Introduction
Almost all signal processing algorithms are initially represented as double precision floating-point in languages such as Matlab. For hardware implementations, these algorithms have to be converted to large precision fixed-point to have a sufficiently large dynamic range. However the inevitable quantization effects and the complexity of converting the floating-point algorithm into a fixed point one, limit the use of fixed-point arithmetic for high precision embedded computing. FPGAs have become an attractive option for implementing computationally intensive applications. However, the common conception has been that efficient FPGA implementations of floating-point arithmetic have a lot of performance, area and power overheads compared to fixed-point arithmetic. With recent technology advances, FPGA densities are increasing at a rate at which area considerations are becoming less significant. These advances have also reduced the performance and power overhead of floating-point arithmetic. With appropriate designs, floating-point applications can even be more efficient than fixed-point ones for large bitwidths. The overheads in the context of the overall application can be quite low. In this paper, we present a preliminary area, and power performance analysis of double precision matrix multiplication, an extensively used kernel in embedded computing and also show that FPGAs are good candidates for implementing high precision floating-point based applications when compared to a general-purpose processor.

Currently many FPGA based floating-point units, both open source [2] and commercial [1], are available. However, most of them consider only single precision floating-point operations, and do not make use of the recent advances in FPGAs. Moreover, an area, and power performance analysis of the floating-point units in the context of a common application is lacking.

Description of our Floating point units and the Matrix Multiply architecture
For matrix multiplication, we require add and multiply floating-point units. Our floating-point units follow the IEEE 754 single and double precision (64-bit) format. We developed both deeply pipelined and moderately pipelined units. The units essentially consist of three stages: denormalization, the add/multiply, and normalization/rounding/renormalization. Exception handling at all stages is done and enable/disable signals are provided for easy integration into a pipelined architecture. The implementation of floating-point units involves extensive use of fast fixed point adder/subtractors, multiplier units, and large bus multiplexers (for shifting operations). Recent FPGAs, such as Virtex-II Pro [4], provide a large number of embedded multipliers as well as fast carry chains for addition. Similarly, large multiplexers used in shifting can make use of the MUXCY, MUXF attributes on the FPGAs. Recent FPGA fabrics also contain a lot of registers, which can be utilized for extensive pipelining between stages.

We used the block matrix multiplication architecture from [3] in which a linear array of \( n \) processing elements is used for an \( n \times n \) matrix multiplication. Each processing elements essentially consists of an adder, a multiplier, storage elements, and related control logic. Since the matrix multiply architecture (see [3] for more details) is modular, multiple chips can be used in an array for large \( n \). Here we use the GFLOPS per device for a given \( n \) as the performance metric.

Analysis of the Floating-point units
Table 1a and 1b show a comparison of the fixed and floating-point units for a bitwidth of 32 and 64. We see that the overhead for double precision is less than that for single precision. Note that, for the fixed-point designs, truncation to make the output bitwidth equal to the input bitwidth results in a lot of quantization error. Moreover the fixed-point multiplier unit takes up more embedded multipliers than the floating-point unit. We also show a comparison between an extensively pipelined and a moderately pipelined version of the floating-point units. We see that extensive pipelining to increase the clock frequency requires a lot of area for the registers in between the pipeline stages. The pipelining done to split the adder/multiplier, the large priority encoder and the shift registers for the normalizing unit shows an immediate improvement in frequency, without much increase in area. Further pipelining, shows diminishing returns in frequency and the area increases significantly. Hence a design trade-off will be the frequency required which influences the number of pipelining stages and area. Here, for the double precision matrix multiply, we decided to use the moderately pipelined units since we can achieve higher GFLOPs. From synthesis results, we saw that normalization takes up a lot of area (560 slices for the deeply pipelined and 200 slices for the moderately pipelined units, for double precision) and can also be the critical path for timing (because of a large priority encoder and shift registers). Hence a design trade-off would be the use of custom formats in the architecture, with conversion from and back to the IEEE754 standard at the interface to say, a processor. Considering power, the 64bit fixed-point multiplier unit with more embedded multipliers consumes a lot more power. Note that, for the power values of individual units, only clocks, logic and signal powers were included.
Area, and Power Performance Analysis of a Floating-point based Application on FPGAs

Department of Electrical Engineering, Systems University of Southern California Los Angeles, CA 90089

Approved for public release, distribution unlimited

See also ADM001694, HPEC-6-Vol 1 ESC-TR-2003-081; High Performance Embedded Computing (HPEC) Workshop (7th)., The original document contains color images.
Analysis of the Matrix Multiply

The area and power performance overhead of the floating-point units has to be seen in the context of an application. Table 2 shows the area and power performance of both fixed and floating-point implementations of a double precision, \( n \)-point matrix multiply on a FPGA. The double precision implementation shows us an interesting result of the floating-point unit having a better performance than the fixed-point implementation. The maximum number of fixed-point processing elements that the device can accommodate when block RAMs are used for storage, is smaller than the number of processing elements when slice based RAM is used. This is probably because of more routing resources used up due to the fixed locations of the block RAMs and the embedded multipliers. Moreover, the number of slices on a given device being constant, the device will accommodate fewer processing elements if deeply pipelined units occupy a large area. Hence, the performance of the device might be lower even if the frequency of the units is high. Also, the overall application’s architecture’s operating frequency should be considered. Performance was measured as one multiplication and one addition happening every clock cycle in each processing element. The total power for the matrix multiply takes into account output, input, quiescent, logic, signals and the clocks power. We see that floating-point unit overheads in terms of area, and power performance are not too drastic. Table 3 shows the performance comparison of a floating-point based \( n \)-point matrix multiplication both on an FPGA and a Pentium4 SSE2, 1.5GHz processor. The performance of the design on FPGAs shows a 3.48x improvement over that of the processor. Moreover the power per GFLOP of the FPGA is much lower than that of the processor.

<table>
<thead>
<tr>
<th>Area (slices)</th>
<th>Max Freq. (MHz) achievable</th>
<th>Power (mW) at 100MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>36, 64bit Fixed-point (with 2, 4 pipeline stages)</td>
<td>250</td>
<td>23.48</td>
</tr>
<tr>
<td>32, 64bit Floating-point (with 18, 23 pipeline stages)</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Area (slices) / Embedded multipliers</th>
<th>Max Freq. (MHz) achievable</th>
<th>Power (mW) at 100MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>32, 64bit Fixed-point (with 5, 7 pipeline stages)</td>
<td>200</td>
<td>136.3</td>
</tr>
<tr>
<td>32, 64bit Floating-point (with 9, 11 pipeline stages)</td>
<td>140</td>
<td>804</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Area (slices) / BRAM / multipliers of each Processing element of matrix multiply</th>
<th>Maximum number of processing elements on the device</th>
<th>Frequency (MHz) of each element</th>
<th>Power of each PE (mW) at 100MHz</th>
<th>Frequency (MHz) achieved for the matrix multiply</th>
<th>Performance of matrix multiply, per device-Virtex2Pro xc2vp125-7</th>
<th>Total Power (W) perGOP or GFLOP for matrix multiply,</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed-point based using block RAM</td>
<td>Floating point based (moderately pipelined) using block RAM</td>
<td>Floating point based (deeply pipelined) (estimated)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Area (slices)</td>
<td>BRAM / multipliers</td>
<td>1344 / 4 / 16</td>
<td>1626 / 0 / 16</td>
<td>1872 / 4 / 10</td>
<td>3441 / 4 / 10</td>
<td></td>
</tr>
<tr>
<td>Maximum number of processing elements on the device</td>
<td>28</td>
<td>32</td>
<td>29</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Frequency (MHz) of each element</td>
<td>130</td>
<td>130</td>
<td>130</td>
<td>200</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Power of each PE (mW) at 100MHz</td>
<td>843</td>
<td>894</td>
<td>762</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Frequency (MHz) achieved for the matrix multiply</td>
<td>110</td>
<td>110</td>
<td>120</td>
<td>200</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Performance of matrix multiply, per device-Virtex2Pro xc2vp125-7</td>
<td>6.16 GOPS</td>
<td>7.04 GOPS</td>
<td>6.96 GFLOPS</td>
<td>6.4 GFLOPS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total Power (W) per GOP or GFLOP for matrix multiply,</td>
<td>27.2 / 6.16 = 4.41</td>
<td>33.04 / 7.04 = 4.68</td>
<td>26.4 / 6.96 = 3.79</td>
<td>-</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 3: A comparison of the performance of Matrix Multiply

<table>
<thead>
<tr>
<th></th>
<th>FPGA (Virtex2Pro xc2vp125-7)</th>
<th>Pentium4 with SSE2 (1.5GHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GFLOPS</td>
<td>6.96</td>
<td>2</td>
</tr>
<tr>
<td>Power (W) per GFLOP</td>
<td>26.4 / 6.96 = 3.79</td>
<td>57.9 / 2 = 28.95</td>
</tr>
</tbody>
</table>

All the above results were obtained after the VHDL code was synthesized and placed and routed using the Xilinx ISE5.2i, on a Virtex2Pro XC2VP125-7f1696 device. Power values were obtained from Xpower. The Pentium4 SSE2 results were from [5]. Better results can be obtained after the units have been optimized more, by manually placing them.

Conclusion and Future Work
We have presented a preliminary analysis of a floating-point implementation of a computationally intensive application on FPGAs. We show that when the floating-point units are considered in the context of an application, their overheads in terms of area, and power performance are not too drastic. We also show that a significant increase in performance can be obtained on FPGAs over general-purpose processors with much lower power expended. Future work will involve extensive analysis of the floating-point units to identify more design trade-offs. We will also provide a documented and extensively tested, open source library of the floating-point units, shortly.

References
Area and Power Performance Analysis of Floating-point based Applications on FPGAs

Gokul Govindu, Ling Zhuo, Seonil Choi, Padma Gundala, and Viktor K. Prasanna
Dept. of Electrical Engineering
University of Southern California
September 24, 2003

http://ceng.usc.edu/~prasanna
Outline

• Floating-point based Applications on FPGAs
• Floating-point Units
  – Area/Power Analysis
• Floating-point based Algorithm/Architecture Design
• Area, Power, Performance analysis for example kernels:
  – FFT
  – Matrix Multiply
• Conclusion
Floating-point based Applications on FPGAs

Applications requiring
• High numerical stability, faster numerical convergence
• Large dynamic range

Examples:
• Audio/Image processing, Radar/Sonar/Communication, etc.

Fixed-point vs. Floating-point
• Resources
  – Slices
• Latency/Throughput
  – Pipeline stages
  – Frequency
• Precision
• Design complexity of fixed/floating-point units

Energy – Area – Performance
Tradeoffs
Floating-point Device Options

FPGAs (Virtex II Pro)

More flexibility, Better performance per unit power

- Low-power Floating-point DSPs (TMS320C54X)
- Low-power Floating-point GPPs (PowerPC G4)
- High-performance Floating-point DSPs (TMS320C67X)
- High-performance Floating-point GPPs (Pentium 4)

- FPGAs (Virtex II Pro)

More flexibility, Better performance per unit power
Need for FPU Design in the Context of the Kernel

Integration

• Latency
  – Number of pipeline stages as a parameter
• Frequency
  – FPU frequency should match the frequency of the kernel/application’s logic
• Area/Frequency/Latency tradeoffs

Optimal Kernel Performance

• High throughput
  – Maximize frequency
• Minimize Energy
  – Architectural tradeoffs - FPUs parameterized in terms of latency/throughput/area
• Optimize F/A for FPU
  – Maximize the performance of the kernel

Algorithm/Architecture Design

• Re-evaluation of the algorithm/architecture
  – Tolerate latencies of FPU - low area vs. high frequency tradeoffs
  – Re-scheduling
Outline

• Floating-point based Applications on FPGAs
• **Floating-point Units**
  – Area/Power Analysis
• Floating-point based Algorithm/Architecture Design
• Area, Power, Performance analysis for example kernels:
  – FFT
  – Matrix Multiply
• Conclusion
Our Floating-point Units

• Now, easier to implement floating-point units on FPGAs
  – Optimized IP cores for fixed-point adders and multipliers
  – Fast priority encoders, comparators, shift registers, fast carry chains….

Our floating-point units

• Precision
  – Optimized for 32, 48 and 64 bits

• IEEE 754 format

• Number of pipeline stages
  – Number of pipeline stages parameterized
    • For easy integration of the units into the kernel
    • For a given kernel frequency, units with optimal pipelining and thus optimal resources, can be used

• Metrics
  – Frequency/Area
  – Overall performance of the kernel (using floating-point units)
  – Energy
Floating-point Adder/Subtractor

**Pipeline stages:** 6-18
**Area:** 390-550; **Achievable frequency:** 150-250MHz
**Xilinx XC2VP125 –7**
Frequency/ Area vs. Number of Pipeline Stages

• Diminishing returns beyond optimal F/A
• Tools’ optimization set as “balanced - area and speed”
  - Area and Speed optimization give different results in terms of area and speed
## Addition Units: Some Trade-offs

<table>
<thead>
<tr>
<th></th>
<th>Fixed-point</th>
<th>Floating-point</th>
<th>Floating-point</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>32 bits</td>
<td>64 bits</td>
<td>32 bits</td>
</tr>
<tr>
<td></td>
<td>with 2 stages</td>
<td>with 4 stages</td>
<td>with 14 stages</td>
</tr>
<tr>
<td><strong>Area (slices)</strong></td>
<td>36</td>
<td>139</td>
<td>485</td>
</tr>
<tr>
<td><strong>Max. Freq. (MHz) achievable</strong></td>
<td>250</td>
<td>230</td>
<td>230</td>
</tr>
<tr>
<td><strong>Power (mW) at 100MHz</strong></td>
<td>23.48</td>
<td>102</td>
<td>200</td>
</tr>
</tbody>
</table>

**Floating-point vs. Fixed-point**
- **Area**: 7x-15x
- **Speed**: 0.8x-1x
- **Power**: 5x-10x
Multiplier Units: Some Trade-offs

<table>
<thead>
<tr>
<th></th>
<th>Fixed-point</th>
<th>Floating-point</th>
<th>Floating-point</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>32 bits</td>
<td>64 bits</td>
<td>32 bits</td>
</tr>
<tr>
<td></td>
<td>with 5</td>
<td>with 7</td>
<td>with 7</td>
</tr>
<tr>
<td>stages</td>
<td>stages</td>
<td>stages</td>
<td>stages</td>
</tr>
<tr>
<td>Area(slices)/Embedded Multipliers</td>
<td>190/4</td>
<td>1024/16</td>
<td>180/3</td>
</tr>
<tr>
<td>Max. Freq. (MHz)</td>
<td>200</td>
<td>130</td>
<td>220</td>
</tr>
<tr>
<td>Achievable</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Power(mW) at 100MHz</td>
<td>136.3</td>
<td>414</td>
<td>227</td>
</tr>
</tbody>
</table>

Floating-point vs. Fixed-point
• Area: 0.9x-1.2x
• Speed: 1.1x-1.4x
• Power: 1x-1.6x
A Comparison of Floating-point units

Our units vs. the units from the **NEU** library*

<table>
<thead>
<tr>
<th></th>
<th>USC 32 bits</th>
<th>NEU 32 bits</th>
<th>USC 64 bits</th>
<th>NEU 64 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>F</td>
<td>A</td>
<td>F/A</td>
<td>F</td>
</tr>
<tr>
<td>Adder</td>
<td>250</td>
<td>551</td>
<td>.45</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>770</td>
<td>.07</td>
<td></td>
</tr>
<tr>
<td>Multiplier</td>
<td>250</td>
<td>182</td>
<td>1.4</td>
<td>95</td>
</tr>
<tr>
<td></td>
<td>90</td>
<td>477</td>
<td>.18</td>
<td></td>
</tr>
</tbody>
</table>

F: Frequency  
A: Slices

Outline

• Floating-point based Applications on FPGAs
• Floating-point Units
  – Area/Power Analysis
• Floating-point based Algorithm/Architecture Design
• Area, Power, Performance analysis for example kernels:
  – FFT
  – Matrix Multiply
• Conclusion
The Approach: Overview

1. Problem (kernel)
   - Algorithm & Architecture
   - Algorithm & Architecture

2. Performance model (Area, Time, Energy & Precision effects)
   - Design tools
     - Device

3. Tradeoff Analysis/Optimizations (Fixed vs. Floating-point)
   - Candidate designs
     - Implementation/ Low-level simulation

4. Domain
   - Refine performance model, if necessary

- e.g. Matrix multiplication

- Implement building blocks
- Estimate model parameters
- Algorithm & Architecture
- Performance model
- Design tools
- Device
- Tradeoff Analysis/Optimizations
- Candidate designs
- Implementation/ Low-level simulation
1. Domain

- FPGA is too fine-grained to model at high level
  - No fixed structure comparable to that of a general purpose processor
  - Difficult to model at high level
- A family of architectures and algorithms for a given kernel or application
  - E.g. matrix multiplication on a linear array
- Imposes an architecture on FPGAs
  - Facilitates high-level modeling and high-level performance analysis

Choose domains by analyzing algorithms and architectures for a given kernel
  - Tradeoffs in Area, Energy, Latency
2. Performance Modeling

• Domain Specific Modeling

• High-level model
  – Model parameters are specific to the domain
  – Design is composed based on the parameters
  – Design is abstracted to allow easier (but coarse) tradeoff analysis and design space exploration
  – Precision effects are studied
  – Only those parameters that make a significant impact on area and energy dissipation are identified

• Benefit: Rapid evaluation of architectures and algorithms without low-level simulation
  – Identify candidate designs that meet requirements
3. Tradeoff Analysis and Manual Design Space Exploration

- Vary model parameters to see the effect on performance
- Analyze tradeoffs
- Weed out designs that are not promising

Example: Energy Tradeoffs
4. Low Level Simulation of Candidate Designs

- Verify high-level estimation of area and energy for a design
- Select the best design within the range of the estimation error among candidate designs
- Similar to low-level simulation of components

![Diagram of simulation process]

- Candidate Designs → VHDL
- VHDL File
- Xilinx XST Synthesis
- Netlist
- Waveforms
- ModelSim
- Power
Outline

• Floating-point based Applications on FPGAs
• Floating-point Units
  – Area/Power Analysis
• Floating-point based Algorithm/Architecture Design
• Area, Power, Performance analysis for example kernels:
  – FFT
  – Matrix Multiply
• Conclusion
Example 1: FFT Architecture Design Tradeoffs

$n$-point FFT

Parallelism

Interconnect

Local Memory

Main Memory

Size $c$

I/O complexity: minimum information to be exchanged to solve a problem

For $n$-point FFT, I/O complexity = $\, ? \, (n \log n / \log c)$
FFT Architecture Design Tradeoffs (2)

For Radix-4, Possible parallelism?  
\[ 1 = Vp = 4 \]

Parallel or serial input?  

\[ n=16 \]

Can some twiddle factor computation be bypassed?  

Can the hardware for Stage 1 be shared with Stage 2  
Or  
More hardware?  
\[ 1 = Hp = \log_4 n \]
**FFT Architecture Design Trade-offs (3)**

- **Optimal FFT architectures with respect to EAT**
  - **Fixed-point**: $(V_p, H_p) = (1,4)$
  - **Floating-point**: $(V_p, H_p) = (4,1)$
Example 2: Matrix Multiplication
Architecture Design (1)

I/O Complexity of Matrix Multiplication

Interconnect

Parallelism

Local Memory

Main Memory

I/O complexity: minimum information to be exchanged to solve a problem

Size $c$

Theorem (Hong and Kung): For $n \times n$ matrix multiplication

$I/O$ complexity $= \frac{n^3}{c}$
Matrix Multiplication Architecture Design (2)

Processing Element Architecture*

Matrix Multiplication Architecture Design (3)

- Our design
  - Number of PEs = \( n \)
  - Storage = \( ? \ (n \ ? \ n) \)
  - Latency = \( ? \ (n^2) \)

- For \( n \times n \) matrix multiplication, I/O complexity = \( ? \ (n^3/\ ? \ c) \)

- Our design has optimal I/O complexity
### Performance of 32, 64 bits Floating-point Matrix Multiplication (4)

<table>
<thead>
<tr>
<th>Pipeline stages</th>
<th>32 bits XC2VP125 –7</th>
<th>64 bits XC2VP125 –7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Min</td>
<td>Max</td>
</tr>
<tr>
<td>Area(slices) of each Processing Element</td>
<td>718</td>
<td>991</td>
</tr>
<tr>
<td>Max. No. PEs</td>
<td>77</td>
<td>56</td>
</tr>
<tr>
<td>Achievable Frequency (MHz)</td>
<td>90</td>
<td>215</td>
</tr>
<tr>
<td>Sustained Performance (GFLOPS)</td>
<td>13.8</td>
<td>24.1</td>
</tr>
</tbody>
</table>

The performance (in GFLOPS) is maximum for the design with floating-point units with maximum frequency/area.
FPGA vs. Processor

<table>
<thead>
<tr>
<th></th>
<th>FPGA XC2VP125 –7 230MHz</th>
<th>TI TMS320 C6713* 225 MHz</th>
<th>Analog TigerSharc * 500 MHz</th>
<th>Pentium 4 SSE2 * 2.53 GHz</th>
<th>PowerPC G4 * 1.25 GHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>GFLOPS</td>
<td>24.7 (sustained)</td>
<td>1.325 (peak)</td>
<td>1.0 (peak)</td>
<td>6.56 (peak)</td>
<td>6.22 (peak)</td>
</tr>
<tr>
<td>Power(W)</td>
<td>26</td>
<td>1.8 (core power)</td>
<td>2.4 (core power)</td>
<td>59.3</td>
<td>30</td>
</tr>
<tr>
<td>GFLOPS/W</td>
<td>0.95</td>
<td>0.7</td>
<td>0.4166</td>
<td>0.11</td>
<td>0.2</td>
</tr>
</tbody>
</table>

FPGA vs. Processor
• Performance (in GFLOPS): up to 24.7x
• Performance/Power (in GFLOPS/W): up to 8.6x

* From data sheets
64 bits floating-point matrix multiplication on FPGA using our FPU and architecture

<table>
<thead>
<tr>
<th></th>
<th>FPGA XC2VP125 –7 200MHz</th>
<th>Pentium 4 SSE2 1.5 GHz*</th>
<th>AMD Athlon 1 GHz*</th>
</tr>
</thead>
<tbody>
<tr>
<td>GFLOPS</td>
<td>8.6 (sustained)</td>
<td>2.0 (peak)</td>
<td>1.1 (peak)</td>
</tr>
<tr>
<td>Power(W)</td>
<td>26</td>
<td>54.7</td>
<td>60</td>
</tr>
<tr>
<td>GFLOPS/W</td>
<td>0.33</td>
<td>0.036</td>
<td>0.018</td>
</tr>
</tbody>
</table>

FPGA vs. Processor
- Performance (in GFLOPS): up to 7.8x
- Performance/Power (in GFLOPS/W): up to 18.3x

* From data sheets
Conclusion

• Floating-point based implementations are not prohibitively expensive either in terms of area or latency or power
• High performance kernels can be designed with appropriate FPUs
• In terms of GFLOPS and GFLOPS/W, FPGAs offer significant over general purpose processors and DSPs

Future Work

• Floating-point based beamforming....
• Tool for automatic integration of FPUs into kernels

http://ceng.usc.edu/~prasanna
MILAN for System-Level Design: Design Flow

Model PARIS kernels, end-to-end application, hardware choices, mission parameters, etc.

PARIS design space

Dynamic programming based heuristics
Multi-rate application optimization
Interval arithmetic

ModelSim, XPower, PowerPC simulators
VHDL and C implementations
Energy, latency, and area estimates

Download-http://www.isis.vanderbilt.edu/Projects/milan/
Questions?

http://ceng.usc.edu/~prasanna