CellBE Programming Tips & Techniques

Cell Programming Workshop
Cell/Quasar Ecosystem & Solutions Enablement

Class Objectives – Things you will learn

- **Key programming techniques to exploit cell hardware organization and language features for**
  - SPE
  - SIMD

Class Agenda

- SPU Programming Tips
  - Level of Programming (Assembler, Intrinsics, Auto-Vectorization)
  - Overlap DMA with computation (double, multiple buffering)
  - Dual Issue rate (Instruction Scheduling)
  - Design for limited local store
  - Branch hints or elimination
  - Loop unrolling and pipelining
  - Integer multiplies (avoid 32-bit integer multiplies)
  - Avoid scalar code
  - Choose the right SIMD strategy
  - Load / Store only by quadword
- SIMD Programming Tips

SPU Programming Tips

SPU Programming Tips

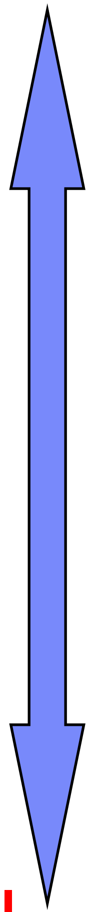
- Level of Programming (Assembler, Intrinsics, Auto-Vectorization)
- Overlap DMA with computation (double, multiple buffering)
- Dual Issue rate (Instruction Scheduling)
- Design for limited local store
- Branch hints or elimination
- Loop unrolling and pipelining
- Integer multiplies (avoid 32-bit integer multiplies)
- Avoid scalar code
- Choose the right SIMD strategy
- Load / Store only by quadword

Programming Levels on Cell BE

Trade-Off

Performance vs. Effort

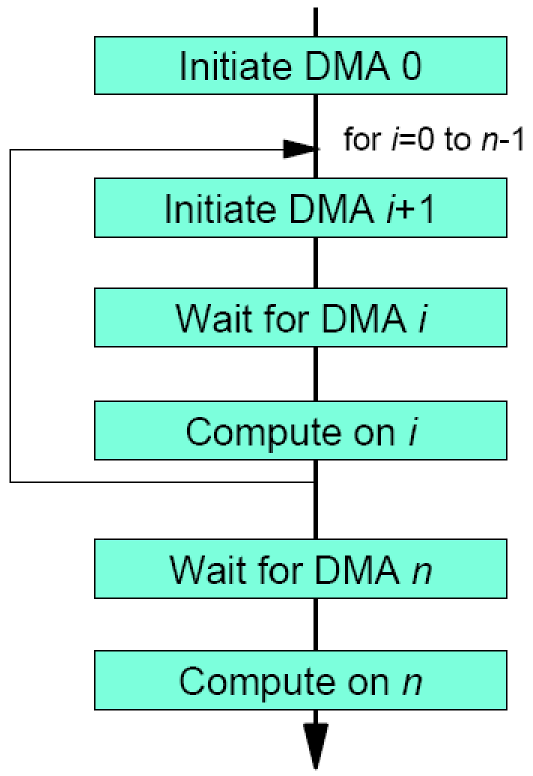
- **Expert level**
  – Assembler, high performance, high efforts
- **More ease of programming**
  – C compiler, vector data types, intrinsics, compiler schedules instructions + allocates registers
- **Auto-SIMDization**
  – for scalar loops, user should support by alignment directives, compiler provides feedback about SIMDization
- **Highest degree of ease of use**
  – user-guided parallelization necessary, Cell BE looks like a single processor



Requirements for Compiler increasing with each level

Overlap DMA with computation

- **Double or multi-buffer code or (typically) data**
- **Example for double buffering n+1 data blocks:**
  - Use multiple buffers in local store
  - Use unique DMA tag ID for each buffer
  - Use fence commands to order DMAs within a tag group
  - Use barrier commands to order DMAs within a queue



Start DMAs from SPU

- **Use SPE-initiated DMA transfers rather than PPE-initiated DMA transfers, because**
  - there are more SPEs than the one PPE
  - the PPE can enqueue only eight DMA requests whereas each SPE can enqueue 16

DMA Transfers and LS Accesses

- When using DMA buffers, declare the DMA buffers as volatile to ensure that buffers are not accessed by SPU load or store instructions until after DMA transfers have completed
  - Channel commands are ordered with respect to volatile-memory accesses. The DMA commands specify LS addresses as volatile, void pointers. By declaring all DMA buffers as volatile, it forces all accesses to these buffers to be performed (that is, they cannot be cached in a register) and ordered.
- When coding DMA transfers, exploit DMA list transfers whenever possible

Instruction Scheduling

- Choose intrinsics/instructions to maximize dual issue rates or reduce latencies (fine tuning)

Pipe 0 Instructions	length	stall
Single precision floating-point ops	6	0
Integer multiplies, convert between float/int, interpolate	7	0
Immediate loads, logical ops, integer add/subtract, sign extend, count leading zeros, select bits, carry/borrow generate	2	0
Double precision floating-point ops	7	6
Element rotates and shift	4	0
Byte ops (count ones, abs difference, averge, sum)	4	0
Pipe 1 Instructions	**length**	**stall**
Shuffle bytes, quadword rotates	4	0
Load/store, branch hints	6	0
Branch	4	0
Channel, move to/from spr	6	0

- Dual issue will occur if:
  - pipe 0 instruction even addressed, pipe 1 instruction odd address
  - no dependencies (operands are available)
- Code generators use nops (nop, lnop) to align instructions for dual issue

Design for Limited Local Store

- **The Local Store holds up to 256 KB for**
  - the program, stack, local data structures, and DMA buffers.
- **Most performance optimizations put pressure on local store (e.g. multiple DMA buffers)**
- **Use plug-ins (runtime download program kernels) to build complex function servers in the LS.**

Branch Optimizations

- **SPE**
  - Heavily pipelined → high penalty for branch misses (18 cycles)
  - Hardware policy: assume all branches are not taken
- **Advantage**
  - Reduced hardware complexity
  - Faster clock cycles
  - Increased predictability
- **Solution approaches**
  - If-conversions: compare and select operations
  - Predications/code re-org: compiler analysis, user directives
  - Branch hint instruction (hbr, 11 cycles before branch)

Branches

- Eliminate non-predicted branches
  - use feedback directed optimization
  - use __builtin_expect when programmer can explicitly direct branch prediction
    - ex:
```
if (a > b)          c += 1
else            c = a+b

if (__builtin_expect(a>b, 0))   c += 1        // predict a is not > b
else                            c = a+b
```
  - utilitze the select bits (spu_sel) instruction.
    - ex:
```
if (a > b)          c += 1
else            c = a+b

select = spu_cmpgt(a, b);
c1 = spu_add(c, 1);
ab = spu_add(a, b);
c   = spu_sel(ab, c1, select);
```

Loop Unrolling

- Unroll loops
  - to reduce dependencies
  - increase dual-issue rates
- This exploits the large SPU register file.
- Compiler auto-unrolling is not perfect, but pretty good.

Loop Unrolling - Examples

```
j=N;
For(i=1, i<N, i++) {
   a[i] = (b[i] + b[j]) / 2;
   j = i;
}
```

$\Rightarrow$

```
a[1] = (b[1] + b[N]) / 2;
For(i=2, i<N, i++) {
   a[i] = (b[i] + b[i-1]) / 2;
}
```

```
For(i=1, i<100, i++) {
   a[i] = b[i+2] * c[i-1];
}
```

$\Rightarrow$

```
For(i=1, i<99, i+=2) {
   a[i] = b[i+2] * c[i-1];
   a[i+1] = b[i+3] * c[i];
}
```

SPU

- Unroll loops to reduce dependencies and increase dual issue rates.
  - exploits large SPU register file
  - Example - xformlight workload (each loop iteration processes 4 vertices)

SW unroll factor	normalized performance	CPI	dual issue	dependency stalls	regs	text size
1 (none)	1.00	1.35	3.3%	27.2%	78	768
2	1.52	0.91	19.8%	5.9%	103	1344
4	1.73	0.76	34.3%	0.9%	128	3076
8	1.66	0.67	35.8%	1.5%	128	5252

  - compiler auto-unrolling is not perfect, but doing pretty good.
  - Results using spuxlc unrolling (unroll_large):

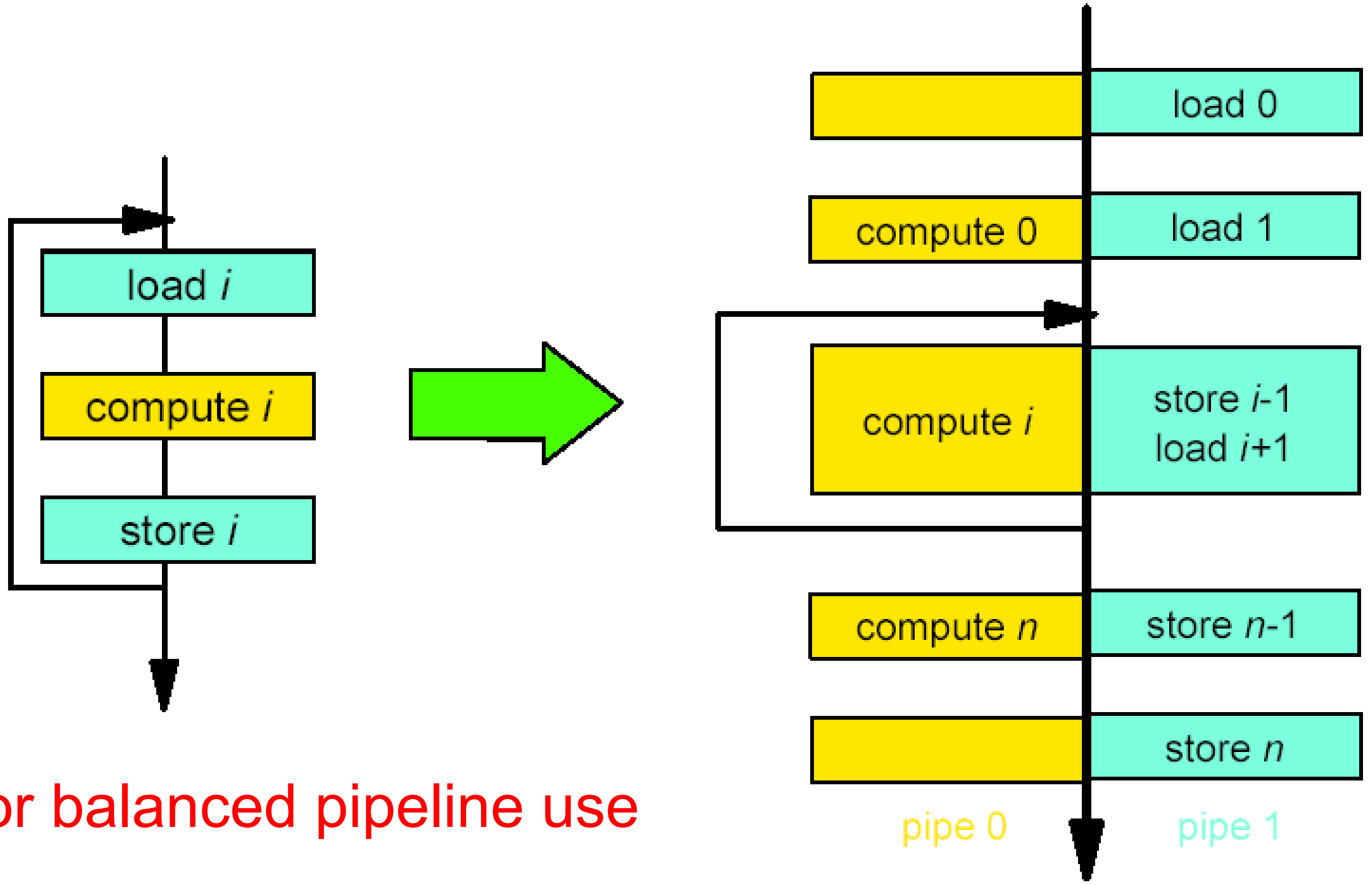
SW unroll factor	normalized performance	CPI	dual issue	dependency stalls	regs	text size
1 (none)	1.57	0.87	21.6%	1.9%	94	1472
2	1.52	0.91	18.4%	5.9%	103	1344
4	1.73	0.76	34.3%	0.9%	128	3076
8	1.66	0.67	35.8%	1.5%	128	5252

Function-Inlining

- Function-inlining eliminates the two branches associated with function-call linkage
- These include the *branch and set link* (such as brasl) for function-call entry, and the *branch indirect* (such as bi) for function-call return
- Notes: Over-aggressive use of inlining and loop unrolling can result in code that reduces the LS space available for data storage or, in the extreme case, is too large to fit in the LS.

SPU – Software Pipeline

- Software pipeline loops to improve dual issues rates.
- spuxlc does some pipelining.



Design for balanced pipeline use

Integer Multiplies

- **Avoid integer multiplies on operands greater than 16 bits**
  - SPU supports only a 16-bit x16-bit multiply
  - 32-bit multiply requires five instructions (three 16-bit multiplies and two adds)
- **Keep array elements sized to a power-of-2 to avoid multiplies when indexing.**
- **Cast operands to *unsigned short* prior to multiplying. Constants are of type *int* and also require casting.**
- **Use a macro to explicitly perform 16-bit multiplies. This can avoid inadvertent introduction of signed extends and masks due to casting.**

```
#define MULTIPLY(a, b)\
   (spu_extract(spu_mulo((vector unsigned short)spu_promote(a,0),\
   (vector unsigned short)spu_promote(b, 0)),0))
```

Avoid Scalar Code

- Scalar load/store are slow with long latency
  - SPU only supports quadword loads and stores
  - Ex:

```
void add1(int *p) {
        *p += 1;
}

add1:   lqd     $4, 0(p)      # load the qword pointed to by p
        rotqby  $5, $4, p     # move *p to element 0 of reg 5
        ai      $5, $5, 1     # add 1
        cwd     $6, 0(ptr)    # generate a shuffle pattern to insert *p+1 into qword
        shufb   $4, $5, $3    # insert scalar into qword
        stqd    $4, 0(p)      # save qword with updated scalar pointed to by p
```

  - Strategies:
    - consider making scalars qword integer vectors
    - load or store scalar arrays as quadwords and perform your own extraction and insertion to eliminate load/store instructions.
  - SDK example is RC4 encryption - scalar, non-parallelizable algorithm

	instructions	cycles	CPI	speedup
scalar	540120	723245	1.34	-
optimized to eliminate scalar overhead	265794	388457	1.46	1.86

Promoting Scalar Data Types to Vector Data Types

- Use spu_promote and spu_extract to efficiently promote scalars to vectors, or vectors to scalars.

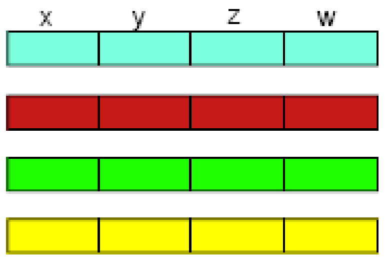
Instruction	Description
d = spu_promote(a, element)	Promote a scalar to a vector.
d = spu_extract(a, element)	Extract a vector element from its vector.

Choose an SIMD strategy appropriate for your algorithm

- Evaluate array-of-structure (AOS) organization
  - For graphics vertices, this organization (also called or vector-across) can have more-efficient code size and simpler DMA needs,
  - but less-efficient computation unless the code is unrolled.
- Evaluate structure-of-arrays (SOA) organization.
  - For graphics vertices, this organization (also called parallel-array) can be easier to SIMDize,
  - but the data must be maintained in separate arrays or the SPU must shuffle AOS data into an SOA form.

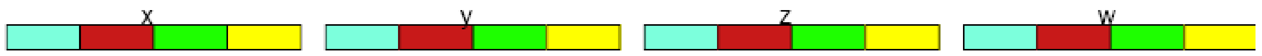
Choose SIMD strategy appropriate for algorithm

- **vec-across**
  - More efficient code size
  - Typically less efficient code/computation unless code is unrolled
  - Typically simpler DMA needs



4 independent 4-component vectors (cyan, red, green and yellow)

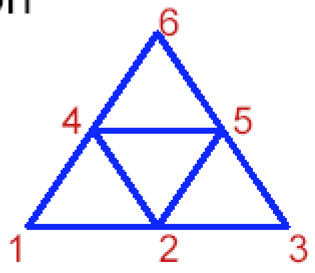
- **parallel-array**



  - Easy to SIMD – program as if scalar, operating on 4 independent objects at a time
  - Data must be maintained in separate arrays or SPU must shuffle vec-across data into a parallel array form
- **Consider unrolling affects when picking SIMD strategy**

SIMD Example

- SIMD example - point-normal triangle subdivision
  - problem: compute subdivided vertices for *n* independent triangles
  - solution 1: vec-across
    - evaluate vertices one at a time, unroll in improve performance
  - solution 2: parallel array
    - evaluate 4 subdivision vertices at a time for a single triangle
  - solution 3: parallel array
    - evaluate 1 subdivision point at a time on 4 independent triangles



solution	normalized performance	CPI	dual issue	dependency stalls	regs	text size
1	1.0	1.10	9.1%	14.1%	72	1472
1 (unrolled by 2)	1.26	1.04	12.6%	13.6%	112	2496
1 (unrolled by 4)	1.54	0.90	18.5%	5.8%	127	4480
2	1.34	0.96	11.9%	2.6%	107	1856
3	1.70	0.99	13.6%	4.7%	113	512

Load / Store by Quadword

- Scalar loads and stores are slow, with long latency.
- SPUs only support quadword loads and stores.
- Consider making scalars into quadword integer vectors.
- Load or store scalar arrays as quadwords, and perform your own extraction and insertion to eliminate load and store instructions.

SIMD Programming Tips

Reference: Dorit Nuzman, IBM Research

SIMD tips

- **Exploit SIMD**
  - Let compilers auto-SIMD
    - Language and algorithms make this challenging
  - Write assembly
    - Poor productivity
    - Not recommended
  - Programmer specified using generic Intrinsics
    - Inline assembly with function call syntax
    - Provide explicit control of the instructions used
    - Eliminates optimizations that compilers are good at.
      - Register coloring
      - Instruction scheduling
      - Data loads and stores
      - Looping and branching
      - Vector literal construction

Use gcc autovectorization

- **gcc –O2 –ftree-vectorize myloop.c**
- **for the PPU may need to add –maltivec**
- **-ftree-vectorizer-verbose=[X]**
  - dumps information on which loops got vectorized, and which didn't and why
  - X=1 least information, X=6 all information
  - dumped to stderr unless following flag is used:
- **-fdump-tree-vect**
  - dumps information into myloop.c.t##.vect
- **-fdump-tree-vect-details**
  - is equivalent to '-fdump-tree-vect -ftree-vectorizer-verbose=6'

Autovectorization programming hints

- **Don't unroll the loop**
- **Use countable loops, with no side-effects**
  - No function-calls in the loop (distribute into a separate loop)
  - No 'break'/'continue'
- **Avoid aliasing problems**
  - Use __restrict__ qualified pointers
- **Keep the memory access-pattern simple**
  - Don't use array of structures, e.g.:

    ```
    for (i=0; i<N; i++)
      a[i].s = x;
    ```
  - Use constant increment. i.e., don't use the following:

    ```
    for (i=0; i<N; i+=incr)
      a[i] = x;
    ```
- **Alignment**
  - Use alignment attributes
  - If have more than a single misaligned store – distribute into a separate loop (currently the vectorizer peels the loop to align a misaligned store).

Compiler Optimization Tips - XLC

Tips for getting the most out of –O2 and –O3

- If possible, test and debug your code without optimization before using -O2
- Ensure that your code is standard-compliant. Optimizers are the ultimate conformance test!
- In Fortran code, ensure that subroutine parameters comply with aliasing rules
- In C code, ensure that pointer use follows type restrictions (generic pointers should be char* or void*)
- Ensure all shared variables and pointers to same are marked volatile
- Compile as much of your code as possible with -O2.
- If you encounter problems with -O2, consider using -qalias=noansi or -qalias=nostd rather than turning off optimization.
- Next, use -O3 on as much code as possible.
- If you encounter problems or performance degradations, consider using –qstrict, -qcompact, or -qnohot along with -O3 where necessary.
- If you still have problems with -O3, switch to -O2 for a subset of files/subroutines but consider using -qmaxmem=-1 and/or -qnostrict.

Tips for getting the most out of -qhot

- Try using -qhot along with -O2 or -O3 for all of your code. It is designed to have neutral effect when no opportunities exist.
- If you encounter unacceptably long compile times (this can happen with complex loop nests) or if your performance degrades with the use of -qhot, try using -qhot=novector, or -qstrict or -qcompact along with -qhot.
- If necessary, deactivate -qhot selectively, allowing it to improve some of your code.
- Read the transformation report generated using –qreport. If your hot loops are not transformed as you expect, try using assertive directives such as INDEPENDENT or CNCALL or prescriptive directives such as UNROLL or PREFETCH.

Tips for getting the most out of -qipa

- When specifying optimization options in a makefile, remember to use the compiler driver (cc, xlf, etc) to link and repeat all options on the link step:
  - LD = xlf
  - OPT = -O3 -qipa
  - FFLAGS=...$(OPT)...
  - LDFLAGS=...$(OPT)...
- -qipa works when building executables or shared objects but always compile 'main' and exported functions with -qipa.
- It is not necessary to compile everything with -qipa but try to apply it to as much of your program as possible.

More -qipa tips

- When compiling and linking separately, use -qipa=noobject on the compile step for faster compilation.
- Ensure there is enough space in /tmp (at least 200MB) or use the TMPDIR variable to specify a different directory.
- The "level" suboption is a throttle. Try varying the "level" suboption if link time is too long. -qipa=level=0 can be very beneficial for little cost.
- Look at the generated code. If too few or too many functions are inlined, consider using -qipa=[no]inline

Coding choices that impact simdization

- How loops are organized
  - Loop must be countable, preferably with literal trip count
  - Only innermost loops are candidates for simdization, except when nested loops have a short literal iteration count
  - Loops with control flow are harder to simdize. Compiler tries to remove control flow, but not always successful
- How data is accessed and laid out in memory
  - Data accesses should preferably be stride-one
  - Layout the data to maximize aligned accesses
  - Prefer use of arrays to pointer arithmetic
- Dependences inherent to the algorithm
  - Loops with inherent data dependences are not simdizable
  - Avoid pointers; pointer aliasing may impede transformations

Assisting the compiler to perform auto-SIMD

- **Loop structure**
  - Inline function calls inside innermost loops
  - Automatically (-O5 more aggressive, use inline pragma/directives)
- **Data alignment**
  - Align data on 16-byte boundaries __attribute__((aligned(16))
  - Describe pointer alignment _alignx(16, pointer)
    - Can be placed anywhere in the code, preferably close to the loop
  - Use -O5 (enables inter-procedural alignment analysis)
- **Pointer aliasing**
  - Refine pointer aliasing #pragma disjoint(*p, *q) or restrict keyword
  - Use -O5 (enables interprocedural pointer analysis)

Other SIMD Tuning

- Loop unrolling can interact with simdization
  - Manually-unrolled loops are more difficult to simdize

- Tell compiler not to simdize a loop if not profitable
  - #pragma nosimd (right before the innermost loop)
  - Useful when loop bounds are small and unknown at compile time

System Level Practices

System Level Practices

- **Partitioning and Work allocation strategies**
- **Algorithmic**
  - Possible self regulated work allocation
- **Work queues**
  - Single – SPE arbitrated
    - Works well when the work task are computationally significant and variable.
  - Multiple – PPE distributed
    - Works well when time to complete the task is predictable.
- **Consider all domains in which to partition the problem.**
  - example: Video application
    - Space – partition scan lines or image regions to a different SPE
    - Time – partition each frame to a different SPE

System Level Practices

- **Offload as much work onto the SPEs as possible**
  - Use the PPE as the control plane processor
    - Orchestrate and schedule the SPEs
    - Assist SPEs with exceptional events
  - Use SPEs as data plane processors

- **Accommodate potential data type differences**
  - SPE is ILP32 (32-bit integers, longs, and pointers) w/ 32 or 64 EAs (effective addresses)
  - PPE is either ILP32 or LP64 (64-bit longs and pointers)

Miscellaneous Programming Techniques

- **Use the software managed cache to handle applications with non-predicable data access patterns.**
- **SDK 3.0 provides an example of SW cache implementation**
  - Source - /opt/cell/sdk/src/examples/cache/
    - cache/sort : quick sort using sw managed cache
    - cache/cache-cp : cache based file copy
    - julia : texture mapping using sw managed cache.
- **Features:**
  - Support for multiple simultaneous caches
  - Attributes: selectable cache size, associatively, read-only or read-write.
  - Supports statistical gathering
  - Functions: read, read 4, write, flush, print stats, touch, wait, lock, and unlock

Special Notices -- Trademarks

Special Notices (Cont.) -- Trademarks

Special Notices - Copyrights