation specification

As mentioned, the precision of each operation is determined by its input and output register precision. For, floating point numbers this equivalent to performing the floating point computation in the largest precision of its interface registers and then using IEEE floating point rounding to reduce the result for the output precision if required. By default, fixed point and unsigned fixed point numbers computation is such that the only error in the result is from the truncation to the output register format. Parenthetical modifiers allow for a specific operator to be rounded in the desired way. For example, “add z a b” could be altered to “add (sat, rnd) z a b” to force add to saturate on an overflow and round on an underflow rather than truncate in either case.

Beyond the window function a number of parameters are required to complete the specification for a stencil kernel described in Equation 2.2.

3.1.2 Stencil kernel parameter

In addition to defining the computation required for each output pixel, DPDA needs to specify which pixels to compute and how to name them. This information is provided by a set of required parameters. There are six such parameters: the pixel formats described earlier, the extent of the window, the pixel centroid, the pixel stride, the pixel emit rate, and the boundary conditions.
The extent of the window is defined as a tuple of length two for two-dimensional images. This defines the size of the window provided as an input to the image function. This is combined with the description of the input registers to generate the input matrices. For example, in Figure 3.1 the input register “ufix_8.0 pix_in[2:0]” is extended with the window definite to create the full input to the function “ufix_8.0 pix_in[2:0][8:0][8:0]”. In the case where an input pixel definition includes multiple registers to build a struct, each register is extended in this way. This parameter corresponds to \((n+1, m+1)\) described in Equation 2.2.

The centroid parameter is the window relative address at which the window is graphically centered in the input image. This determines the relationship between the slice indices for the input window to the indices of the output pixel. This is defined as a tuple of length two for two-dimensional images. This parameter corresponds to \(\delta\) described in Equation 2.2.

Down sample kernels are defined in terms of the number of pixels strided in the input space (Fig 2.6). This is defined as a tuple of length two for two-dimensional images. This corresponds to the numerator of \(\alpha\) described in Equation 2.2. For example, to down-sample by two in each dimension every other row and every other column would be strided over.

Up sample is defined in terms of the number of pixels produced at the output (Fig 2.7). This is defined as a tuple of length two for two-dimensional images. The corresponds to the denominator of \(\alpha\) described in Equation 2.2. For example, to up-sample by two in each dimension, four pixels would need to be produced for every stride in the input space. This output stride is combined with the output pixel definition to describe the output pixel window in the same way that the input register definitions are combined with the window parameter. For an output strider greater than a single pixel, a function must specify how each portion of the output window is calculated. This parameterization of up-sample and down-sample is a slight departure from Equation 2.2. This parameterization makes the process of up-sampling more explicit as a process whose function generates multiple pixels per invocation.

Note that for each of these tuple parameters, the dominant dimension is always the left most value in the tuple. For a two dimensional image this would mean that the left most value is the horizontal or \(x\) index and that the right most value is the vertical or \(y\) index.

Finally the boundary conditions defined by selecting among possible keywords listed in Table 3.11 and combining them with any keyword parameter if they exist. The boundary conditions determines how pixel addresses which are outside of the input image are interpreted. For example, it is fairly common in algorithms which find the maxima in a
window to set the constant fill to the minimum representable value using a constant fill. The purpose of the boundary condition code is to remove special case code or hardware that would only be executed at the boundaries. Instead the image can be padded appropriately by a pre-process as in Darkroom or incorporated into hardware that is already concerned with testing boundary conditions. While there is not a system to define arbitrary boundary conditions, it is possible to pad an input image prior to entering the processing pipeline using a custom routine and then using the “None” option to achieve the desired effect. Note that when selecting the “None” case the output image will be smaller than the input image for any window size greater than a single pixel. When selecting “None”, some care has to be taken that the input and output image sizes match up with respect to the window size. It is also possible to select the “Zero Fill” option and then incorporate the special case boundary code into the image function.

<table>
<thead>
<tr>
<th>Boundary Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero Fill</td>
<td>Outside pixels are zero valued</td>
</tr>
<tr>
<td>Constant Fill</td>
<td>Outside pixels are some specified constant value</td>
</tr>
<tr>
<td>Hold</td>
<td>Outside pixels are the same as the nearest pixel in image</td>
</tr>
<tr>
<td>Mirror</td>
<td>Outside pixels are reflection across the nearest boundaries</td>
</tr>
<tr>
<td>Wrap</td>
<td>Outside pixels wrap to the opposite side of image</td>
</tr>
<tr>
<td>None</td>
<td>Output pixels dependent on out of image pixels are suppressed</td>
</tr>
</tbody>
</table>

Table 3.11: Common boundary conditions for stencil kernels.

The parameters are summarized in Table 3.12. An example of specific parameter configuration is provided in Figures 3.1, 3.2.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Window Size</td>
<td>The total size and shape of input pixels for kernel</td>
</tr>
<tr>
<td>Input Pixel</td>
<td>The DPDA register format of pixels in the input space</td>
</tr>
<tr>
<td>Output Pixel</td>
<td>The DPDA register format of pixels in the output space</td>
</tr>
<tr>
<td>Centroid</td>
<td>The stencil relative address of the “center” of the window</td>
</tr>
<tr>
<td>Coefficients</td>
<td>The DPDA register format configurable constant coefficients</td>
</tr>
<tr>
<td>Input Stride</td>
<td>The number of rows, columns skipped iterated for every output</td>
</tr>
<tr>
<td>Output Stride</td>
<td>The number of pixels produces for every stride of the input</td>
</tr>
<tr>
<td>Boundary Condition</td>
<td>How out of image pixels are interpreted</td>
</tr>
<tr>
<td>DPDA Function</td>
<td>The DPDA function used by this stencil kernel</td>
</tr>
</tbody>
</table>

Table 3.12: A stencil kernel is defined in terms of the above parameters including the DPDA function referenced.
3.1.3 Stencil application topology

A stencil application builds on top of the stencil kernel definition by linking them together. This consists of a call graph of the stencil kernels defined in the specification. This graph must be a write once DAG structure (Fig. 2.9) to guarantee that an interconnection of stencil kernels is still a stencil operator. This follows from the same arguments made regarding a write once DAG for DPDA graphs (Sec. 3.1.1).

Conceptually, this is just a set of image function calls. For example, Figure 3.4 presents the topology definition for the simple camera pipeline described in Figure 2.2. A full configuration would be broken into three kernel definitions, one each for the black level correction, demosaic, and color correction matrix. The topology description of the specification described the data dependence between the function calls.

```
top:
  input_image:
    - pixel_in
  output_image:
    - pixel_out
blc: #Black Level Correct
  PixelIn: pixel_in
  PixelOut: blc_to_dem
  Config: blc

dem: #Demosaic
  PixelIn: blc_to_dem
  PixelOut:
    - dem_to_ccm_R
    - dem_to_ccm_G
    - dem_to_ccm_B
  Config: dem
ccm: #Color Correction Matrix
  PixelIn:
    - dem_to_ccm_R
    - dem_to_ccm_G
    - dem_to_ccm_B
  PixelOut: pixel_out
  Config: ccm
```

Figure 3.4: Data path description topology definition for simple camera pipeline. The “Config” key points to kernel configurations to be invoked. For example “dem” invokes the kernel in Figure 3.3.
3.1.4 Specifying higher dimensional stencil applications

This specification language is extensible to higher dimensional problems by simply increasing the dimensionality of the registers and the length of the tuples used in the stencil kernel parameters. Recalling Equation 2.2 any two element tuple could be replaced with an \( N \) element vector to transition the formulation from a function on two dimensional images to a function on \( N \) dimensional tensors (Eq. 3.1).

\[
y_i = f_w \left( x_{\vec{k},\vec{n}} \right) \quad | \quad \vec{k} = \alpha \odot \vec{k} - \delta \\
\quad | \quad \alpha, \delta \in \mathbb{Q}^N \\
\quad | \quad \vec{k}, \vec{n} \in \mathbb{N}^N
\] (3.1)

For example, we can build a one dimensional convolution (Fig. 3.6) expression by reducing the length of the tuples and reducing the dimensionality of the operations relative to the two dimensional convolution (Fig. 3.1)

```plaintext
conv:
  Name: convolution
  Window: [9] # Window size
  StrideIn: [1] # Down sample rate
  StrideOut: [1] # Up sample rate
  Centroid: [4] # Window offset
  PixelIn: ufix_8_0 pix_in # Pixel Format
  PixelOut: ufix_12_0 pix_out # Pixel Format
  Coeff: ufix_0_8 coeff[8:0] # Prog. Const.
  Boundary: ZeroFill
  Function: conv_fun

conv_fun: # 9 wide convolution
  ufix_9_8 partial_res[8:0] # Partial results
  mul partial_res pix_in coef # 9 multiply
  sum pix_out partial_res # 9:1 reduction
```

Figure 3.5: Data path description assembler 9 wide convolution on a one dimensional stream. The entry conv_func defines the function while the entry conv lists the kernel parameters and points to the window function conv_fun. Relative to Figure 3.1, the three channels per element have been dropped and the dimensionality has been reduced by one.

Conversely, two dimensional convolution can be extended to a large number of dimensions by increasing the length of the tuples. For example, we can express a five dimensional
convolution as might be seen when converting a bilateral filter into a bilateral grid[43] by increasing the number of dimensions.

```plaintext
conv:
Name: convolution
Window: [5,5,5,5,5] # Window size
StrideIn: [1,1,1,1,1] # Down sample rate
StrideOut: [1,1,1,1,1] # Up sample rate
Centroid: [2,2,2,2,2] # Window offset
PixelIn: ufix_8_0 pix_in # Pixel Format
PixelOut: ufix_12_0 pix_out # Pixel Format
Coeff: ufix_0_8 coeff[4:0][4:0][4:0][4:0][4:0]
Boundary: ZeroFill
Function: conv_fun
conv_fun: # 5 wide convolution
ufix_9_8 partial_res[4:0][4:0][4:0][4:0][4:0]
  mul partial_res pix_in     coeff # 5x5x5x5x5 multiply
  sum pix_out    partial_res # 3125:1 reduction
```

Figure 3.6: Data path description assembler of a five-dimensional convolution. Relative to Figure 3.1, the three channels per element have been dropped and the dimensionality has been increased from two to five.

Like the two-dimensional case, the dominant traversal dimension is assumed to be the left most dimension listed. For the naive loop based implementation, the left most indice would correspond to the inner most loop variable. However, unlike the two-dimensional case, it is likely that the dominance of each dimension’s traversal may need to be re-ordered in order to achieve higher energy efficiency. As this abstraction provides no capacity to designate the order of invocations, it is possible to interpret such a behavioral specification using any loop iteration order.

When the dimensionality of a stencil’s input is greater than the dimensionality of the stencil’s window and function the implication is that the window is flat in the high dimensional space. For example, a $3 \times 3$ window function provided a four dimensional space would be interpreted as a $3 \times 3 \times 1 \times 1$ window function for that space.

No matter the dimensionality, the data path description assembly is only helpful as an intermediate if there are high level languages than can be compiled into it.
3.1.5 Intermediate representations for hardware generation

Intermediate representations like Data path description assembly appears to be a common design pattern in the implementation of generative systems. The idea of a specified-precision, write-once DAG is common in systems including Labview\cite{13, 127} or as an intermediate representation as in Spiral\cite{132, 130} and high level synthesis\cite{53, 49}. Data path description assembler’s inclusion of dimensional register types is common for intermediate representations of linear algebra \cite{110, 32, 188, 6, 38}. Even the inclusion of parameters to describe how the functions are invoked and organized mirror many of the parameters the underly pragmas in high level synthesis, describing specific properties of the computation\cite{53, 49}.

Specifically, we were inspired by LLVM’s\cite{106, 105} intermediate byte code representation which serves to separate the problem parsing a front-end high level language and the process of assembling and optimizing an instruction sequence.

Our contribution here isn’t that such an intermediate representation is practically useful. Other authors have already made arguments for such representations\cite{106, 105, 13, 127, 132, 130, 53, 49, 6, 38}. Our contribution is identifying how that practically useful intermediate can be formalized as an expression of extreme locality.

3.1.6 Intermediate representations of stencil computation

Hameed\cite{82, 80}, Qadeer\cite{161, 162}, and Luzhou\cite{121} present intermediate representations of stencil computation. Our data path description assembly extends the computation model that underlies these representations. Further, the representation we propose, while useful in the generation of hardware, is general and not tightly tied to a specific architecture.

Qadeer and Hameed\cite{82, 80, 161, 162} present a convolutional compute model (Eq. 2.1) as the basis for their instruction set extensions. We extend this compute model in two ways. First, the model is extended to support arbitrary windowed image functions removing the restriction that the operations be convolutional. This generality is important in supporting the broader set of image signal processing, computer vision, and computational photography applications. For example, while SIFT\cite{118} kernels like difference of Gaussian can be formulated as a convolutional operator, other SIFT kernels like Harris\cite{143, 85} do not fit
cleanly in this model\(^6\). Second, the stencil compute model extends the convolutional compute model to incorporate the concept of interconnected kernels. This extension is required in order to express the producer and consumer locality that allows for pipelines of kernels to increase their aggregate compute to bandwidth ratio.

Aside from the computation model, the data path description assembler differs from the convolution engine instruction set extensions\([82, 80, 161, 162]\) as their assembly specifically targets their convolution engine execution engine while our assembler is abstracted from the implementation. Luzhou\([121]\) presented a stencil domain specific language for systolic arrays focused on scientific computation. In the same way that the convolutional instruction extensions conformed to the convolution engine, Luzhou’s assembly is specific to the execution of Jacobi kernels, a specific kind of stencil kernel, on his systolic array.

In order to demonstrate the utility and underlying energy efficiency of the stencil virtual machine model, we show how fixed function hardware can be generated from data path description assembler.

### 3.2 Generating fixed function hardware

While a programmable architecture is often desirable, there will be applications which are critical enough to require the advantages of being significantly reduced in flexibility. This is the reason most current image signal processing products are fixed function designs. We generate that fixed function hardware by building a template (Sec. 2.4.4) of the micro-architecture for each stencil kernel (Sec. 3.2.1), setting each design’s parameters to the kernel definition (Sec. 3.1.2), and then interconnecting those kernel engines according to the topology (Sec. 3.1.3). Such a system dramatically reduces the cost of creating custom hardware or to create an FPGA implementation. That generator represents not only a validation of the abstract architecture, but a benchmark for understanding the energy efficiency of any future work exploring more flexible architectures (Sec. 3.3). Methodologically, we organized this as a one button flow which takes the DPDA produced by Darkroom as the input configuration to a generator written in Genesis2 capable of producing the Verilog of a fixed function implementation for both an ASIC flow and an FPGA flow (Sec. 3.2.2).
3.2. GENERATING FIXED FUNCTION HARDWARE

Figure 3.7: More detailed than Fig. 2.3, the stencil engine kernel microarchitecture for two dimensional images show how the abstract line buffer is implemented using a traditional memory for the line buffer array and a set of transposing buffers to provide the required data flow. The figure depicts the configuration for a stencil kernel which operates on a $3 \times 3$ window.

3.2.1 Stencil engine microarchitecture

Figure 3.7 is a diagram outlining the micro-architecture for a stencil kernel. As mentioned before (Sec. 3.1) an application is built by connecting kernels together. A kernel is composed of: a line buffer array, transpose buffer, stencil registers, arithmetic data path, pipeline registers, FIFO’s, and control logic. The color codes used in Figure 3.7 match the colors used in Figures 3.14 and 3.15 in the results section (Sec. 3.3).

Line buffer array

The first component is the line buffer array, highlighted in the color green in Figure 3.7. The purpose of the line buffer memory array is to store all of the data required for the window of re-use that occurs in a stencil kernel. Minimizing the energy of the line buffer array requires the fewest possible number of read cycles and a regular read pattern that supports wider fetches. However, optimizing these concerns in the memory array results in a mismatch in the read pattern and the write pattern.

As shown in the canonical architecture (Fig. 2.3), the write side of the array receives

\[\text{Qadeer and Hameed get around this issue by running these kernels on the vector core or scalar core}\]
pixels in row major order while the read side of the array must produce columns of pixels. While, these columns are also row major in order, the contents are shared across rows and reread. So we can make two choices. First, we can design the line buffer array to match the read-side pattern, where an array entry contains a pixel column, and an array read results in the required pixel column. Though, this column major organization would require incremental writes to the entries to make sure that the subsequent read contains the updated column. Second, we can design the line buffer array to match the write-side pattern, where an array entry contains a subset of the pixel row, and an array read results in the generation of multiple partial columns. This has the advantage of eliminating the incremental update writes, but this row major organization requires an additional buffer to collect multiple reads into a column.

In this first column major configuration, a single read is required to generate the column of pixels for the stencil register. For a $3 \times 3$ kernel, pictured in Figure 3.7, the array entries would be formatted such that a three pixel entry would contain $i, j, i + 1, j, i + 2$, while the next entry would contain $i + 1, j, i + 1, j + 1, i + 1, j + 2$. However, after this entry is read, it must be updated so it contain $i, j + 1, i, j + 2, i, j + 3$. In the cases where pixel $i, j + 3$ arrive at the line buffer array as this entry is read, that read can be followed by a partial write of that entry. So this column major organization would require one column read and one column partial write for every pixel calculation.

The second, row major, configuration would format the array entry such that pixels produced consecutively by a preceding stage could be stored in the same entry. For a $3 \times 3$ kernel, a four pixel entry would be used and it would contain pixels $i, j, i+1, j, i+2, j, i+3, j$, and the next entry would contain $i + 4, j, i + 5, j, i + 6, j, i + 7, j$ unless $i + 4$ exceeded the row width. To read a three pixel column which contains $i, j$ from this array, three array entries would need to be read. However, after reading three array entries, four successive pixel columns could then be buffered. This buffering of row major entries to build the pixel columns is done in the transpose buffer. An entry would be overwritten once it had been read three times. So, this column major organization would require three reads and one write for every four pixels calculated.

From the perspective of minimizing the number of read operations to the memory array, the row major choice will result in the fewest number of memory transactions. Additionally, it provides a greater degree in flexibility in that the line buffer array entry size is not tightly coupled to the pixel column height. For our stencil engine template microarchitecture we
implemented this row major line buffer array. However, as we mentioned the second design requires a transpose buffer which incurs some energy cost and area cost. Future work might explore this column major vs row major line buffer implementation trade-off. Such a change would not change the required capacity for the line buffer array.

To reduce the requirements on the memory blocks, the logic design assumes that the memory array is single ported. Increasing the read and write pixel rate of the array is accomplished by increasing the number of pixels \( p_e \) stored in the entry for kernel. If a new pixel is received from the preceding stage every cycle, then the array must be written once every \( p_e \) cycles.\(^7\) The remaining cycles account for the required read bandwidth.

For a new pixel to be calculated every cycle, the window of pixels must be updated every cycle. In a row major traversal this corresponds to shifting in a new a column of pixels.\(^7\) The column’s height is equivalent to the kernel window height \( R_y \). So the total number of pixels that must be stored in an entry to produce the required kernel pixel read and write rate is:

\[
p_e = R_y + 1 \tag{3.2}
\]

The size of the line buffer can be set by either the vertical height of the kernel’s window or the by the buffering requirements of successive stages. In the case of a straight pipeline the kernel’s line buffer capacity \( C \) is determined only by the window of pixel re-use in the stencil (Fig. 3.8), and the pixel occupancy can therefore be calculated as the row major distance between the leading pixel and the trailing pixel based on the image width \( W \) and the kernel window height \( R_y \) (Eq. 3.3).

\[
C = (R_y - 1) \ast W \tag{3.3}
\]

If you are observing closely, Equation 3.3 does not account for three additional pixels shaded in the window of reuse in Figure 3.8. While these pixel are stored in the kernel engine, they do not need to be stored in the line buffer array. The three trailing pixels don’t need to be stored in the line buffer as they are already stored in the stencil register and will not be read again.

In the case where the stencil kernels’ interconnection includes a fan-out followed by a

\(^7\)ignoring issues with up-sample and down-sample in the kernel or preceding kernel
fan-in, additional buffering is required to accommodate the concurrent working set. A fan-out is when multiple kernels read from the same intermediate image, while a fan-in is when a single kernel reads from multiple images. A fan-out therefore creates a scenario where the window of re-use is from the leading pixel of the leading kernel to the trailing pixel of the trailing kernel. We can minimize this affect by scheduling the leading and trailing kernel windows to overlap as much as possible, effectively making the window of reuse equivalent to the window of reuse of the tallest kernel. However, a fan-in creates constraints on the schedule by deciding the relationship of the two kernels to each other, bounding the flexibility in the schedule. For example, Figures 3.9 and 3.10 show the additional buffering required for image $x$ or image $y_2$ when the shorter path’s “delay” doesn’t match the longer path when converging at the fan-in of kernel $d_w$.

There is some flexibility in how that additional buffering is allocated. That flexibility allows us to exchange buffering from an expensive line buffer array for buffering in a cheap line buffer array. Figures 3.9 and 3.10 show two naively calculable buffering strategies that
underly this trade-off.

Figure 3.9: The amount of buffering for each stage the application pipeline is visualized using the same method as in Figure 3.8. In this example the buffering for the input image $x$ is greater than either of the naively calculated occupancies for the stencil kernel $f_w1$ or $g_w$. The additional buffering required is to provide the full working support implied by the other “longer” path.

Figure 3.9 utilized the naive strategy of keeping the leading pixel of each line buffer consistent with the furthest leading pixel of the stencil registers. This is calculated linearly in the number of kernels by walking backwards breadth-first from the outputs to the inputs and forcing the leading pixel for each window to be produced just in time by the prior stage. At the fan-out stage this means that one kernel needs the intermediate before another kernel due to a disparity in delay creating the difference between the leading kernel and trailing kernel.

Figure 3.10: The amount of buffering for each stage the application pipeline is visualized using the same method as in Figure 3.8. In this example the buffering for the input image $y_1$ is greater than the support in Figure 3.9 because the support for $x$ has been reduced by allowing $g_w$ to work ahead.

An alternative naive strategy shown in Figure 3.10, is to make the trailing pixel of every line buffer match the trailing pixel of every stencil register. This is calculated linearly in the
number of kernels by iterating through the solution for Figure 3.9 and forcing trailing pixels of fan-out kernels to match. This requires that some kernels’ line buffers will be working ahead of the kernel datapath by storing pixels ahead of the kernel’s leading pixel.

In the case where the buffer for $x$ is more expensive than the buffer for $y_2$ then Figure 3.9 would be the optimal solution for this pipeline. Conversely, if buffer $y_2$ is more expensive than the buffer for $x$ then Figure 3.10 would be the optimal solution for this pipeline. The area cost of a line buffer is determined by the number of bits per pixel. The energy cost of a line buffer is determined by the number of reads per pixel (determined by the window height) and the number of bits per pixel.

This flexibility is important as it allows for an arbitrage in buffer allocation. This arbitrage can be done in terms of the area expense by the source compiler before DPDA is generated\cite{87} as this is solely determined by the number of bits stored in the array. Each of the stencil kernel $P$’s line buffer capacity $C_P$ can be formed as an optimization problem based on the graphical constraints show in Figures 3.9 and 3.10. This is equivalent to the integer linear programming (ILP) approach based on register retiming\cite{113} presented by Hegarty for Darkroom\cite{87}.

**Transpose buffer**

The transpose buffer highlighted in the color brown in Figure 3.7, is the structure which allows for the row major entries of the line buffer array to be buffered into the sequence of stencil window update pixel columns.

As shown in Figure 3.7, the structure is two interconnected circular buffers. The first circular buffer interface has $R_y$ set of $R_y + 1$ pixel entries and is write only. Each entry is written over a common bus using an enable signal to selectively latch the write bus contents. $R_y$ consecutive writes are used to build an $R_y \times (R_y + 1)$ window of pixels. The second circular buffer interface has $R_y + 1$ set of $R_y$ pixel entries and is read only. The read is implemented through a mux, though a more a more efficient approach might include a common tri-state bus for the read as in a traditional memory array. $R_y + 1$ consecutive reads are used to generate $R_y$ pixel update columns for the shift register In all a complete write and read cycle takes $2R_y + 1$ clock cycles.

Because the shift networks are intermingled and the structure can not be written while being read, two transpose buffers are allocated. This ping-pong buffering\footnote{alternating the buffers between a read only state and a write only state} allows for the
update column pixels to be generated once per cycle.

The total number of pixels stored in the kernel’s transpose buffer $p_{\text{tran}}$ can be calculated as:

$$p_{\text{tran}} = 2(R_y + 1)R_y$$

Stencil shift registers

The stencil shift registers, highlighted in the color red in Figure 3.7, are the part of the memory hierarchy that contains the sliding window that will be immediately operated on and captures the data re-use that occurs between consecutive windows. The capacity of the kernel’s stencil register in pixels $p_{sr}$ can be directly calculated as:

$$p_{sr} = R_yR_x$$

It is implemented as a shift register where the entire contents of the register can be read by the arithmetic data path.

Arithmetic data path

The arithmetic data path implements the window function and is highlighted in the color orange. This implements the function described in the abstract architecture (Sec. 2.1.2) and is equivalent to interpreting the data path assembly (Sec. 3.1.1) as a module level netlist.

For example, the $7 \times 7$ sum of accumulated differences in Figure 3.2 would generate 147 subtract unit, 147 absolute value units, and three 49-to-1 summation units. To hit a specified cycle time, registers are added to the functional units to pipeline them. In terms of reporting this cost, the arithmetic is considered independently of the pipeline registers required to meet the target clock frequency.

Pipeline registers

The pipe stage registers highlighted in the color yellow. These are considered independently of the cost of the arithmetic as they represent the practical cost of achieving realistic clock
rates. The number of pipe stages required for a given arithmetic data path is calculated by finding the longest combinatorial latency in the arithmetic DAG and then solving for the number of clock periods while providing for sufficient clock margins.

For ASIC synthesis, this would require placing all of those pipeline registers at the beginning or end of the data path and allowing the re-timing tool\cite{113} to determine their location in the datapath. For synthesis flows where re-timing has not been fully or robustly implemented, as in some older FPGA flows, the flip flops in the data path can be distributed according to the combinatorial delay using a greedy module level version of Leiserson re-timing\cite{113}.

**FIFOs**

The FIFO’s, highlighted in the color turquoise, provide the communication between stencil kernel instance. These FIFO’s follow directly from the FIFO’s described in the abstract architecture (Sec. 2.1.2).

These FIFO’s serve to provide the ready-valid hand shake interface that allows for flow control between stencil kernel hardware instances and for the interface to the stencil application as a whole. Principally this is to allow for the generation of hardware that is robust to non-deterministic rates of global inputs and outputs. For example, a shared interface to global memory may stall unpredictably or the interface to the camera controller may idle some number of cycles between each row of pixels.

The FIFO’s are relatively small, holding only two pixels each. Principally their function is to reduce the complexity of distributing a stall signal for the whole application pipeline. Generating and distributing stall signals with significant scope can be expensive or infeasible due to the silicon areas involved. The hand-shake interfaces provided between stencil kernel instances also allows for stall signals to be locally determined, generated, and distributed for a kernel instance rather than for the whole application. In cases where this interface is unnecessary it can be procedurally eliminated\cite{97}.

When the input interface of the kernel engine is starved, the datapath pipeline continues to run until there are no new windows to process in the line buffer array. So the pipeline will drain all of its work in a starvation condition. When the output interface of kernel engine is blocked, the datapath pipeline will continue to run as long as it output does not have results to be latched in the FIFO. This back-pressure does not prevent the line buffer array from continuing to fill from the input interface.
3.2. GENERATING FIXED FUNCTION HARDWARE

Many of the control signals that determine this flow control and other behavior are locally calculated and locally distributed.

Control logic

The control logic, highlighted in the color dark blue, encompasses the control for the line buffer array, the transpose buffer, the stencil register, the coefficient registers, and the global control interface.

The line buffer controller maintains the write address and read address in the memory array while controlling the shift signals inside the transpose registers. It also generates a local set of signals which are taken together globally with other local signals to determine whether the aggregate of stencil kernels are busy versus idle.

The stencil core control logic determines whether the product of the arithmetic pipeline is valid on any given cycle. The controller also sets the shift signals and flush signals on the shift register to account for pixel address increments and row returns. It also generates a local set of signals which advertise that the math core is busy or idle.

Globally, the control provides an interface to allow: the coefficients to be written, the pipeline to receive commands (eg. run), and allow pipeline status to be read (eg. idle, running). For our FPGA implementation this is coordinated through C-code run on the ARM core which communicates over an AXI interface. The AXI interface is implemented using the interface generator described in Danowitz’s dissertation[58], which converts all of the AXI commands into local operations on control status registers (CSR).\footnote{borrowing the Intel blue book terminology for registers which contain state relevant to the control and status of various micro-architecture components}

The CSRs are distributed locally. To distribute a global “run” command the run CSR is set in each kernel engine. To read a global “idle” signal from the pipeline, the local status “idle” registers are globally reduced to a global “idle” virtual register. This distinction between global and local state of the pipeline is to allow for multiple jobs to be concurrently resident in the pipeline. For example, as job “A” completes, job “B” can begin entering the pipeline. All status registers are held locally to the kernel pipeline and read, written, and cleared through memory mapped AXI operations.

Similarly, the coefficients are held locally. They are actually distributed in two sets of registers. The first register contains the coefficients that are ready by the current computation of the pipeline and are local to the execution unit that reads them. The second
register, referred to as a shadow register, holds the value currently written through the AXI memory mapped interface. The shadow registers are simultaneously copied through to the local registers when a kernel level signal indicates a coefficient read-enable. This double buffered coefficient configuration allows for the configuration of the coefficients for the next job “B” before job “A” has completed reducing the overhead for configuration.

This AXI interface was required in order to build a functioning pipeline that could be mapped to an FPGA and run in real time. These FPGA mapped designs are an important part of our methodology which included the validation of these real-time designs.

3.2.2 Methodology

![Diagram of design flow]

Figure 3.11: Stencil engine design flow demonstrates the repeated design pattern of building domain specific abstractions on top of meta-programming environments in order to generate design instances consumable by tradition development flows.

The design flow is visualized in Figure 3.11. The methodology is organized into a one button flow with three major steps. First, the Darkroom program is compiled into the
DPDA format[87]. Second, the DPDA is assembled into a SystemVerilog design using a chip generator[185, 178, 181]. Third the SystemVerilog design is verified and synthesized using industry standard tools.

Darkroom[87] is a Lua[14] based DSL written in Terra[64]. A Darkroom program is written in terms of functions on images whose support in the input image is statically analyzable. The Darkroom compiler can translate these image functions into DPDA stencil kernels and a combination of these functions into a DPDA stencil application.

DPDA is used as a module level netlist for the design and is fed directly to the stencil engine generator. Figure 3.1 shows the result of translating the Darkroom into DPDA.

![Diagram](image)

Figure 3.12: This stencil engine generator call graph shows how the generators call each other on the left to produce the design hierarchy in the middle. Abstractly the flow conforms to the pattern of a set of generator invocations resulting in design components.

A stencil path generator consumes the DPDA and invokes a stencil kernel generator and line buffer generator to implement each stencil kernel instance. The stencil kernel generator further invokes a data path description assembler generator to build the arithmetic data path from the assembly. This process is summarized in Figure 3.12 as a hierarchy of hardware
generators and their inputs.

The stencil path generator constructs the DAG of stencil kernels and line buffers. It is responsible for calling the required generators, calculating the required working sets, and then interconnecting the generated instances. The configuration of this unit is determined by the interconnection of stencils described in DPDA.

The stencil engine generator is invoked for each stencil kernel and creates a stencil register coupled to a pipelined data path. The pipelined data path is generated by calling a DPDA generator. This generator is also responsible for generating all of the control logic for the data path and setting up the input and output FIFO’s. Much of the configuration of this unit is based on the stencil parameters.

The data path is generated using a general data path generator which consumes the assembler of the data path assembly descriptor (Sec. 3.1.1). This is equivalent to interpreting the assembly as a module level netlist and then generating a new module for each instruction based on the implied precision and operation while interconnecting these modules with wires implied by register declarations. This is a general module that could be used in any generator with an emphasis on arithmetic data paths.

Currently this flow does not generate the SRAM arrays.\textsuperscript{10} Instead the flow makes a query to cacti in order to estimate the costs of the array. On the other hand, when generating code for FPGA synthesis, the SRAM array is tagged for implementation as a BRAM.

As mentioned the result of the generator is a SystemVerilog design. Producing a SystemVerilog design has the convenient property of being compatible with current industry standard tools,\textsuperscript{11,12} which was used to verify the design and synthesize hardware. The simulator used is Synopsys VCS coupled to a test bench written in Python. The synthesis tools used is Synopsys Design Compiler invoked using topographical mode to provide the impacts of placement and routing in the prediction of the quality of design. The synthesis for FPGA is accomplished using Synopsys Synphony which generates a netlist that Xilinx ISE can use for place and route to finally generate a bit file.

Verification is based on three components. A number of assertions distributed through the generated SystemVerilog check some of the formal properties of the design[186, 182].

\textsuperscript{10}though in a production environment where SRAM generators and static SRAM libraries are available this would be feasible.

\textsuperscript{11}Though there is some hoop jumping required in order to gain compatibility with the FPGA tools which traditionally utilize dated versions of Verilog or VHDL.

\textsuperscript{12}This also creates an issue for generic portability because there are no fully complete SystemVerilog simulators available in the open source community.
3.3. RESULTS

Figure 3.13: FPGA prototype platform for the stencil engine based architecture using Xilinx Zynq ZC702 development board.

The image resulting from running the Verilog simulation is compared to the result from an x86 binary generated from Darkroom is used to check very high level behavior. Finally, the design is mapped to an Zynq ZC702 development board connected to a VITA 2000 image sensor (Fig. 3.13). The Zynq series of parts combines a dual-core ARM A9 processor with Artix and Kintex-series FPGA fabric.

Activity factors are extracted from the RTL simulation in order to calculate the power consumption of the design. The energy presented later in this chapter are based on the combination of this calculated power and the query to Cacti for the power consumption of the line buffer arrays. The total area of the design is reported as $1.4 \times$ the total cell area, which is a conservative 70% cell density. The cell library used for synthesis is a 45nm, nominal threshold voltage, 0.9V domain voltage, library from a leading foundry. The result of this area and energy analysis are provided in the next section.

3.3 Results

This section presents the results of synthesizing the designs in a 45nm CMOS technology. We compare the on-die cost (Sec. 3.3.1) of the stencil engines generated for each application to other published ASIC designs (Sec. 3.3.2) and argue that the on-die costs of these system are close to their lower bound (Sec. 3.3.3). These results show that there are applications in this space that are not dominated by global memory energy (Sec. 3.3.4), a fundamental concern in changing the computation (Sec. 1.3). All of the results here are for a system.
running with a clock frequency of 1.0GHz and an image that is 256 × 256 pixels in size.\textsuperscript{13}

### 3.3.1 Hardware cost

![Diagram showing energy per pixel for different applications](image)

Figure 3.14: ASIC implementation’s energy efficiency for each application. Applications are sorted from smallest energy at the top to greatest energy at the bottom. The energy cost is broken into the relevant architectural components and colored to match Figure 3.7.

The energy per pixel is shown in Figure 3.14 while the area per performance is shown in Figure 3.15. As discussed earlier (Sec. 1.1), energy per task and area per throughput are the figures of merit for the energy cost and area cost of extremely parallel systems. The stacked costs are color-coded to match the microarchitecture shown in Figure 3.7. The only cost not discussed earlier is the energy cost for distributing the clock using a clock tree shown in the color gray. This clock tree energy was an estimate made available by the topological analysis of Design Compiler.

These results easily point out the importance of algorithm design and optimization. Harris and Fast9 fulfill relatively similar roles in computer vision pipelines by detecting

\textsuperscript{13}While this may seem like a small image size, many of the results are reported in terms of per pixel cost. Larger images will be a very similar cost, this is discussed further in Chapter 4.
3.3. RESULTS

Figure 3.15: ASIC implementation’s area efficiency for each application. Applications are sorted from smallest energy at the top to greatest energy at the bottom. The area cost is broken into the relevant architectural components and colored to match Figure 3.7.

interesting points in the image. While the scope of this work did not include an analysis of how these two kinds of computation differ in the quality of their results\textsuperscript{14} given their similar quality of result it is clear that some algorithm implementations are trivially better in both energy and area. In this case, FAST has been presented as a superior algorithm to Harris\textsuperscript{[170]} and replaces the role of Harris in newer feature extraction pipelines\textsuperscript{[114]}. FAST is a much cheaper kernel than Harris as it is based on finding gradients using morphological operators favoring low precision comparators and Boolean operations rather than determinants favoring high precision multiplies.

3.3.2 ASIC quality design

It is difficult to compare these results against commercial chips for two reasons. First vendors rarely publish detailed specifications we need for comparison, and when that data

\textsuperscript{14} usually measured in terms of repeat-ability or invariance to scale, shift, rotation, or other affine rotations in finding the same points, the uniqueness or quality of those points, and the quantity of points found\textsuperscript{[191]}
is published the definition of an operation varies wildly.

For our results the number of operations is calculated as the sum of all scalar operations in the DPDA program after a program is converted to scalar operations. Figure 3.16 shows the number of operations required to calculate each pixel. One problem with this definition is that binary operators can misrepresent the arithmetic complexity of some applications. For example, Canny utilizes a morphological filter to implement the hysteresis kernel rather than the branch based filter suggested for a programmable architecture. As a result most of its operations are actually single bit ops. Though similar in nature, FAST is organized into bit string operations rather than single bit operations, so its numbers are less effected.

The energy per op is calculated from the energy pixel and operation count and is shown in Figure 3.17. Prior work has suggested that a 45nm ASIC will on average achieve an efficiency of about $1\frac{pJ}{op}$ without considering the off-chip memory costs[127]. The average application in this work achieved an efficiency of about $0.4\frac{pJ}{op}$ with a range from $0.1pJ$ to $1.3pJ$. Richardson-Lucy Deconvolution and the Harris corner detector both have a relatively

---

15Published systems do not generally include any global memory communication costs when presenting their efficiency.
3.3. RESULTS

Figure 3.17: ASIC implementation’s energy per operation for each application. This is calculated by dividing the total energy per pixel of an application (fig. 3.14) by the number of operations per pixel in that application (fig. 3.16). The darker dashed line indicates the average energy efficiency for published fabricated ASIC designs\[127]. The lighter dashed line indicates the average energy efficiency across applications.

High energy per operation compared to the other applications because they rely on higher precision operations. By comparison Fast9, which is a morphological corner detector, has a significantly lower per operation cost because it maintains a low precision for many of its operations and performs much of its calculation using binary ops.

The performance per unit area is calculated from the area per pixel throughput and the operation count and is shown in Figure 3.18. Prior work has suggested that a 45nm ASIC will on average achieve an efficiency of about 125\(\frac{Gops}{mm^2}\)[127]. The mean application achieves a throughput density slightly north of 1\(\frac{TOps}{mm^2}\). The high area and energy efficiency of these solutions shows the power of creating application optimized solutions, and clearly shows that the overhead of using generators to create those solutions, if it exists at all, is small.
Figure 3.18: ASIC implementation’s performance density for each application. The performance density is calculated by taking the operations per pixel of an application (fig. 3.16) and dividing it by the area cost to achieve a pixel throughput (fig. 3.15). The darker dashed line indicates the average area efficiency for published fabricated ASIC designs[127]. The lighter dashed line indicates the average area efficiency across stencil engine applications.

### 3.3.3 Understanding the (in)efficiency of ASIC design

While the energy and area costs are good, it is interesting to examine these numbers to see what percentages come from the functional units and what percentage is from other memory or overhead structures. The cost of the line buffer array, the stencil register, and the combinational arithmetic can be thought of as the costs that are fundamental to this kind of calculation. In that sense their cost represent an estimate for the absolute lower bound on the cost of an ASIC implementation.

Figure 3.19 presents that total energy per pixel of each application from Figure 3.14 normalized to the total cost to better show the relative energy cost of each component. On average the fundamental costs represent about 45% of the total cost of the system, indicating that the total cost is likely around 2× the absolute lower bound implementation cost.
3.3. RESULTS

Figure 3.19: ASIC implementation’s normalized energy cost relative to fundamental cost. Stacked costs are organized so that the more fundamental costs appear towards the left. The dashed line indicates the average percentage of area cost consumed by fundamental components.

Figure 3.20 presents that total area efficiency of each application from Figure 3.15 normalized to the total cost to better show the relative energy cost of each component. On average the fundamental costs represent about 60% of the total cost of the system, indicating that the total cost is likely around $2 \times$ the absolute lower bound implementation cost.

Note that this area argument is slightly nuanced as the proportion of the design cost consumed by the line buffer array can be arbitrarily increased to accommodate larger working sets. Chapter 4 presents exactly what proportion of the total cost the line buffer array should consume. Naturally, that proportion found there is similar to the proportions presented here.

Additionally, the decision between which costs are fundamental and which costs are “overhead” is somewhat arbitrary. It would be possible to argue for the inclusion of additional costs. However, costs like control, pipelining, and clock distribution, which this
3.3.4 Amortizing global memory energy

As the efficiency of the computing core improves it takes more operations to match the cost of fetching an operand from DRAM. This is clearly shown in Figure 3.21, which incorporates the cost of DRAM energy into the earlier energy per pixel from Figure 3.14. DRAM energy cost is estimated to be $20\frac{nJ}{bit}[124, 125]$ for pixels read from memory and written back as a final result. For example, the FCAM application reads 8 bits per pixel, writes 24 bits per pixel (RGB) as a final result, and has a total DRAM energy of $640pJ$.

With the exception of FCAM and SLIC, many applications energy per pixel is still significantly dominated by the cost of reading from and writing to DRAM. However, this isn’t as discouraging as this might first seem. Many of these algorithms exist as components that are multiply invoked in processing pipelines. For, example any algorithm that reads directly
3.3. RESULTS

Figure 3.21: ASIC implementation's energy cost including DRAM per application from a sensor would include a pipeline like FCAM in order to reduce noise and generate an image format that is more natural to process. Further, applications like Harris, FAST, and Canny are feature detectors that are usually invoked many times and are coordinated with other processing steps in order to build a robust feature extraction pipeline\[114, 118, 192]. This high DRAM cost is exactly the reason for long image pipelines, to amortize the high DRAM energy cost over as many operations as possible.

Practically, it would be useful to understand how long such a pipeline should be and how many operations it should execute so that DRAM energy consumes less than 50% of the total system energy and the global memory communication is amortized. Application developers could then use this as basis for increasing the number of arithmetic operations per global memory operation. Traditionally, in image signal processors this is accomplished by adding more processing stages to a pipeline, for example, expanding the simple camera pipeline (Fig. 2.2) to the FrankenCamera pipeline (Sec. 2.1.1). The way we measure the length of a pipeline is by calculating its effective stencil (Eq. 2.7), which represents the stencil computations window size if all kernels were fused (Eq. 2.6). This metric is useful when combined with the image size as it determines the total number of pixels stored in
the system (Eq. 3.3).

To create an estimate of the number of required operations, we assume that a pipeline’s arithmetic complexity scales linearly with its effective stencil. This models a pipeline extended by kernels which are similar to the kernels already in that pipeline. For example, scaling RGB-Conv-5x5 by a factor of 2× would be equivalent to running two 5 × 5 convolutions back to back. For applications like SLIC and Richardson-Lucy Deconvolution this is a very good model as increasing the number of iterations unrolled on die would simply involve duplicating the kernels that already exist. We discuss this model more thoroughly in Section 4.4.

<table>
<thead>
<tr>
<th>Application</th>
<th>Dram energy per pixel (pJ)</th>
<th>Energy per op (pJ)</th>
<th>Required OpCount</th>
<th>Required Window Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLIC</td>
<td>640.0</td>
<td>0.237</td>
<td>2704.0 (1.1×)</td>
<td>(42, 42)</td>
</tr>
<tr>
<td>RL-Deconvolution</td>
<td>1280.0</td>
<td>0.827</td>
<td>1549.0 (2.8×)</td>
<td>(43, 43)</td>
</tr>
<tr>
<td>FCam</td>
<td>640.0</td>
<td>0.171</td>
<td>3746.0 (1.7×)</td>
<td>(42, 42)</td>
</tr>
<tr>
<td>Stereo</td>
<td>960.0</td>
<td>0.087</td>
<td>11040.0 (4.0×)</td>
<td>(149, 149)</td>
</tr>
<tr>
<td>canny</td>
<td>520.0</td>
<td>0.075</td>
<td>6968.0 (4.3×)</td>
<td>(82, 82)</td>
</tr>
<tr>
<td>Harris</td>
<td>500.0</td>
<td>1.271</td>
<td>394.0 (5.5×)</td>
<td>(50, 50)</td>
</tr>
<tr>
<td>RGB-Conv-5x5</td>
<td>960.0</td>
<td>0.445</td>
<td>2159.0 (12.6×)</td>
<td>(63, 63)</td>
</tr>
<tr>
<td>Fast9</td>
<td>500.0</td>
<td>0.108</td>
<td>4612.0 (11.4×)</td>
<td>(80, 80)</td>
</tr>
<tr>
<td>Mean Application</td>
<td>750.0</td>
<td>0.403</td>
<td>1864.0</td>
<td>(68, 68)</td>
</tr>
</tbody>
</table>

Table 3.13: Operation count required to amortize DRAM energy

The exact amount that these algorithms need to be scaled up to amortize DRAM, such that DRAM is 50% of the total system energy, is quantified in Table 3.13 in the parenthetical of the required operation count column. For example SLIC requires a 1.1× increase in operation count while Fast9 requires a 11.4× increase in operation count.

The result on average is that an application requires order of 2000 ops/pix while operating on a working set that is the result of an effective stencil window of about 100 × 100 pixels. Generally applications which use a large number of low bitwidth operations like FAST, Canny, and Stereo require larger stencils while applications that depend on high precision operations like FCAM and Harris need a smaller stencil. Chapter 4 discusses an alternate definition for an “ideal” amount of work in an application (Sec. 4.4).
3.4 Stencil engine contributions and related work

The stencil engine is a template based generator (Sec. 2.4.4) that allows us to transform a domain specific language into hardware (Sec. 2.4.1), increasing the design productivity of building fixed function hardware (Sec. 2.4.3). In Section 2.4 we discussed prior work in the use of procedural design techniques in reducing the design effort of creating hardware. While such an approach greatly reduces the non-recurring cost of engineering[178], there is a concern that this proceduralization leaves some ASIC design efficiency on the table. Galal’s floating point generator[75] provides an example for how small this gap might actually be, though the results here were compared against a popular industry floating point generator. Our work builds on this by showing how larger procedurally generated systems have a similarly small gap (Sec. 3.3.2). While Richardson’s FFT generator[168] might achieve such a level of efficiency, ASIC quality of result was not reported as a part of their contribution. Milder’s DFT generator[129] based on SPIRAL[160] has the same level of efficiency as our work at about 1.0pJ/op and 1 TOps/mm² for fixed point implementations.[16] Taken together, Milder’s DFT generator and our Stencil generator show that, for extreme locality and high energy efficiency applications, it is possible to procedurally generate hardware at the quality of hand tuned ASIC RTL.

We used that high quality procedural generator to show that this specific subset of extreme locality applications, stencil computation, is highly energy efficient in practice. Our work demonstrated an efficiency improvement over Qadeer and Hameed’s Convolution Engine where our mean energy per op and harmonic mean performance density are 15× smaller and 53× greater respectively[81].[17,18] Going beyond the stencil computation model, prior work has shown the practical efficiency of building hardware that supports extreme locality systems. As we mentioned, Milder’s DFT generator showed how the digital signal processing computation model can achieve similar efficiencies. Similarly, Chen’s convolutional neural network accelerator[44] achieves an efficiency that is three order’s magnitude improvement over GPU’s. In the case of Milder’s floating point DFT generator[129] and Pedram’s systolic array based matrix multiply[155], the efficiency was limited to two and one order of magnitude improvement over GPU’s respectively. So, taken together with

[16] after converting from 65nm to 45nm
[17] With respect to their fixed function unit our design is approximately 3× smaller energy cost and 10× greater performance density
[18] This was not a comparison of the efficiency of the same application implemented in both systems
prior work, extreme locality applications lead to high energy efficiency in practice. Though, for extreme locality applications with high precision integer or float point arithmetic this efficiency gain is limited.

In Chapter 4 we extend our Stencil engine framework by examining the area and energy trade-off in working set size and pipeline size, which have implications for how we might build programmable hardware to execute this energy efficient extreme locality computation.
Chapter 4

Optimization of a Stencil Engine

At the heart of this thesis is the minimization of energy cost \( \frac{E}{pix} \) and area cost \( \frac{mm^2}{pix/ns} \) for stencil operators. Chapter 3 presented a design that minimize both energy and area costs by reducing or eliminating expensive system overheads. However, some design choices represent a trade-off in area and energy cost. This chapter uses relative marginal cost analysis, an analytic method for evaluating how a design parameter affects the energy and area cost in a framework, to discuss the analytic properties of several optimizations and trade-offs (Sec. 4.1). This includes working set partitioning (Sec. 4.3) and application partitioning (Sec. 4.4). The value of this framework is that the cost models are based on the abstract architecture rather than the specifics of the hardware generator. So a more flexible design whose structure is informed by the execution of the stencil virtual machine model can be optimized by the same framework.

4.1 Relative marginal cost analysis

As mentioned this thesis is primarily concerned with minimizing two objectives: energy cost \( E = \frac{pJ}{pix} \) and area cost \( A_p = \frac{mm^2}{pix/ns} \) (Sec. 1.1). From this perspective the problem of making design choices has been defined in terms of a dual objective function. This naturally leads to a definition for a Pareto optimal design (Fig. 1.3) as a design whose energy cost and area cost is such that there is no other design that has both smaller energy cost and smaller area cost. The ambiguity in such a design space definition is that there are often a set of Pareto optimal designs.

A classical parameterization of this space would solve for the weighted sum of the two
objective functions. In this case the optimal design would exist at the point on the Pareto frontier with a slope or marginal cost $M$ of the design relative to its parameter $P$ (eq. 4.1).

$$M = \frac{\delta E}{\delta P} / \frac{\delta A}{\delta P}$$  \hspace{1cm} (4.1)

If the marginal cost for a design choice happens to be positive, this indicates that both the energy cost and area cost can be simultaneously reduced by changing $P$. So, any design with positive marginal costs can be trivially optimized by simply altering $P$ such that the system enters the part of the design space where marginal cost is negative.

On the other hand systems in which the marginal costs are negative but unequal across all of the design choices are sub-optimal as there are arbitrage opportunities, allowing for an expensive design choice to be traded against a less expensive design choice for equivalent value at smaller cost. From this perspective an optimal design is one in which the marginal costs of all parameters are equivalent, and all negative, and therefore no opportunities for arbitrage or optimization exist. Making design choices, therefore, becomes a matter of identifying where on the Pareto frontier a design should exist.

Unfortunately, $M$ is not dimensionless, which makes setting its value difficult. It is often more convenient to parameterize in terms of a normalized or relative marginal cost (eq. 4.2). This relative cost model also fits with the natural language of design (eg. $2 \times$ better, or $30\%$ smaller). It also makes discussion and comparison of global $P$ that are common across many designs (eg. domain voltage, threshold voltage, clock frequency, circuit style) much easier.

$$M = \frac{\delta E / E}{\delta A / A}$$  \hspace{1cm} (4.2)

For the stencil abstract machine model energy efficiency is managed by balancing the cost of off chip communication with the cost of on-die computation while the area efficiency is managed by balancing the cost of on-die storage for the working set versus the area cost of the functional units. Partitioning the image reduces on die storage requirements but increases global memory traffic.

---

1 this is equivalent to optimizing a single objective by differentiating the weighted joint objective function with respect the design parameter $P$ and the solving for zero.

2 a notable exception is when a parameter is constrained in such a way that it cannot be further traded. In this case the solution lies along the simplex created by such constraints.
4.2 Working set partitioning

A sufficiently large image is, for practical purposes, an infinitely large image. In the designs presented earlier (Sec. 3), the line buffer for a $256 \times 256$ image consumed about 30% of the area (Fig. 3.20). If a design were required to support more realistic image sizes\(^3\) the area allocated to the line buffer would dominate the area cost of the design.\(^4\)

![Diagram of block partitioning](image)

Figure 4.1: Row major traversal of block partitions results in unnecessary refetch and recompute at the bottom and top of each partition. Iterating the blocks in a column major traversal allows for the boundaries at the top and bottom of the pixel to be captured in the line buffer. A column major traversal of block partitions would be equivalent to stripping (Fig. 4.2)

The total on die buffering required can be reduced by partitioning the working set into more manageably sized chunks. This is a well known approach to optimizing application

---

\(^3\) a modern SLR will generate 18.0MPix or $5184 \times 3456$ or $20 \times$ the image width

\(^4\) a $7 \times$ total area increase in which the line buffer would represent 90% of the total design area cost.
performance given a fixed memory hierarchy\cite{89}. The trade-off here is that smaller partitions require a greater number of refetches. So decreasing partition size will generally reduce performance, increase energy, and decrease total area. Figure 4.1, shows the regions of recomputation and refetch in red for a blocked image computation where blocks are iterated in a row major order.

By iterating the blocks in a column major order, the refetch inside a column can be eliminated by buffering the tail of the prior block in the line buffer for the next block. This section will use stripping, an approach from Zhu, that parameterizes the partitioning in terms of the width of the partition\cite{220}. Figure 4.2, shows the new regions of recomputation and refetch for stripping. Because the height of the blocks no longer affect the refetch rate, this parameter is assumed to be the full image height, and working set partitioning is only parameterized by the strip width $W$.

![Diagram](image.png)

Figure 4.2: Column major traversal of block partitions eliminates the refetch at the column boundary and removes any dependence on the block height. Instead the partitions can be considered strips as opposed to blocks. Refetch and recomputation still occur at the strip boundaries.
4.3. **OPTIMAL WORKING SET PARTITION SIZE**

The trade-off for increasing this parameter $W$ lies in the linear increase of aggregate line buffer capacity $C$ (Eq. 4.3) and the hyperbolic reduction in refetch per pixel $\Gamma_{pix}$ (Eq. 4.4). The aggregate line buffer capacity is the sum of capacity of each kernel $P$. When the strip width is identical for all kernels, this can be simplified to the product of the effective stencil (Eq. 2.7) height $R_y$ and the strip width $W$. The refetch $\Gamma_{pix}$ per pixel is the ratio of refetched pixel $R_x - 1$ to the strips width $W + R_x - 1$ (Fig. 4.2) which can be approximated as $\frac{R_x}{W}$.

$$C = \sum_{P=0}^{N-1} (R_y^P - 1) \cdot (W^P) \approx R_y \cdot W \quad (4.3)$$

$$\Gamma_{pix} = \frac{R_x - 1}{W + R_x - 1} \approx \frac{R_x}{W} \quad (4.4)$$

For example, the $5 \times 5$ convolution considered in Chapter 3 would have a refetch and recompute of $5/256$ such that the total energy per pixel would increase by 2% when operating on infinite images. On the other hand cascading 20 of these kernels in series would result in an effective stencil of $81 \times 81$, which would have a total energy per pixel increase of 32% due to the $81/256$ refetch and recompute per pixel.

The optimal strip width is determined by balancing increasing refetch with decreasing line buffer area to match the marginal cost for the system.

### 4.3. Optimal working set partition size

To generate a marginal cost expression, the area cost $A_p$ (Eq. 4.5) and energy cost $E$ (Eq. 4.6) are written to include the affect of strip width $W$. The total line buffer area is modeled as the approximate number of pixels in line buffers $R_y \cdot W$ multiplied by the average line buffer area cost per pixel $A_{pixel}$. The total area is then the cost of the line buffer added to the area cost of all of the non-line-buffer logic $A_{logic}$. The decreasing performance due to recomputation is modeled by the additional number of pixels that must be recomputed per pixel $R_x/W$. The energy is similarly calculate as the sum of the global memory energy per pixel $E_{DRAM}$ and the stencil engine’s energy $E_{SE}$. The increased energy

---

5Looking at Figure 4.2 you may assume that the numerator of $\Gamma_{pix}$ should be $\approx 2(R_x - 1)$ as a strip appears to have a recompute region of $R_x - 1$ on either side. However, looking closer you will note that for $N$ strips there would be $N - 1 \approx N$ recompute regions, so we can attribute on recompute region to each strip.
due to recomputation and refetch is modeled by scaling the compute energy by the amount of pixels refetched and recomputed per pixel $R_x/W$.

$$A_p = (A_{pixel}R_yW + A_{logic})(1 + \frac{R_x}{W}) \quad (4.5)$$

$$E = (\frac{R_x}{W} + 1)(E_{DRAM} + E_{SE}) \quad (4.6)$$

The costs can be differentiated with respect to the strip-width.

$$\frac{\delta A_p}{\delta W} = R_yk_a \left( 1 + \frac{R_x}{W} \right) - \frac{R_x}{W^2} (WR_yk_a + k_l) \quad (4.7)$$

$$\frac{\delta E}{\delta W} = -\frac{R_x}{W^2} (k_c + k_d) \quad (4.8)$$

Normalizing the derivatives and creating a ratio results in a marginal cost expression (Eq. 4.9):

$$M = -\frac{R_x (WR_yk_a + k_l)}{W^2R_yk_a - R_xk_l} \quad (4.9)$$

The strip-width can then be calculated by solving the quadratic system. The results in a function for $W$ dependent on the desired marginal cost $M$ (Eq. 4.10).

$$W = \frac{R_x}{2M} + \sqrt{(\frac{R_x}{2M})^2 + \frac{(M - 1)A_{logic}R_x}{MA_{pixel}R_y}} \quad (4.10)$$

It is important at this point to note, that this marginal cost and thus the strip-width are not directly dependent on the energy ratio of global memory energy cost $E_{DRAM}$ to compute energy cost $E_{SE}$. Instead the only terms from energy that are important are the relative increase in energy due to refetch and recompute.

Additionally, from this expression alone we can identify the limit for an area efficient

---

6The actual energy model is much more complicated. In a pipeline the recomputation of each kernel is reduced arithmetically as it approaches the end of the pipeline. Additionally, as the working set increases the energy cost of accessing a line buffer array increases. Both effects are ignored in the analytic model and only the second is considered in the empirical analysis.
4.3. **OPTIMAL WORKING SET PARTITION SIZE**

Design $M \rightarrow -\infty$ as the point at which (Eq. 4.11):

$$W_{M \rightarrow -\infty} = \sqrt{\frac{R_x k_l}{R_y k_a}}$$ (4.11)

This $M \rightarrow -\infty$ point represents the crossover point in the design space where the reduction in area due to decreasing working set partition size is perfectly matched with decreasing performance due to recomputation. Reducing the strip size further would result in increasing area cost because of the increase in required computation. From another perspective, in order for marginal cost to be negative, the inequality of Equation 4.12 must be true. This is equivalent to saying that all optimal designs have the property that the portion of line buffer array area in the design must be greater than the portion of pixels that are refetched and recomputed. This proportionality inequality makes perfect sense given the area performance crossover.

$$\frac{WA_{pixel} R_y}{A_{logic} + WA_{pixel} R_y} \geq \frac{R_x}{W + R_x}$$ (4.12)

This inequality also defines a hard boundary on the energy increase due to refetch and recompute relative the minimum possible energy (Eq 4.13).

$$\frac{E_{M \rightarrow -\infty}}{E_{M = -0}} = \left(1 + \sqrt{\frac{A_{pixel} R_x R_y}{A_{logic}}}ight)$$ (4.13)

So, while there is finite limit on how much energy can increase due to refetch and recompute that is often practically small (eg. Canny is approximately 1.3×), this is no such limit on how much the area cost can increase. This implies that the trade-off is flat and that, because of the hyperbolic trend which causes $M$ to approach $-0$ quickly with respect to $W$, One might think that most of the reasonable designs will lie relatively close to the $M = -1$ design on the knee of the curve. Yet because there is an energy and latency cost to blocking, the most energy efficient design point is where $W$ is the image width. In fact current image signal processors do not partition the working set and stream directly from the image sensor. The formulation here does not consider the energy cost of storing the input image in DRAM.
Using Equation 4.10 and the system costs calculated from the designs presented earlier in Chapter 3 the strip width for various marginal costs can be calculated. Those strip widths for each application are presented in Table 4.1.

<table>
<thead>
<tr>
<th></th>
<th>M</th>
<th>fast9</th>
<th>C-5x5</th>
<th>SLIC</th>
<th>RL-D</th>
<th>Stereo</th>
<th>Harris</th>
<th>Canny</th>
<th>FCam</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_x$</td>
<td>7</td>
<td>5</td>
<td>37</td>
<td>15</td>
<td>37</td>
<td>9</td>
<td>19</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>$R_y$</td>
<td>7</td>
<td>5</td>
<td>37</td>
<td>15</td>
<td>5</td>
<td>9</td>
<td>19</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>$W$</td>
<td>$-\infty$</td>
<td>83</td>
<td>60</td>
<td>65</td>
<td>75</td>
<td>131</td>
<td>92</td>
<td>101</td>
<td>301</td>
</tr>
<tr>
<td>$W$</td>
<td>$-2$</td>
<td>107</td>
<td>75</td>
<td>82</td>
<td>94</td>
<td>167</td>
<td>123</td>
<td>128</td>
<td>378</td>
</tr>
<tr>
<td>$W$</td>
<td>$-1$</td>
<td>128</td>
<td>89</td>
<td>96</td>
<td>109</td>
<td>198</td>
<td>150</td>
<td>151</td>
<td>445</td>
</tr>
<tr>
<td>$W$</td>
<td>$-1/2$</td>
<td>164</td>
<td>113</td>
<td>121</td>
<td>136</td>
<td>253</td>
<td>201</td>
<td>192</td>
<td>560</td>
</tr>
<tr>
<td>$W$</td>
<td>$-1/50$</td>
<td>387</td>
<td>249</td>
<td>256</td>
<td>277</td>
<td>1752</td>
<td>2061</td>
<td>421</td>
<td>1202</td>
</tr>
</tbody>
</table>

Table 4.1: The marginal cost analysis for optimal application strip width resulted in the values in the table listed above.

Figure 4.3: Energy and area trade-off for application strip width

Figures 4.3, 4.4, 4.5 were generated using the values of Table 4.1 to seed a design space sampling of $W$ using the methodology described in Section 3.2.2. Figure 4.3 shows the area and energy efficiency trade-off parameterized by strip width. Figure 4.4 shows the
increasing area cost as strip width increases. Figure 4.5 shows the high energy cost at small strip widths and the increasing energy cost at higher strip widths. These results include the total system cost including global memory operations.

![Area per Performance graph](image)

Figure 4.4: Area per unit of performance as a function of strip width

The increasing energy cost for larger strip sizes is caused by the increasing energy cost of an array access into a larger memory array. This increased energy due to increasing energy cost of a larger line buffer size was not modeled in this relative marginal cost analysis. This is another factor beyond the shallow curve that make large strip width undesirable and favor designs closer to the knee of the curve.

As the efficiency of the system decreases and computational area costs increase, as they would in a more flexible system, the efficient working set size will increase in the root of that inefficiency. For example, an FPGA with 10× the ASIC area cost \( A_{\text{logic}} \) would require 3× the working set partition size. So even if the storage efficiency \( A_{\text{pixel}} \) of flexible systems remains relatively constant compared to an ASIC implementation, decreases in the efficiency of the computational area cost will require other increases in area cost. Revisiting the FCAM application in Figure 3.15 an increase in computational area cost by 10× would result in roughly \( .2 \cdot 3 + .8 \cdot 10 = 9 \times \) increase in area cost.
4.4 Optimal application partition size

Returning to the earlier question (Sec. 3.3.4) regarding the “optimal” application size, the optimal strip-width’s dependence on the amount of “work” done indicates that there is an arbitrage opportunity when selecting how a very large application should be partitioned into multiple hardware passes. This partitioning of the application is required when there are not sufficient on-die hardware resources to map the entirety of the application.

Figure 4.6: Stencil applications which iterate the same set of kernels repeatedly can unroll the iteration loop to increase the amount of work done in a single pass.
4.4. **OPTIMAL APPLICATION PARTITION SIZE**

For example, Figure 4.6 relates how a long pipeline that executes a looped kernel repeatedly could be unrolled to increase the work per pass. In the domain of computer vision, clustering algorithms which attempt to locate local regions of similar pixels operate by running the same set of kernels repeatedly to iterate and find the solution. For example, this thesis benchmarks the SLIC super pixel segmenter (Sec. 3.3) which iterates a kernel for local k-means clustering.

Additionally, some application’s scope may exceed the total available on die resources. In a more flexible implementation of the stencil virtual machine model, the full application can be implemented by breaking the application into multiple iterations and re-mapping the resources between each iteration. In this model, each pass requires a memory barrier where the full intermediate results are written back into main memory.

So in evaluating the “ideal” application partition size, there is a trade-off between minimizing the global memory traffic energy required for intermediates with increasing partition size against the the decrease in area efficiency due to increasing partition size. Note that Section 3.3.4 presents a naive perspective to the question of ideal work by identifying the amount of work required such that global memory energy is roughly half of the total energy. This perspective will be called the amortization perspective.

Instead of thinking about this from the perspective of a single application that must be scaled up to amortize the cost of global memory accesses, instead we are concerned with the problem of a very large or practically infinite application. The model for how an applications partitioning is scaled is based on Figure 4.6. The idea here is that an application’s effective stencil, on die energy cost $E_{SE}$, and compute area cost $A_{logic}$ all scale linearly. Additionally, it assumes that an application when partitioned has a constant bisection cost, so $E_{DRAM}$ is only affected by the number of partitions not which partition. For example, two convolutions in series, an $8 \times 8$ followed by an $8 \times 8$, would have an effective stencil of $15 \times 15$. Splitting the pipeline in half into a pass for each kernel would reduce the work and effective stencil linearly.

Such a model is limited in two respects. First if a kernel needs to be partitioned, as might be the case for very large kernels like a Gaussian convolution for bilateral filtering, the work changes in the square of the effective stencil. For example, a partition of a $15 \times 15$ convolution kernel into an $8 \times 8$ convolution kernel would require 4 passes or a $4 \times$ reduction.

---

7 alternatively the reverse operation shows how an unrolled loop could be re-rolled to reduce work per pass
in work per pass. Second, actual application graphs can be fairly sophisticated[163, 164] and breaking an application might require creating more than one new memory barrier per partition. So finding ideal partitions of an actual application given a fixed on die resource would require solving variant of the minimum flow graph cuts problem. However, this simple linear model for work is helpful since it demonstrates how the phenomena works and provides a general estimate for the amount of work required for each marginal cost regime. Finally, even given its limitations many applications conform or could conform after some transformation to this model[87].

Altering the cost equation from Section 4.3, this section optimizes the size of a pass $s$ for an application. For $s = 1$ the application requires $T$ passes with a stencil size of $R_x$. Assuming a simple model for work where an applications stencil width can be linearly increased scaling the work, we are interested in the optimal effective stencil $R'_x = s \cdot R_x$. Equations 4.14 and 4.15 represent the cost of an $R'_x$ unit of work resulting in the relative marginal cost expressions in equations 4.16 and 4.17.$^{9}$

$$E = \frac{1}{s} \left( E_{DRAM} + s E_{SE} \right) \left( \frac{sR_x}{W} + 1 \right)$$  \hspace{1cm} (4.14)

$$A_p = \frac{1}{s} \left( sA_{pixel}R_yW + sA_{logic} \right) \left( \frac{sR_x}{W} + 1 \right)$$ \hspace{1cm} (4.15)

$$M_{R'_x} = -k_d + \frac{R'_x R'_k}{W} \left( k_d + \frac{R'_k}{R_x} \right)$$ \hspace{1cm} (4.16)

$$M_W = - \frac{R'_x \left( W R_y k_a + k_l \right)}{W^2 R_y k_a - R'_x k_l}$$ \hspace{1cm} (4.17)

The marginal cost with respect to strip width (Eq. 4.17) is similar to the earlier expression (Eq. 4.9) but includes terms which scale up the cost of the design. The numerator for marginal cost with respect to $R'_x$ (Eq. 4.16) is determined by the difference between the global memory energy cost per pixel and the per pixel energy cost of recomputing pixels at the boundary of the working set partition. The denominator is equivalent to the sum of refetch and recomputation energy. From this perspective, an area efficient design would

---

$^8$the number of partitions should be minimized to maximize throughput, the total flow should be minimized to reduce energy, and a partition is constrained to fit inside the resources that exist. Not only this, but individual kernels can be manually fissioned to produce better min flow opportunities and provide better packing into the available resources.

$^9$Note that formulating the cost as the total work $\frac{T}{s}$ vs $\frac{1}{s}$ results in the same marginal cost expression
have small recompute and refetch and an energy efficient design would balance the cost of recompute against the limit for global memory traffic.

The area efficiency asymptote $M \to -\infty$ for Equation 4.16 occurs as $R_x' \to 0$, the point at which there is an infinite amount of DRAM traffic but very little recompute. On the hand the energy efficiency limit $M = -0$ occurs as the energy cost of recompute $R_x' W k_x / R_x$ approaches the energy cost of fetching a pixel from global memory $E_{DRAM}$. However, if $W$ is increased simultaneously with $R_x'$ so that their marginal costs match this limit is unachievable and no energy efficiency limit exists\(^\text{10}\)

Each marginal cost equation can be solved in terms of the appropriate design parameters. The solution for strip width $W$ (eq. 4.18) is similar to the earlier result (eq. 4.10) and accommodates the scale-up factor $s = R_x' / R_x$. The solution for $W$ may appear to be independent of $R_x$. However, replacing the terms $A_{\text{logic}}$ and $R_y$ with $A_{\text{logic}} / R_x$ and $R_y / R_x$ respectively does not change the equation, but does show the affect of assuming that work increases linearly with with the scale-up factor $s$.

The solution for an ideal stencil $R_x'$ is given in Equation 4.19.

\[
W = -\frac{R_x'}{2M} \sqrt{\left(\frac{R_x'}{2M}\right)^2 + \frac{(M-1)A_{\text{logic}}R_x'}{MA_{\text{pixel}}R_y}}
\]  
\[
R_x' = \frac{1}{2}\frac{M - 1}{M} \frac{E_{DRAM}}{E_{SE/R_x}} \left(\sqrt{4W^2 \left(1 - \frac{M E_{SE}/R_x}{M^2 E_{DRAM}}\right)} + 1 - 1\right)
\]  

For reference, the amortization solution to $R_x'$ given a linear model for increase in work is effectively Equation 4.20).

\[
R_x' \geq \frac{E_{DRAM}}{E_{SE/R_x}}
\]  

As one would expect $R_x'$ is dependent on this amortization perspective, but is not directly $R_x \cdot E_{DRAM} / E_{SE}$. Making some simplifications and approximations $R_x'$ grows in the square root of the product of strip-width and this amortization perspective (eq. 4.21).

\(^{10}\)as determined by this model. Recall that increasingly larger working set partitions practically require more energy, which is not captured in this model. So it is likely that some crossover point exists. However, this solution would use very large line buffer widths.
可以使一系列简化应用到 $W$ (eq. 4.22)，将结果代入

\[ R'_x \propto \sqrt{\frac{W}{E_{DRAM}} \frac{E_{SE}}{R_x}} \]  

(4.21)

\[ W \propto \sqrt{\frac{A_{logic}}{A_{pixel} R_y}} \]  

(4.22)

\[ R'_x \propto \left( \frac{E_{DRAM}}{E_{SE}/R_x} \right)^2 \frac{A_{logic}/R_x}{A_{pixel} R_y/R_X} \right)^{1/3} \]  

(4.23)

\[ W \propto \left( \frac{E_{DRAM}}{E_{SE}/R_x} \left( \frac{A_{logic}/R_x}{A_{pixel} R_y/R_X} \right)^2 \right)^{1/3} \]  

(4.24)

Making a similar set of simplifications for $W$ (eq. 4.22), substituting the result into Equation 4.21, and solving for $R'_x$ results in the approximate proportionality independent of $W$ (eq. 4.23). So while the solution for $R'_x$ is strongly dependent on energy amortization $R_x \cdot E_{DRAM}/E_{SE}$ in the $\frac{2}{3}$ root; it is also dependent on area amortization in the $\frac{1}{3}$ root.

Using the energy and area costs found earlier (Sec. 3.3.4) and equations 4.18 and 4.19 the increased stencil size and appropriate strip width can be found for each marginal cost regime. These values are provided in Table 4.2. Additionally, the marginal cost and matching strip width is calculated for the stencil size of the original application and for the application size if it were scaled for the amortization perspective.

Applications scaled up to amortize DRAM energy\(^{11}\) had a mean marginal cost of $M = -1.7$. So, many of the implications for optimal application size we made earlier in Section 3.3.4 apply here for designs on the knee of the trade-off curve. Additionally, $W/R_x$ is about 5 for designs on the knee of the curve and is weekly dependent on $M$ until $M = -1/8$. Finally, the number of operations in an application is $\propto -1/M$, for example halving the marginal cost results in a doubling of the number of operations.

### 4.5 Contribution and related work

The observation that there is a trade-off between locality in the working set and the efficiency of that computation is not new\(^{155, 221, 177, 60}\). Ragen-Kelly proposed a similar trade-off for this computation between locality, parallelism, and recomputation\(^{164, 163}\). Zhu and

\(^{11}\) 50% of the total system energy was DRAM
then Hegarty present how image partitioning can be done in terms of a column major block traversal or image strips\[220, 87\]. Though they did not suggest what optimal working set sizes might look like.

<table>
<thead>
<tr>
<th>$R_x$</th>
<th>canny</th>
<th>harris</th>
<th>fast9</th>
<th>C-5x5</th>
<th>FCam</th>
<th>SLIC</th>
<th>RL-D</th>
<th>Stereo</th>
<th>Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Orig.</td>
<td>19</td>
<td>9</td>
<td>7</td>
<td>5</td>
<td>25</td>
<td>37</td>
<td>15</td>
<td>37</td>
<td>19</td>
</tr>
<tr>
<td>W</td>
<td>94</td>
<td>65</td>
<td>69</td>
<td>78</td>
<td>151</td>
<td>126</td>
<td>113</td>
<td>328</td>
<td>128</td>
</tr>
<tr>
<td>$Op$</td>
<td>1.6k</td>
<td>72</td>
<td>405</td>
<td>172</td>
<td>2.2k</td>
<td>2.4k</td>
<td>552</td>
<td>2.7k</td>
<td>1.2k</td>
</tr>
<tr>
<td>$W_x$</td>
<td>4.9</td>
<td>7.2</td>
<td>9.9</td>
<td>15.6</td>
<td>6.0</td>
<td>3.4</td>
<td>7.5</td>
<td>8.9</td>
<td>7.9</td>
</tr>
<tr>
<td>$M$</td>
<td>-4.28</td>
<td>-6.19</td>
<td>-9.16</td>
<td>-14.38</td>
<td>-3.76</td>
<td>-1.77</td>
<td>-4.79</td>
<td>-6.18</td>
<td>-6.3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$R_p$</th>
<th>canny</th>
<th>harris</th>
<th>fast9</th>
<th>C-5x5</th>
<th>FCam</th>
<th>SLIC</th>
<th>RL-D</th>
<th>Stereo</th>
<th>Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amor.</td>
<td>19</td>
<td>9</td>
<td>7</td>
<td>5</td>
<td>25</td>
<td>37</td>
<td>15</td>
<td>37</td>
<td>19</td>
</tr>
<tr>
<td>$W$</td>
<td>94</td>
<td>65</td>
<td>69</td>
<td>78</td>
<td>151</td>
<td>126</td>
<td>113</td>
<td>328</td>
<td>128</td>
</tr>
<tr>
<td>$Op$</td>
<td>1.6k</td>
<td>72</td>
<td>405</td>
<td>172</td>
<td>2.2k</td>
<td>2.4k</td>
<td>552</td>
<td>2.7k</td>
<td>1.2k</td>
</tr>
<tr>
<td>$W_x$</td>
<td>4.9</td>
<td>7.2</td>
<td>9.9</td>
<td>15.6</td>
<td>6.0</td>
<td>3.4</td>
<td>7.5</td>
<td>8.9</td>
<td>7.9</td>
</tr>
<tr>
<td>$M$</td>
<td>-4.28</td>
<td>-6.19</td>
<td>-9.16</td>
<td>-14.38</td>
<td>-3.76</td>
<td>-1.77</td>
<td>-4.79</td>
<td>-6.18</td>
<td>-6.3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$R_x$</th>
<th>canny</th>
<th>harris</th>
<th>fast9</th>
<th>C-5x5</th>
<th>FCam</th>
<th>SLIC</th>
<th>RL-D</th>
<th>Stereo</th>
<th>Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>−4</td>
<td>20</td>
<td>17</td>
<td>27</td>
<td>32</td>
<td>22</td>
<td>11</td>
<td>18</td>
<td>61</td>
<td>26</td>
</tr>
<tr>
<td>$W$</td>
<td>98</td>
<td>94</td>
<td>147</td>
<td>218</td>
<td>140</td>
<td>57</td>
<td>127</td>
<td>440</td>
<td>165</td>
</tr>
<tr>
<td>$Op$</td>
<td>1.7k</td>
<td>136</td>
<td>1.6k</td>
<td>1.1k</td>
<td>2.0k</td>
<td>717</td>
<td>662</td>
<td>4.5k</td>
<td>1.5k</td>
</tr>
<tr>
<td>$W_x$</td>
<td>4.9</td>
<td>5.5</td>
<td>5.4</td>
<td>6.8</td>
<td>6.4</td>
<td>5.2</td>
<td>7.1</td>
<td>7.2</td>
<td>6.1</td>
</tr>
<tr>
<td>$M$</td>
<td>150</td>
<td>62</td>
<td>101</td>
<td>62</td>
<td>52</td>
<td>62</td>
<td>31</td>
<td>99</td>
<td>77</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$R_p$</th>
<th>canny</th>
<th>harris</th>
<th>fast9</th>
<th>C-5x5</th>
<th>FCam</th>
<th>SLIC</th>
<th>RL-D</th>
<th>Stereo</th>
<th>Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amor.</td>
<td>19</td>
<td>9</td>
<td>7</td>
<td>5</td>
<td>25</td>
<td>37</td>
<td>15</td>
<td>37</td>
<td>19</td>
</tr>
<tr>
<td>$W$</td>
<td>94</td>
<td>65</td>
<td>69</td>
<td>78</td>
<td>151</td>
<td>126</td>
<td>113</td>
<td>328</td>
<td>128</td>
</tr>
<tr>
<td>$Op$</td>
<td>1.6k</td>
<td>72</td>
<td>405</td>
<td>172</td>
<td>2.2k</td>
<td>2.4k</td>
<td>552</td>
<td>2.7k</td>
<td>1.2k</td>
</tr>
<tr>
<td>$W_x$</td>
<td>4.9</td>
<td>7.2</td>
<td>9.9</td>
<td>15.6</td>
<td>6.0</td>
<td>3.4</td>
<td>7.5</td>
<td>8.9</td>
<td>7.9</td>
</tr>
<tr>
<td>$M$</td>
<td>-4.28</td>
<td>-6.19</td>
<td>-9.16</td>
<td>-14.38</td>
<td>-3.76</td>
<td>-1.77</td>
<td>-4.79</td>
<td>-6.18</td>
<td>-6.3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$R_x$</th>
<th>canny</th>
<th>harris</th>
<th>fast9</th>
<th>C-5x5</th>
<th>FCam</th>
<th>SLIC</th>
<th>RL-D</th>
<th>Stereo</th>
<th>Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>−4</td>
<td>20</td>
<td>17</td>
<td>27</td>
<td>32</td>
<td>22</td>
<td>11</td>
<td>18</td>
<td>61</td>
<td>26</td>
</tr>
<tr>
<td>$W$</td>
<td>98</td>
<td>94</td>
<td>147</td>
<td>218</td>
<td>140</td>
<td>57</td>
<td>127</td>
<td>440</td>
<td>165</td>
</tr>
<tr>
<td>$Op$</td>
<td>1.7k</td>
<td>136</td>
<td>1.6k</td>
<td>1.1k</td>
<td>2.0k</td>
<td>717</td>
<td>662</td>
<td>4.5k</td>
<td>1.5k</td>
</tr>
<tr>
<td>$W_x$</td>
<td>4.9</td>
<td>5.5</td>
<td>5.4</td>
<td>6.8</td>
<td>6.4</td>
<td>5.2</td>
<td>7.1</td>
<td>7.2</td>
<td>6.1</td>
</tr>
<tr>
<td>$M$</td>
<td>150</td>
<td>62</td>
<td>101</td>
<td>62</td>
<td>52</td>
<td>62</td>
<td>31</td>
<td>99</td>
<td>77</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$R_p$</th>
<th>canny</th>
<th>harris</th>
<th>fast9</th>
<th>C-5x5</th>
<th>FCam</th>
<th>SLIC</th>
<th>RL-D</th>
<th>Stereo</th>
<th>Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amor.</td>
<td>19</td>
<td>9</td>
<td>7</td>
<td>5</td>
<td>25</td>
<td>37</td>
<td>15</td>
<td>37</td>
<td>19</td>
</tr>
<tr>
<td>$W$</td>
<td>94</td>
<td>65</td>
<td>69</td>
<td>78</td>
<td>151</td>
<td>126</td>
<td>113</td>
<td>328</td>
<td>128</td>
</tr>
<tr>
<td>$Op$</td>
<td>1.6k</td>
<td>72</td>
<td>405</td>
<td>172</td>
<td>2.2k</td>
<td>2.4k</td>
<td>552</td>
<td>2.7k</td>
<td>1.2k</td>
</tr>
<tr>
<td>$W_x$</td>
<td>4.9</td>
<td>7.2</td>
<td>9.9</td>
<td>15.6</td>
<td>6.0</td>
<td>3.4</td>
<td>7.5</td>
<td>8.9</td>
<td>7.9</td>
</tr>
<tr>
<td>$M$</td>
<td>-4.28</td>
<td>-6.19</td>
<td>-9.16</td>
<td>-14.38</td>
<td>-3.76</td>
<td>-1.77</td>
<td>-4.79</td>
<td>-6.18</td>
<td>-6.3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$R_x$</th>
<th>canny</th>
<th>harris</th>
<th>fast9</th>
<th>C-5x5</th>
<th>FCam</th>
<th>SLIC</th>
<th>RL-D</th>
<th>Stereo</th>
<th>Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>−4</td>
<td>20</td>
<td>17</td>
<td>27</td>
<td>32</td>
<td>22</td>
<td>11</td>
<td>18</td>
<td>61</td>
<td>26</td>
</tr>
<tr>
<td>$W$</td>
<td>98</td>
<td>94</td>
<td>147</td>
<td>218</td>
<td>140</td>
<td>57</td>
<td>127</td>
<td>440</td>
<td>165</td>
</tr>
<tr>
<td>$Op$</td>
<td>1.7k</td>
<td>136</td>
<td>1.6k</td>
<td>1.1k</td>
<td>2.0k</td>
<td>717</td>
<td>662</td>
<td>4.5k</td>
<td>1.5k</td>
</tr>
<tr>
<td>$W_x$</td>
<td>4.9</td>
<td>5.5</td>
<td>5.4</td>
<td>6.8</td>
<td>6.4</td>
<td>5.2</td>
<td>7.1</td>
<td>7.2</td>
<td>6.1</td>
</tr>
<tr>
<td>$M$</td>
<td>150</td>
<td>62</td>
<td>101</td>
<td>62</td>
<td>52</td>
<td>62</td>
<td>31</td>
<td>99</td>
<td>77</td>
</tr>
</tbody>
</table>

Table 4.2: Marginal cost analysis for optimal application size. The original and amortization applications’ strip width were calculated for the marginal cost determined by its stencil size.
Our contribution here is identifying how this optimal working set size can be determined analytically in terms of the properties of the computation. This perspective is an improvement over the idea that these concerns can be parameterized but must be explored stochastically\cite{116, 117, 163}. Additionally, while this framework was used to justify the design of ASIC components the model is general enough to support arbitrary implementations of these applications.

The next chapter suggest further and future work that would explore the intersection for how the compute model and abstract architecture could be expanded while increasing the flexibility of the fabric that executes the virtual machine model.
Chapter 5

Conclusion

Energy efficient computation can be described as the space of applications which have a high compute to bandwidth ratio, while iterating through a small working set with significant locality, such that the dominant cost of the algorithms implementation in the system is the arithmetic. This extreme local computation (Sec. 1.3) forms the superset for the canonical image signal processor pipeline presented in this thesis (Sec. 2.1.2). The stencil virtual machine model (Sec. 3.1) formalizes how an application’s behavior can be described for execution in any hardware that implements this computation model and interprets the virtual machine model.

As a demonstration for how this works in practice, this thesis presents a stencil engine generator (Sec. 3.2), which can translate a specific application’s behavior described in the virtual machine model into fixed function hardware. Practically, such a process is useful in reducing the design effort for energy efficient SoC components. Further, the fixed function hardware it produces represents a validation of the abstract architecture and virtual machine model functionality. It also allowed us to verify the high energy efficiency of the system that motivated its design in the first place. This system cost provides an estimate for the practical lower bound of any implementation of these algorithms. Such a model is important as it suggests how and how-much both fixed function hardware and more flexible hardware can be optimized (Ch. 4). Generally, an efficient application partition will contain 10,000’s of operations with a working set for these applications on the order of 10,000’s to 100,000’s of pixels.

There are many applications which exhibit the properties of extreme locality computation but contain kernels that are not stencils. We could address this by either creating new
virtual machine models for those domains or extending the stencil virtual machine model to incorporate these other kinds of energy efficient computation. For example, the portions of computer vision which form the later stages of computation is composed primarily of linear algebra operations. Linear algebra operations like matrix multiplication represent a generalization of stencil kernels (e.g., image convolution can be represented as a tensor multiplication). Though the specification of real-time rendering pipelines might require a separate computation model entirely\[153].

While fixed function hardware can be extremely efficient, the lack of flexibility presents a huge opportunity cost that makes more flexible hardware desirable. Going forward, it is important to understand how the characteristics of a sub-domain of efficient computation determines the kind and detail of a compute fabric. We can start by identifying a template hardware (Sec. 2.4.4) representing the space of efficient but programmable architectures onto which our set of extreme locality computations can be efficiently and optimally compiled. Then, for an application candidate set, we could iteratively calcify instances of that template to create a smooth trade-off between programmability and energy efficiency. Naturally, such a template should include some portion of well known design patterns that we know to be broadly programmable and relatively energy efficient like FPGAs, GPUs, systolic arrays, and data flow processors.

Though, there is a significant system integration problem if we have a multitude of computation models and a diversity of these accelerators to run them on. Frameworks like Terra\[64] and Delite\[110], certainly provide the infrastructure for our polyglot application. Yet, we still need to identify methods to determine which computation should be run on which architecture and how those computations would efficiently communicate with each other at runtime. The goal here is to use the fact that our computation is restricted to known energy efficient design patterns to help us bridge the gap between energy efficient application processors composed of a many fixed function components and generally programmable but energy expensive processing fabrics.
Bibliography


[128] A. Mihailidis, B. Carmichael, and J. Boger. The use of computer vision in an intelligent environment to support aging-in-place, safety, and independence in the home.


[167] Justin Rattmer. Making the right hand turn to power efficient computing. 35th Microarchitecture Symposium. Keynote address.


[177] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan primitives for gpu computing. In Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS


[197] Texas Instruments, Texas Instruments, Post Office Box 655303, Dallas, Texas 75265. OMAP3525-HiRel and OMAP3530-HiRel Applications Processor, 1 edition, 8 2010.


ACM.


