



# Cell DLA Algorithms

Fred Gustavson  
assisted by Lars Karlsson  
F12: Design & Analysis of Algorithms for  
HPC Systems,  
Umeå University, Sweden, May 9, 2008

1



## Summary of these talks

- Part One: Limitations of the PlayStation 3 for High Performance Cluster Computing & LAWN 189
  - Briefly describe Cell Architecture
  - Matrix multiply on Cell
  - Matrix multiply kernel routine on the Cell for SGEMM
- Part Two
  - Cholesky Factorization on Cell: LAWN # 184
- Two parts: recent work of Jack Dongarra's team
  - relate these works to work of the Umea team

2

# Matrix Multiply on Cell : Part One

Fred Gustavson

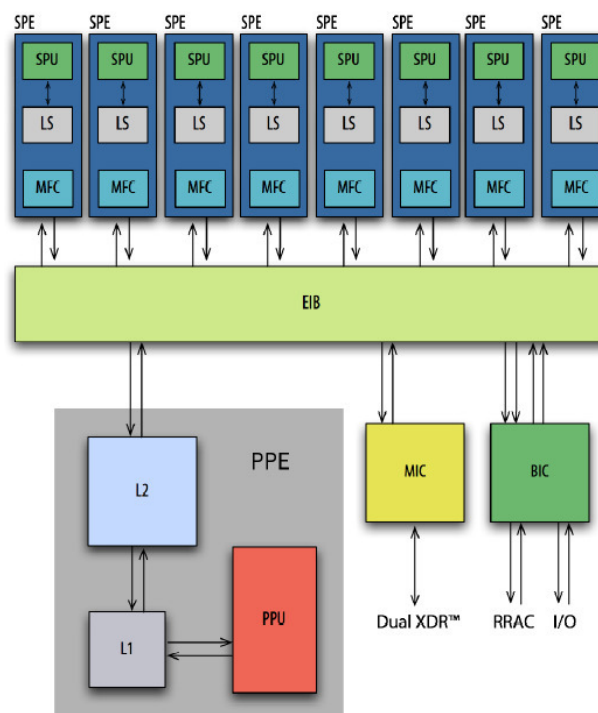
assisted by Lars Karlsson

F12: Design & Analysis of Algorithms  
for HPC Systems,

Umea University, Sweden, May 9, 2008

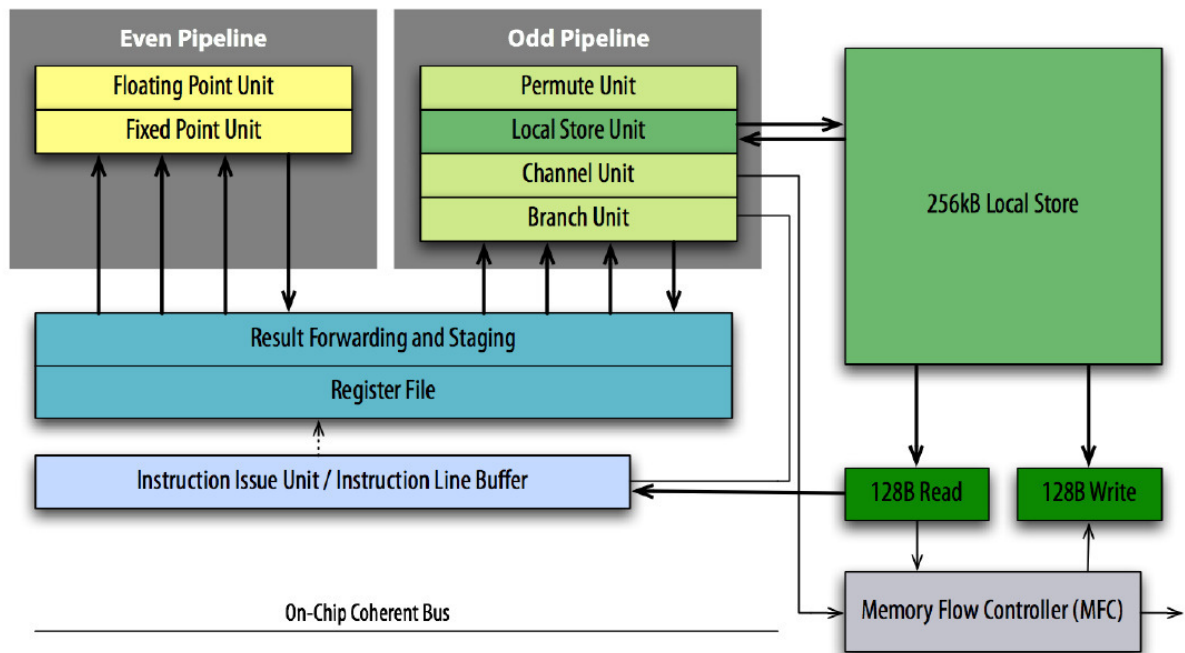
3

## Cell Architecture



4

# SPE Architecture



# Programming Rules for Cell

- Vectorize
  - SPEs are vector units, scalar operations "promoted" to vector operations
- Keep data aligned
  - Optimizes memory transfers, important both for main memory and local store accesses
- Implement double-buffering
  - Hiding the reads and writes with concurrent computation crucial, double-buffering one way to do that (compute with one buffer, communicate with the other)
- Improve data reuse
  - To reduce memory traffic, data which is brought into the local store should be reused aggressively
- Unroll loops
  - High register count on the SPEs combined with primitive branch predictions makes loop unrolling an attractive optimization



# Parallel Matrix-Matrix Multiply-1

- Cannon, 1967, Summa, 1993, 1997  
Pumma, 1994, ScaLapack, 1995 are examples that are still used today
- Rely on serial matrix multiply done on each node
  - Gemm operation done on sub-matrices
- Good benchmark to evaluate a || machine

7



# Parallel Matrix-Matrix Multiply - 2

- Matching Algorithm to Architecture is key
  - Umea concurs
- Summa algorithm was chosen
- Natural choice is a block layout
  - works equally well for a block-cyclic layout
- MPI and local computations easily overlapped
  - Umea comment: not necessarily true
  - Need non-blocking communication
    - Not part of every MPI implementation

8



## Parallel Matrix-Matrix Multiply - 3

- Summa algorithm of 1993 overlapped computation with communication
- Obtained perfect speed-up on a 512 node Intel Touchtone Delta Machine
- showed how to hide all the communication cost in the computation
- Feature: peak performance for large matrices even when communication network is slow.

9



## Summa on Cell - 1

- Atomic unit is a SB of order 64
- Each cell processor get rectangular matrix of SB's; Fig 4, p 10
- See Algorithm 1, p 10
  - $C = \text{sum } (i=1:n) (A_{*i} * B_{i*})$
  - details given on next slide

10



# The SUMMA Algorithm

---

## Algorithm 1 SUMMA

---

```
1: for  $i = 1$  to  $n/nb$  do
2:   if I own  $A_{*i}$  then
3:     Copy  $A_{*i}$  in  $buf1$ 
4:     Bcast  $buf1$  to my proc row
5:   end if
6:   if I own  $B_{i*}$  then
7:     Copy  $b_{i*}$  in  $buf2$ 
8:     Bcast  $buf2$  to my proc column
9:   end if
10:   $C = C + buf1 * buf2$ 
11: end for
```

---

11



## Summa on Cell - 2

- performance modeled on p 12
- overlap communication with computation
- uses double buffering
- time =  $\max(t_{\text{comm}}, t_{\text{comp}})$  ; equation (8)

12



## Summa on Cell - 3

- performance model very accurate; p 12-14
- runs for  $t_{\text{comp}} < t_{\text{comm}}$  &  $t_{\text{comp}} > t_{\text{comm}}$ 
  - six SPE's per node; fig. 7 & fig. 8 right
  - one SPE per node; fig. 8 left
- see page 15 top for explanations

13



## Summa on Cell - 4

- high performance at low cost
- serious limitations
  - true for other processors
- slow main memory
- network interconnect
  - out of balance with peak speed of each node
- small main memory & slow d.p. arithmetic
- programming paradigm
  - simple & directly controllable architecture

14

# Fast and Small Short SIMD Matrix Multiplication Kernels for the Synergistic Processing Element of the Cell Processor

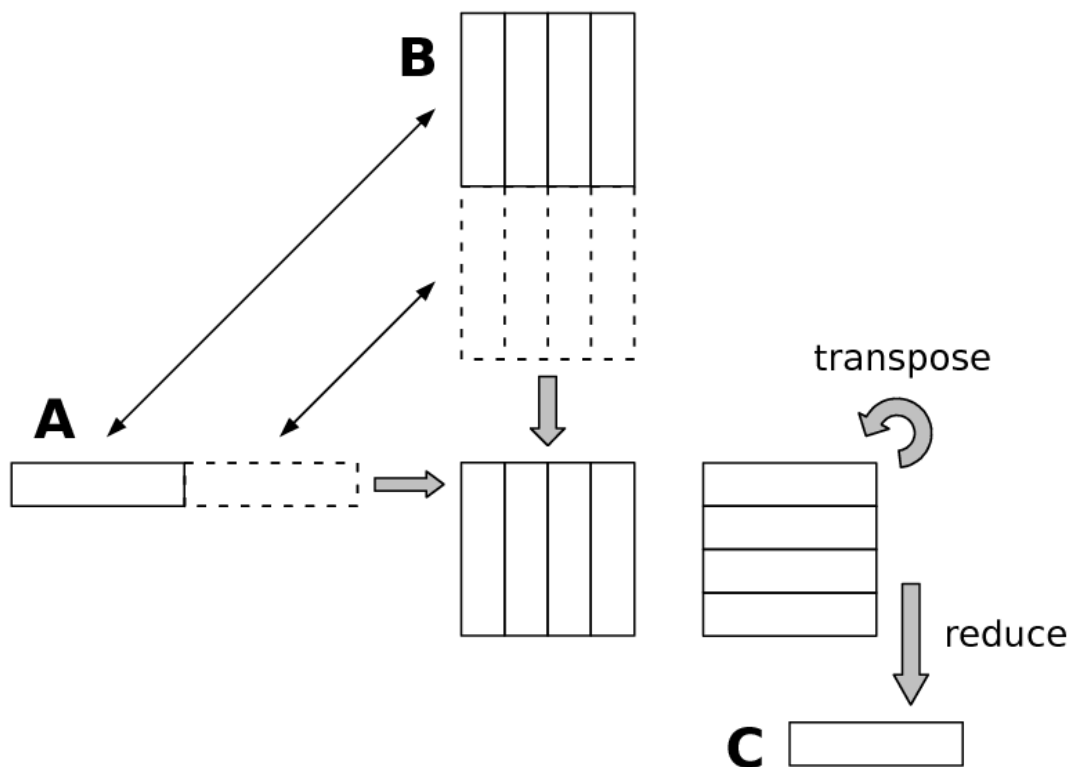
## ■ $C = C - A * B$

- For  $m=n=k=64$  gets 99.80% of peak
- Uses 5.9 KB of storage code for code and auxiliary data structures; Fig. 3, p. 8

## ■ $C = C - A * B^T$

- See page 7 :  $64 / (64 + 4) \times 25.6 = 24.09$  or 94%
- Fig. 2, page 6 of above paper

15



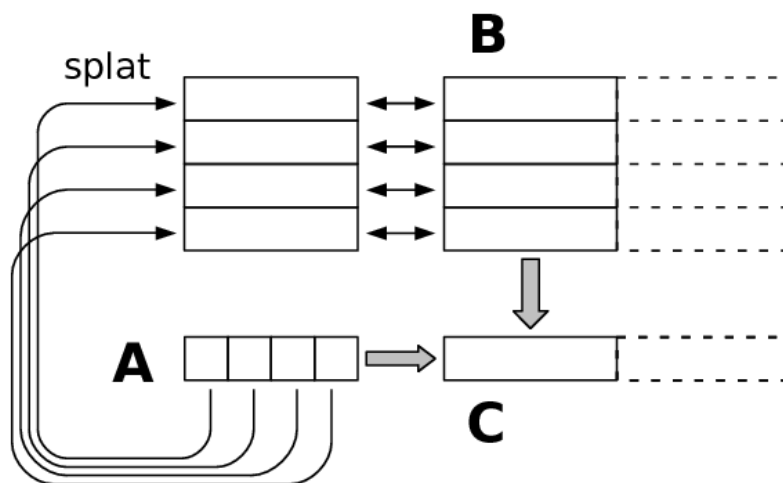
16



# Dot product and partial sums

- Latency problem  $c = \sum_{i=0:n-1} (x(i) * y(i))$
- Overcome by using partial sums
- Example  $ps=4$ ;  $ps(0:3)$ ,  $ps(j) = \sum_{i=0:n/4-1} (x(4*i+j) * y(4*i+j))$ 
  - Get  $ps$  partial sums; add these  $ps$  # at the end
- This idea used on the previous slide in a different context.

17

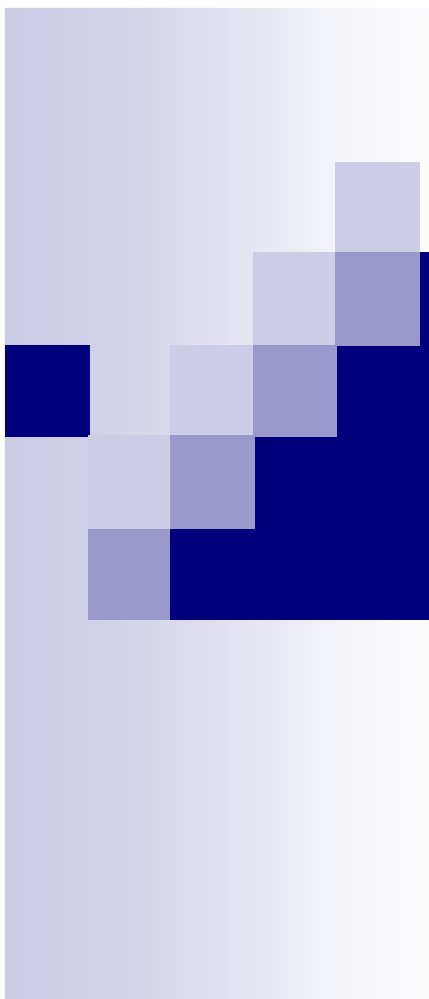


18



# End of Matrix Multiply on Cell

19



## Cholesky Factorization on Cell : Part Two

Fred Gustavson

assisted by Lars Karlsson

F12: Design & Analysis of Algorithms  
for HPC Systems,

Umea University, Sweden, May 9, 2008

20



# Cholesky Factorization on Cell

- Review paper by Kurzak, Buttari and Dongarra : Lapack Working Note 184
- Improvements and relation to Umea's research given

21



## Abstract

- Cell is a pioneering architecture
- effectively exploit single & double precision on Cell via iterative refinement
- efficient parallelization of short SIMD
  - mixed precision algorithm
  - exploits fine grain parallelization
- very good performance

22



# Motivation

- goal is speed: SPE is 4 x faster than PPE
  - data movement halved
  - twice the FMA power
- Cell has eight SPE and only one PPE
- much larger register file
- Many other vendors have multi-core
- Shift is away from ILP to TL ||-ism

23



# Motivation continued

- need new paradigms
  - departure from the BLAS ||-ism concept
  - more flexible ways of scheduling work
- above also says apply the A & A approach
  - NDS and kernel level BLAS
  - Umea work since 1998 suffices on Cell

24



## Related Work

- Pioneering work of Wilkinson, Moler, Stewart & later Higham & later still by Langou, et.al. on iterative refinement
- paper by authors on  $LU = PA$
- No mention of other work
  - Umea research appears quite relevant

25



## Algorithm

- See next slide & pages 3 & 4 of paper
- SP Left Looking (LL) Cholesky factor
  - gave better || code than traditional Right Looking (RL) Cholesky
  - RL Cholesky has more stores in outer loops
- Original paper LL and RL was Dongarra, Gustavson and Karp: SIAM review, 1984
  - Recursion paper; Gustavson IBM JR&D 1997

26

# Algorithm 1

---

**Algorithm 1** Solution of a symmetric positive definite system of linear equations using mixed-precision iterative refinement based on Cholesky factorization.

---

```
1:  $A_{(32)}, b_{(32)} \leftarrow A, b$ 
2:  $L_{(32)}, L_{(32)}^T \leftarrow \text{SPOTRF}^a(A_{(32)})$ 
3:  $x_{(32)}^{(1)} \leftarrow \text{SPOTRS}^b(L_{(32)}, L_{(32)}^T, b_{(32)})$ 
4:  $x^{(1)} \leftarrow x_{(32)}^{(1)}$ 
5: repeat
6:    $r^{(i)} \leftarrow b - Ax^{(i)}$ 
7:    $r_{(32)}^{(i)} \leftarrow r^{(i)}$ 
8:    $z_{(32)}^{(i)} \leftarrow \text{SPOTRS}^b(L_{(32)}, L_{(32)}^T, r_{(32)}^{(i)})$ 
9:    $z^{(i)} \leftarrow z_{(32)}^{(i)}$ 
10:   $x^{(i+1)} \leftarrow x^{(i)} + z^{(i)}$ 
11: until  $x^{(i)}$  is accurate enough
```

<sup>a</sup>LAPACK name for Cholesky factorization

<sup>b</sup>LAPACK name for symmetric back solve

64-bit representation is used in all cases where 32-bit representation is not indicated by a subscript.

---

27

## Page 3

- an  $LL^T$  factorization
  - same as  $U^T U$  factorization in Fortran
- DSPOSV
- Crout L formula shows that a left looking (LL) algorithm has less stores
  - Saves half the bandwidth in the || case

28



# Implementation

- Essential Hardware features already covered
- See Part One of these lectures

29



## Section 4.2

- the next slide gives a picture of the four BLAS kernel routines that work on Square Block Packed (SBP) data format
  - these fundamental ideas introduced by Umea research several years ago starting in 1997
- use of tiling is given as a key to performance
  - claim not good unless tiles are in SBF

30

# Figure 1: page 4

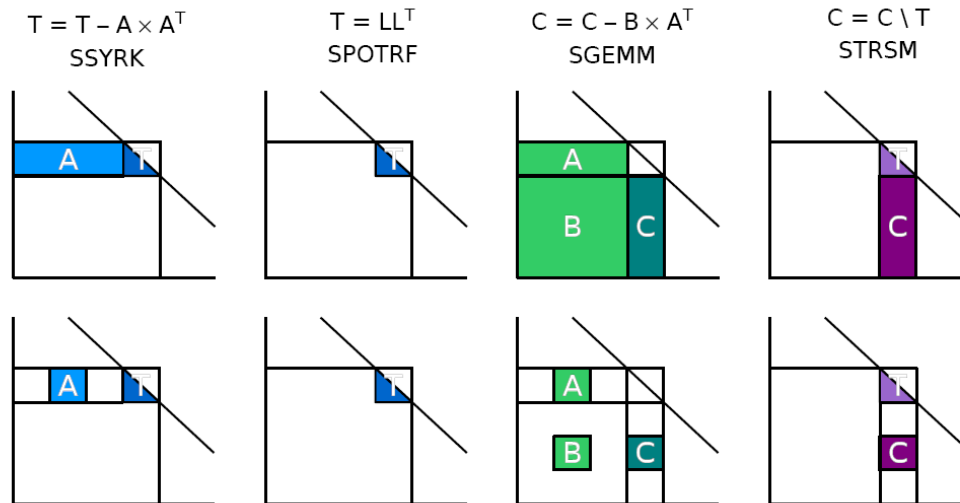


Figure 1: Top - steps of left-looking Cholesky factorization. Bottom - tiling of operations.

31

## Remarks

- In 1999 and many times later a paper on the A & A approach featuring NDS took the same approach for Cholesky factor on starting on the IBM Power 3 platform
- Tile kernels here were called BLAS kernels there

32






## Section 4.2.1

- new code is LL Cholesky on sub-matrices of order 64.
- MM is most important: SGEMM does  $O(n^3)$  part of the flops;  $n = N/NB$ 
  - see previous lecture for reason; point made by Umea research several years ago in concert with using NDS
- SSYRK is MM on a symmetric matrix so half the operations can be saved by exploiting symmetry

33



## Section 4.2.1 continued

- STRSM & SSYRK kernels consume  $O(n^2)$  flops of the total  $O(n^3)$  flops
- SPOTRF kernel consumes  $O(n)$  flops
  - first time JD is using level 3 for factorization
  - Umea has used factor kernels many times
- the four kernels use “register blocking”
  - see Para1998 Super Scalar BLAS paper

34



## 4.2.1 continued

- right & left looking first introduced in 1984 in SIAM review paper by Dongarra, Gustavson and Karp
- In IBM J R & D paper by Gustavson in 1997, recursive blocking introduced
  - Different than IJK orders of 1984 paper
  - Idea of NDS and BLAS kernels introduced

35



## Section 4.2.1 continued (p. 6)

- typo on page 6: 4 (not 16) 4-element as a Register Block (RB) has order 4
- Algorithm 2 uses a hardware feature of Cell; allows one to keep all RB in simple RM order
  - see MDC paper in Para 06
  - see steps 4 & 7 of Algorithm 2
  - Algorithm 2 might be simplified and run faster

36

## Four BLAS kernels; Table 1, p. 5

Operation	BLAS / LAPACK Call
$T \leftarrow T - A \times A^T$	<code>cblas_ssyrc(CblasRowMajor, CblasLower, CblasNoTrans, 64, 64, 1.0, A, 64, 1.0, T, 64);</code>
$C \leftarrow C - B \times A^T$	<code>cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasTrans, 64, 64, 64, 1.0, B, 64, A, 64, 1.0, C, 64);</code>
$B \leftarrow B \times T^{-T}$ ( $B = B/T^T$ ) <sup>a</sup>	<code>cblas_strsm(CblasRowMajor, CblasRight, CblasLower, CblasTrans, CblasNonUnit, 64, 64, 1.0, T, 64, B, 64);</code>
$T \leftarrow L \times L^T$	<code>lapack_spotrf(lapack_lower, 64, trans(T), 64, &amp;info);</code> <sup>b</sup>

<sup>a</sup>using MATLAB notation  
<sup>b</sup>using LAPACK C interface by R. Delmas and J. Langou,  
<http://icl.cs.utk.edu/~delmas/lapwrapc.html>

Table 1: Single precision Cholesky tile kernels.

37

## SSYRK kernel on page 5

---

**Algorithm 2** SSYRK tile kernel  $T \leftarrow T - A \times A^T$ .

---

```
1: for  $j = 0$  to  $15$  do
2:   for  $i = 0$  to  $j - 1$  do
3:     Compute block  $[j,i]$ 
4:     Permute/reduce block  $[j,i]$ 
5:   end for
6:   Compute block  $[j,j]$ 
7:   Permute/reduce block  $[j,j]$ 
8: end for
```

---

*block* is a  $4 \times 4$  submatrix of tile  $T$ .

---

38



# SSYRK kernel

- Use fusion
- Do extra ops on the diagonal block so all blocks are like gemm blocks
- interleave as in the gemm kernel

39



# SGEMM kernel on page 6

---

**Algorithm 3** SGEMM tile kernel  $C \leftarrow C - B \times A^T$

---

- 1: Compute block 0
- 2: **for**  $i = 1$  to 127 **do**
- 3:   Permute/reduce blk  $2i - 2$  & compute blk  $2i - 1$
- 4:   Permute/reduce blk  $2i - 1$  & compute blk  $2i$
- 5: **end for**
- 6: Permute/reduce blk 254 & compute blk 255
- 7: Permute/reduce blk 255

---

*blk* is a  $4 \times 4$  submatrix of tile  $C$ .

---

40



## Remarks on SGEMM kernel

- can overlap communication with computation
  - see Part One of this talk
- a hardware feature of Cell allows one to store a SB in RM order and permute it in the inner loop getting:
  - 64 by 64 matrix as 16 by 16 matrix of 4 by 4 RB's.

41



## STRSM kernel on page 6

---

**Algorithm 4** STRSM tile kernel  $B \leftarrow B \times T^{-T}$ .

---

```
1: for  $j = 0$  to  $15$  do
2:   for  $i = 0$  to  $j - 1$  do
3:     Apply block  $i$  towards block  $j$ 
4:   end for
5:   Solve block  $j$ 
6: end for
```

---

*block* is a  $64 \times 4$  submatrix of tile  $B$ .

---

42



## Remarks on STRSM kernel

- made kernel perfectly parallel
- introduced the need to transpose B matrices
- STRSM kernel lacks detail; step 3 and 5
- see some detail on page 6 of paper

43



## SPOTRF kernel of page 6

---

**Algorithm 5** SPOTRF tile kernel  $T \leftarrow L \times L^T$ .

---

```
1: for  $k = 0$  to  $15$  do
2:   for  $i = 0$  to  $k - 1$  do
3:     SSYRK (apply block  $[k,i]$  to block  $[k,k]$ )
4:   end for
5:   SPOTF2 (factorize block  $[k,k]$ )
6:   for  $j = k$  to  $15$  do
7:     for  $i = 0$  to  $k - 1$  do
8:       SGEMM (apply block  $[j,i]$  to block  $[j,k]$ )
9:     end for
10:  end for
11:  for  $j = k$  to  $15$  do
12:    STRSM (apply block  $[k,k]$  to block  $[j,k]$ )
13:  end for
14: end for
```

---

*block* is a  $4 \times 4$  submatrix of tile  $T$ .

---

44



## Remarks on SPOTRF

- Umea research has emphasized using level three factorization kernels
- paper claims the usual way is to use level two factorization kernels
- NDS and kernel routines go hand in hand
- this is the A & A principle
- see earlier lecture where this point is made for DLA

45



## Some Details on the Factor kernel

- need to factor a 4 by 4 block
  - recursion useful to see the processing
- scaling or trsm
  - store 4 by 4 diag. block in full format
  - a block of B is in row format
- $r4 = r4 * (\text{one}/u44)$
- $r3 -= r4 * u34; r2 -= r4 * u24; r1 -= r4 * u14$

46



## Factor kernel; trsm scaling

- $r3 = r3 * (one/u33)$
- $r2 -= r3 * u23; r1 -= r3 * u12$
- $r2 = r2 * (one/u22)$
- $r1 -= r2 * u12$
- $r1 = r1 * (one/u11)$

47



## Factor kernel; overall

- Do  $j = 0, 15$ 
  - syrk update  $a(j,j)$
  - factor  $a(j,j)$
  - do  $i = j + 1, 15$ 
    - gemm update  $a(i,j)$
    - scale  $a(i,j)$
  - enddo
- enddo

48



# Factor kernel; overall

- possibly modify gemm kernel of Table 1 and use it for the syrk and gemm updates of the previous slide

49

# Table 2 of page 7

Kernel	Source Code [LOC]	Compilation	Object Code [KB]	Execution Time [ $\mu$ s]	Execution Rate [Gflop/s]	Fraction of Peak [%]
SSYRK	160	spuxlc <sup>a</sup> -O3	4.7	13.23	20.12	78
SGEMM	330	spu-gcc <sup>b</sup> -Os	9.0	<b>22.78</b>	<b>23.01</b>	<b>90</b>
STRSM	310	spuxlc <sup>a</sup> -O3	8.2	16.47	15.91	62
SPOTRF	340	spu-gcc <sup>b</sup> -O3	4.1	15.16	5.84	23

<sup>a</sup>version 1.0 (SDK 1.1)  
<sup>b</sup>version 4.0.2 (toolchain 3.2 - SDK 1.1)

Table 2: Performance of single precision Cholesky factorization tile kernels.

50



## Remarks on Table 2

- SPOTRF kernel can be improved and kept simple
- see sketch of this on previous slides

51



## General Remarks on Kernels

- The authors seem to imply that the high performance obtained is remarkable for such small granularity. They attribute this occurrence to the Cell architecture and especially its register file and memory organization.
- The Umea group has observed the same results for other architectures and believe these results are due to NDS and use of kernel routines

52



## Parallelization: Section 4.2.2

- Loading three tiles to do SGEMM stresses the bandwidth of Cell to much.
  - calculation shows three blocks require peak BW
- Simple blocking as in the first column of page 8 will reduce the bandwidth by a factor of .5
- A 1-D row data layout is good for LL algorithms
- This necessitated some scheduling to improve load balancing
- See the next two figures for a picture description

53



## Reducing BW by about half

- in syrkh one SPE applies  $j - 1$  tiles left of  $T$ ; tile  $T$  is read and written only once
- in gemm one SPE reads  $j - 1$  tiles of  $A$  &  $B$  and applies them to  $C$ ;  $C$  is read once & then  $T$  can be read and applied before  $C$  is finally stored

54

# Bad Load Balance on Last rows

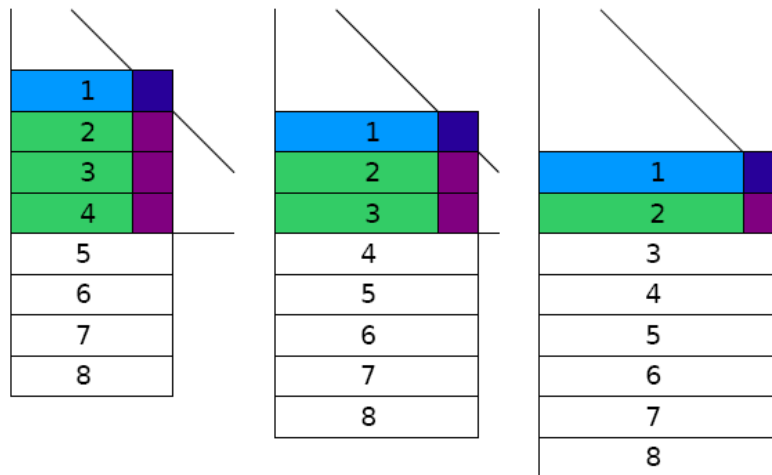


Figure 2: Load imbalance caused by 1D processing.

55

## A load balancing problem

- the last factorization stages of the LL code cannot be assigned
- for many processors and large memories a 2-d scheme would probably be required
- on page 9 in Section 4.2.3 a novel solution is proposed that reduces the present load unbalance to an acceptable level
  - see Gantt chart in Figure 4 for results

56

## Fig. 3, p. 8: Pipelining factorization

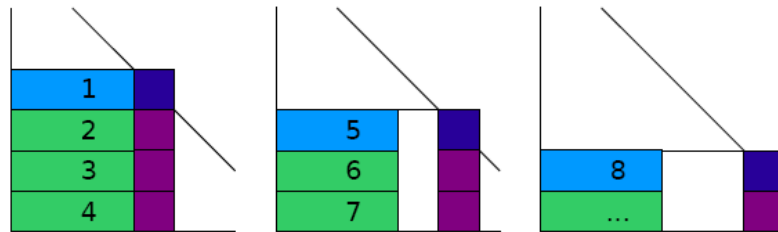


Figure 3: Pipelining of factorization steps.

57

## Fix: Schedule more work for each factorization step

- Schedule extra work to any idle processor during any factorization step from upcoming steps of the factorization.
- See the above Figure 3 of page 8
- Needs dependency tracking
  - 2-d tracking with tracking duplicated on each SPE for fast checking

58



## More on Scheduling

- Static schedule with cyclic distribution of work from the steps of the factorization
- Works well because SBF does away with non-deterministic phenomena like cache misses
  - use of SB format & alignment is crucial here
- see Gantt chart for a matrix of order 1024

59



## Quantitative details of Gantt Chart

- For  $N = 1024$  there are  $n = 16$  block steps
- i f t s g sum
- 1 1 0 0 0 1
- 2 2 1 1 0 4
- 3 3 3 3 1 10
- 4 4 6 6 4 20
- 5 5 10 10 10 35
- k k CK2 CK2 CK3 CK+2,3
- 16 16 120 120 560 816

60

# Gantt chart for A of order 1024

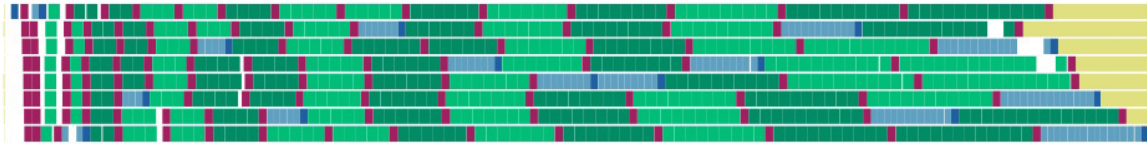


Figure 4: Execution chart of Cholesky factorization of a matrix of size  $1024 \times 1024$ . Color scheme follows the one from Figure 1. Different shades of green correspond to odd and even steps of the factorization.

61

## Section 4.2.3: Synchronization

- Dependencies or the critical path
  - SSYRK and SGEMM cannot use their A & B operands to update a C block unless they are factorized or scaled; i.e., completed
  - Any off diagonal block must be scaled to be completed; scaling requires that the associated diagonal block be completed
  - Any diagonal block must be factored to be completed

62



## Synchronization continued

- Introduce a progress table
  - standard idea
  - called a Directed Acyclic Graph (DAG);
- Duplicating the progress table on each SPE is a good idea
  - can be done via DMA at the byte level
  - small amount of traffic must be fast even though it covers up to some  $64^3$  operations

63



## Synchronization continued

- page 9 top of second column
  - Lars Karlsson's remark about MPI on slide 8
- second sentence; another usage of the A & A approach

64





## Section 4.2.4 Communication

- Double Buffering very important
  - eight buffers allow each operation to be buffered or pre-fetched; a use of A & A
  - see magnified Figure 5 of Figure 4 showing full overlap of comp. and comm.; four kernels allow this
  - blocks always read from memory to SPE or written from SPE to memory

65



## 4.2.4: More communication

- must do dependency checks to pre-fetch
  - if check fails just abort the pre-fetch.
  - on failure and before processing one must busy wait for a blocking send to complete
- Now see good overlap in Figure 5
  - 2<sup>nd</sup> row shows six syrks and 66 factor
  - 1<sup>st</sup> row, 2<sup>nd</sup> red is completion 5<sup>th</sup> trsm (10)

66

## 4.2.4: More communication

- Now see good overlap in Figure 5
  - 3<sup>rd</sup> row shows start of 6<sup>th</sup> trsm at 2<sup>nd</sup> red
  - 4<sup>th</sup> row shows seven syrks and 77 factor
  - 3<sup>rd</sup> row, 3<sup>rd</sup> red is completion 6<sup>th</sup> trsm (9)
  - dark greens in middle show several 6 gemms
  - light green in 8<sup>th</sup> row 5 gemms
  - light greens to right are starts of 7 gemms

67

## Figure 5

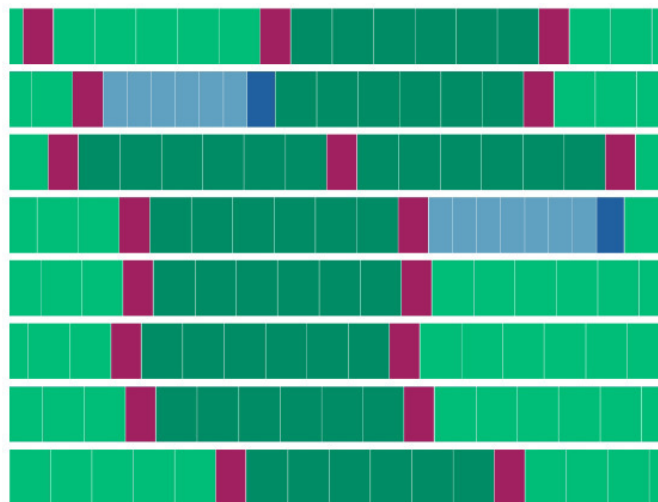


Figure 5: Magnification of a portion of the execution chart from Figure 4.

68

# Performance: Section 4.2.5

- See Figure 6 and Table 5

## Figure 6

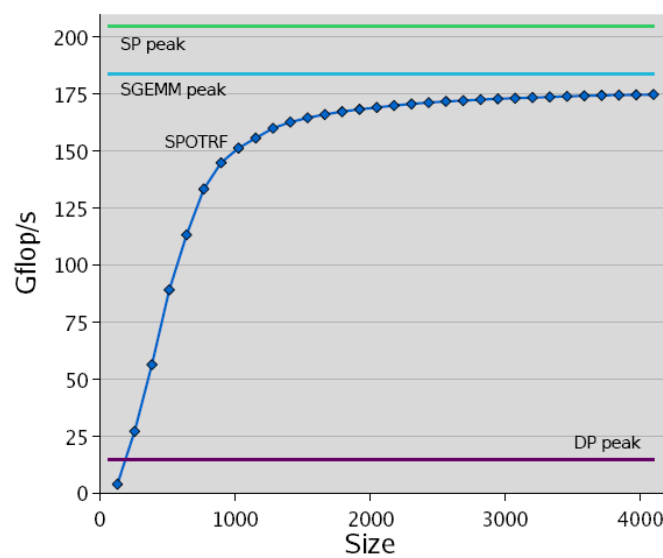


Figure 6: Performance of single precision Cholesky factorization.



## Table 3

Size	Gflop/s	% CELL Peak	% SGEMM Peak
512	92	45	50
640	113	55	62
1024	151	74	82
1280	160	78	87
1536	165	80	89
1664	166	81	90
2176	170	83	92
4096	175	85	<b>95</b>

Table 3: Selected performance points of single precision Cholesky factorization.

71



## Section 4.3 Refinement

- Steps 3 & 8 of Algorithm 1 are identical subroutine calls; this is a STRSV Blas 2 computation
- Step 6 is a DSYMV computation
  - it is performed in parallel on all eight SPE's but using only slow DP arithmetic
- usually memory bound; STRSV is
  - DSYMV is not; it is borderline

72



## Section 4.3.1 Triangular Solve

- Simple calculation shows only 6.25% of peak flops can be obtained
- No mention is made that  $1^{\text{st}} L^*y = b$  could be buried in the factorization
- Analysis showed a parallel version was required to exploit the peak BW of Cell
  - another use of the A & A approach

73



## 4.3.1 continued

- want to pipeline; overlap computation with computation
- found a so-called sweet spot; number of SPE's was found to be four
- did not define what continuous generation of traffic meant
- page 11, top of column two; very sketchy details given
- see Figure 7 for the data layout on four processors

74

# Figure 7

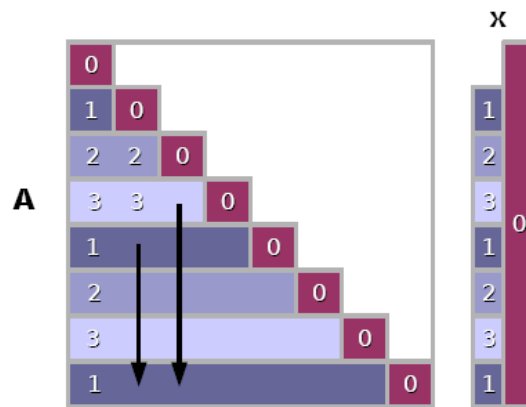


Figure 7: Distribution of work in the triangular solve routine.

75

## STRSV details

- solve is done in-place: this is a requirement of STRSV
  - each part of b/x is read into its SPE at the start and is written out by SPE zero at the end
- the same progress table (aka dag) is used as in the short Cholesky factor code
- performance is given in Figure 8
  - we see no reason why performance should differ; perhaps the matrix A is already partly in the SPE's for step 3 of Algorithm 1

76

## Figure 8

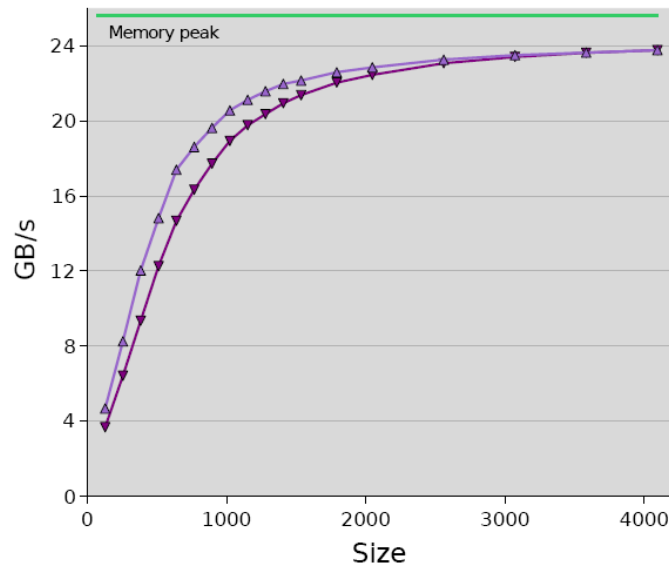


Figure 8: Performance of the triangular solve routines.

77

## 4.3.2 Matrix Vector Product

- $y \leftarrow y - A^*x$ ;  $y$ ,  $x$  are vectors &  $A$  a matrix
  - $A = A^T$
- usually a memory bound operation in long precision
- an SPE does two DP FMA's in 7 cycles and so is 14 times slower than an SP FMA
- achieves 82% of peak BW = 20.93 GB/s

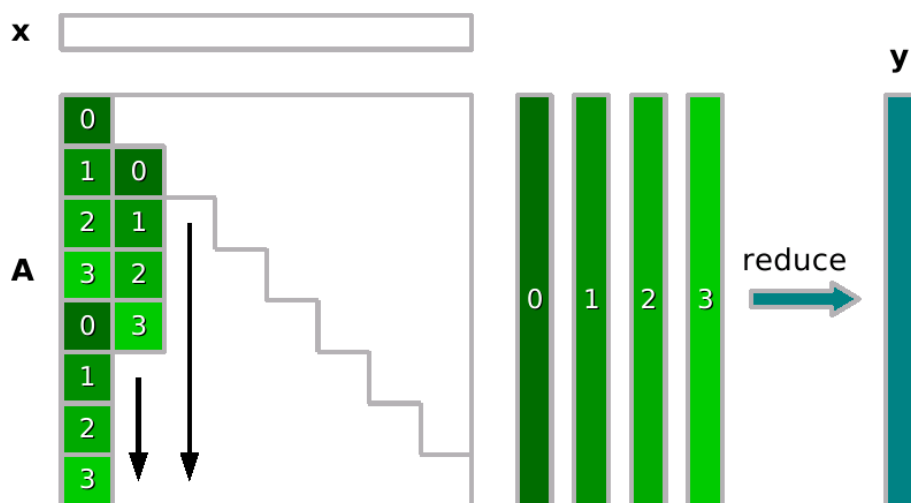
78

## 4.3.2 Details

- See next slide for the layout of A on the 8 SPE's (column-wise)
- for each order 32 sub-matrix both  $A^*x$  and  $A^T*x$  is computed
  - sub-matrix A is not transposed & is read once
  - A is applied twice with and without transpose
- copies of x & y reside on all SPE's

79

### Figure 9



80



## 4.3.2 Remaks

- See Figure 10 for a performance graph
- Could use PPE to increase performance
  - $A = B + C$ ; column-wise concatenation ;  $y = u + v$ ; compute  $u$  &  $v$  in  $\parallel$  using PPE and SPE's and sum
  - 6.4 & 14.6 split gives 44% gain modulo BW considerations

81

## Figure 10

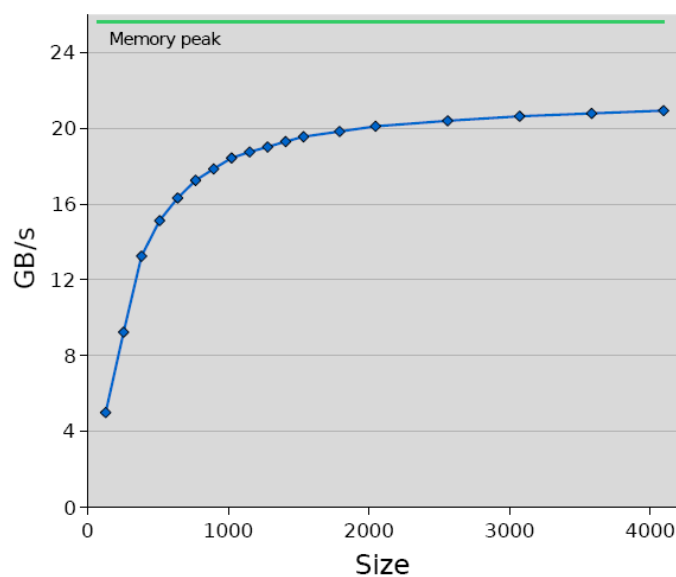


Figure 10: Performance of the matrix-vector product routine.

82



# Limitations

- Proof of concept prototype
- order of A must be a multiple of 64
  - could easily be any order
- no tests for overflow: DP to SP
- A is in full format in both SP & DP format
  - could be in SBPF or RFP format
- no mention of CM to SB layout

83



# Removal of Limitations

- Use SBP format or RFP format
  - SBP format appeared as early as 1999
  - RFP format appeared first in 2005
- if N is not a multiple of  $NB = 64$  use LDA a multiple of 64 and pad the last off-diagonal blocks with zero and the last diagonal block with zeros and ones on the diagonal

84



## Removal of Limitations continued

- One can start with standard column major format and transform in-place to SB format
  - Here my last lecture on dimension theory and on fast block in-place transposition

85

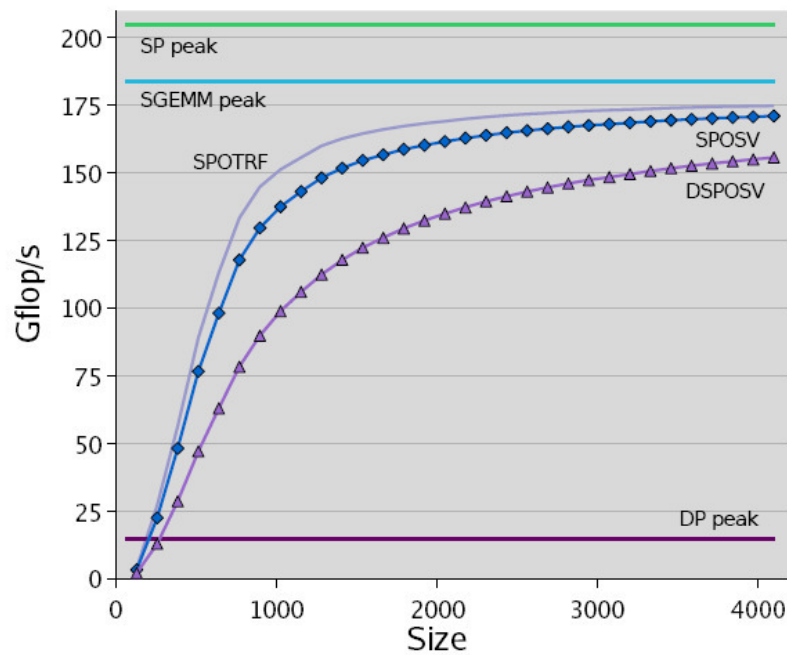


## Results and Discussion

- Fig. 11 gives performance of SPOTRF, SPOSV and DSPOSV
- huge 16 MB pages used
- well conditioned matrices used
  - only two steps of iterative refinement needed
- over 10x faster than a DP implementation

86

# Figure 11



87

# Figure 12 discussion

- Sony PlayStation 3 result
- Six SPE's and 256 KB memory & 16 MB pages gave 104 GFlops for 2048 order matrix at DP accuracy; 16% overhead

88

## Figure 12

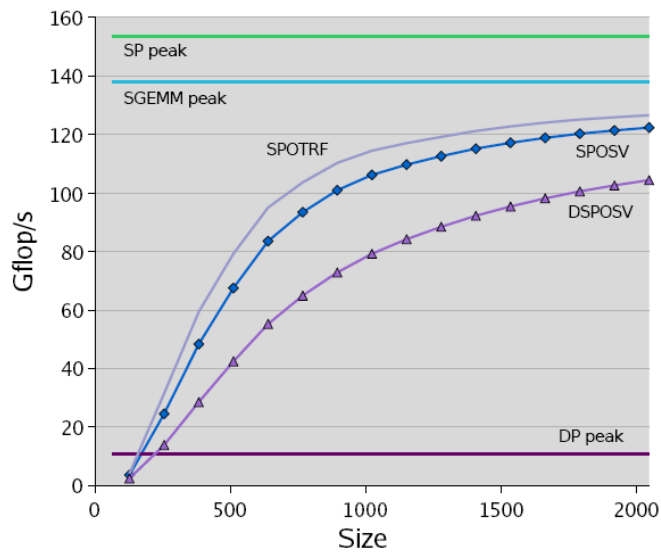


Figure 12: Performance of the mixed-precision algorithm versus the single precision algorithm on Sony PlayStation 3.

89

## Conclusions

- High potential for DLA workloads
- A & A approaches give impressive DP accurate results using very fast SP and iterative refinement

90