Smart Memories Software
M. Horowitz, W. Dally, P. Hanrahan, M. Lam, K. Olukotun, M. Rosenblum

Motivation
Smart memories architecture stresses conventional programming languages:
• Wide-word integer units require support
• New instructions need to be made visible to application developers
• Parallel execution must be found to divide up jobs so they can run on several tiles at once
  – Data parallelism can be extracted into stream (SIMD) execution
  – Data parallelism can also be divided into MIMD threads
  – Task parallelism can be split into multiple independent threads
• Libraries are much more complex internally
  – Must be able to deal with wide variety of data types and memory layouts
  – May have multiple varieties for different divisions of work

New Integer Data Types
Multimedia applications often use small and/or fractional fixed-point numbers:
• Need to be able to represent fixed-point values of all sizes and types in C

<table>
<thead>
<tr>
<th>Type</th>
<th>Range</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal 16-bit integers</td>
<td>0 to 65,535</td>
<td>16</td>
</tr>
<tr>
<td>Normal 32-bit integers</td>
<td>0 to 4,294,967</td>
<td>32</td>
</tr>
<tr>
<td>Normal 64-bit integers</td>
<td>0 to 18,446,729,055</td>
<td>64</td>
</tr>
<tr>
<td>Normal 128-bit integers</td>
<td>0 to 3.4028234663852886e+38</td>
<td>128</td>
</tr>
</tbody>
</table>

• A possible solution: “DSP Integers”
  – Definition contains (bit width, binary point location) specification. Example: int i((24,23));
  – Map to native architecture integer formats to provide extra precision, and not range, when there is no exact match
  – Explicit bit length aids compilation to multimedia SIMD instructions
• Other formats may also be desirable
  – Complex numbers
  – Short FP numbers

New C Operators
New operators need to be added to C to support new ISAs:
• Special new multimedia instructions are inaccessible
  – Only +, -, X, ÷, shift, logical, and bitfield operators are in C
• Several operators would be helpful
  – Saturating +/−, type conversion, permutation, pack/unpack, etc.
  – Other common operations found in a variety of multimedia ISAs

Finding parallelism in code
Most important job for language extensions are making parallelism easy to find and extract
• Task-style parallelism is marked using threading
  – Phrased can be used to mark most task division
  – Extensions to phrased have been added to ease the marking of common task-style parallelism such as data pipelines
• Data-style parallelism can be found implicitly
  – Constrained access arrays (CAAs), which cannot have C pointers to locations within the array, allow Fortran-style vectorizing of code
  – Vectorizing compiler technology allows parallelization of many normal loops in C code that have data-independent loop iterations
  – Loops with simple reduction accumulator dependencies can be parallelized by reassociation of the reduction functions
• Data-style parallelism can also be marked explicitly
  – Brook extensions allow a programmer to explicitly define parallel inputs to a special kernel loop structure
  – Explicit definition makes compilation much more predictable at the expense of a slightly more complicated programming model
• Both data-style techniques create portable code
  – Both rely on the compiler to lay out CAAs or stream buffers in the available local memory spaces
  – pragma options and other architecture-specific flags allow programmers to include occasional optimizations for a target architecture

New library support
Old-fashioned C libraries cannot provide enough flexibility and portability for PCA architectures
• A library call may map to several different library routines optimized in different ways
  – Different architectures
  – Different morph configurations
  – Different memory configurations or states
  – Different numbers or arrangements of tiles
  – We need to separate the call interface from the routine implementation in a simple and easily extendable manner!
• We need a meta-compilation layer to match calls to the appropriate routine at compile time
  – Need to provide “smart polymorphism” for different data types that may be used by the routine
  – A script associated with each call takes the state of the machine at the time of the call and chooses the best match from the library of available routines
  – Scripts can also generate some inline code to make the final routine decision at runtime, if necessary

Compiler memory analysis
Memory analysis and management in a PCA architecture with a complicated memory hierarchy can have a significant impact on performance with a streaming app
• Problem can be simplified into a 3-level hierarchy in most architectures
  – Compiler must dynamically manage the different layers of memory over the course of the program
  – This differs from conventional architectures, which normally have no compiler-controlled local memory (caches are implicitly controlled)

<table>
<thead>
<tr>
<th>Type</th>
<th>Range</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal 16-bit integers</td>
<td>0 to 65,535</td>
<td>16</td>
</tr>
<tr>
<td>Normal 32-bit integers</td>
<td>0 to 4,294,967</td>
<td>32</td>
</tr>
<tr>
<td>Normal 64-bit integers</td>
<td>0 to 18,446,729,055</td>
<td>64</td>
</tr>
<tr>
<td>Normal 128-bit integers</td>
<td>0 to 3.4028234663852886e+38</td>
<td>128</td>
</tr>
</tbody>
</table>

Footprint: Per-iteration I/O:

Footprint of a 2×2 blurring filter footprint analysis example:

- Looping over a 2-D data array
- A single accumulator repeatedly in an accumulator
- Compiler must dynamically manage the different layers of memory over the course of the program
- Need to provide “smart polymorphism” for different data types that may be used by the routine
- A script associated with each call takes the state of the machine at the time of the call and chooses the best match from the library of available routines
- Scripts can also generate some inline code to make the final routine decision at runtime, if necessary

New library support
Old-fashioned C libraries cannot provide enough flexibility and portability for PCA architectures
• A library call may map to several different library routines optimized in different ways
  – Different architectures
  – Different morph configurations
  – Different memory configurations or states
  – Different numbers or arrangements of tiles
  – We need to separate the call interface from the routine implementation in a simple and easily extendable manner!
• We need a meta-compilation layer to match calls to the appropriate routine at compile time
  – Need to provide “smart polymorphism” for different data types that may be used by the routine
  – A script associated with each call takes the state of the machine at the time of the call and chooses the best match from the library of available routines
  – Scripts can also generate some inline code to make the final routine decision at runtime, if necessary

Compiler memory analysis
Memory analysis and management in a PCA architecture with a complicated memory hierarchy can have a significant impact on performance with a streaming app
• Problem can be simplified into a 3-level hierarchy in most architectures
  – Compiler must dynamically manage the different layers of memory over the course of the program
  – This differs from conventional architectures, which normally have no compiler-controlled local memory (caches are implicitly controlled)

<table>
<thead>
<tr>
<th>Type</th>
<th>Range</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal 16-bit integers</td>
<td>0 to 65,535</td>
<td>16</td>
</tr>
<tr>
<td>Normal 32-bit integers</td>
<td>0 to 4,294,967</td>
<td>32</td>
</tr>
<tr>
<td>Normal 64-bit integers</td>
<td>0 to 18,446,729,055</td>
<td>64</td>
</tr>
<tr>
<td>Normal 128-bit integers</td>
<td>0 to 3.4028234663852886e+38</td>
<td>128</td>
</tr>
</tbody>
</table>

Footprint: Per-iteration I/O:

Footprint of a 2×2 blurring filter footprint analysis example:

- Looping over a 2-D data array
- A single accumulator repeatedly in an accumulator
- Compiler must dynamically manage the different layers of memory over the course of the program
- Need to provide “smart polymorphism” for different data types that may be used by the routine
- A script associated with each call takes the state of the machine at the time of the call and chooses the best match from the library of available routines
- Scripts can also generate some inline code to make the final routine decision at runtime, if necessary

New library support
Old-fashioned C libraries cannot provide enough flexibility and portability for PCA architectures
• A library call may map to several different library routines optimized in different ways
  – Different architectures
  – Different morph configurations
  – Different memory configurations or states
  – Different numbers or arrangements of tiles
  – We need to separate the call interface from the routine implementation in a simple and easily extendable manner!
• We need a meta-compilation layer to match calls to the appropriate routine at compile time
  – Need to provide “smart polymorphism” for different data types that may be used by the routine
  – A script associated with each call takes the state of the machine at the time of the call and chooses the best match from the library of available routines
  – Scripts can also generate some inline code to make the final routine decision at runtime, if necessary

Compiler memory analysis
Memory analysis and management in a PCA architecture with a complicated memory hierarchy can have a significant impact on performance with a streaming app
• Problem can be simplified into a 3-level hierarchy in most architectures
  – Compiler must dynamically manage the different layers of memory over the course of the program
  – This differs from conventional architectures, which normally have no compiler-controlled local memory (caches are implicitly controlled)

<table>
<thead>
<tr>
<th>Type</th>
<th>Range</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal 16-bit integers</td>
<td>0 to 65,535</td>
<td>16</td>
</tr>
<tr>
<td>Normal 32-bit integers</td>
<td>0 to 4,294,967</td>
<td>32</td>
</tr>
<tr>
<td>Normal 64-bit integers</td>
<td>0 to 18,446,729,055</td>
<td>64</td>
</tr>
<tr>
<td>Normal 128-bit integers</td>
<td>0 to 3.4028234663852886e+38</td>
<td>128</td>
</tr>
</tbody>
</table>

Footprint: Per-iteration I/O:

Footprint of a 2×2 blurring filter footprint analysis example:

- Looping over a 2-D data array
- A single accumulator repeatedly in an accumulator
- Compiler must dynamically manage the different layers of memory over the course of the program
- Need to provide “smart polymorphism” for different data types that may be used by the routine
- A script associated with each call takes the state of the machine at the time of the call and chooses the best match from the library of available routines
- Scripts can also generate some inline code to make the final routine decision at runtime, if necessary

New library support
Old-fashioned C libraries cannot provide enough flexibility and portability for PCA architectures
• A library call may map to several different library routines optimized in different ways
  – Different architectures
  – Different morph configurations
  – Different memory configurations or states
  – Different numbers or arrangements of tiles
  – We need to separate the call interface from the routine implementation in a simple and easily extendable manner!
• We need a meta-compilation layer to match calls to the appropriate routine at compile time
  – Need to provide “smart polymorphism” for different data types that may be used by the routine
  – A script associated with each call takes the state of the machine at the time of the call and chooses the best match from the library of available routines
  – Scripts can also generate some inline code to make the final routine decision at runtime, if necessary

Compiler memory analysis
Memory analysis and management in a PCA architecture with a complicated memory hierarchy can have a significant impact on performance with a streaming app
• Problem can be simplified into a 3-level hierarchy in most architectures
  – Compiler must dynamically manage the different layers of memory over the course of the program
  – This differs from conventional architectures, which normally have no compiler-controlled local memory (caches are implicitly controlled)