Embedding Application Knowledge for Improved Dynamic Adaptation

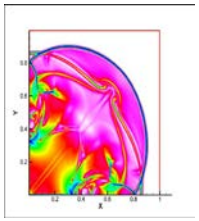
Shonan Meeting

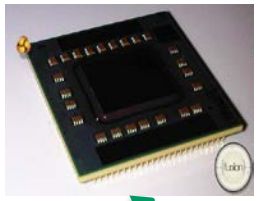
May 2014, Shonan Village

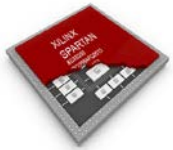
Artem Shinkarov, Sven-Bodo Scholz

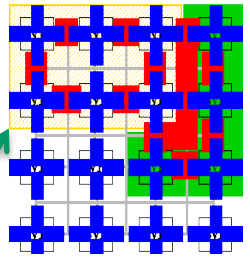
HP³ Vision

HERIOT WATT UNIVERSITY









compiler

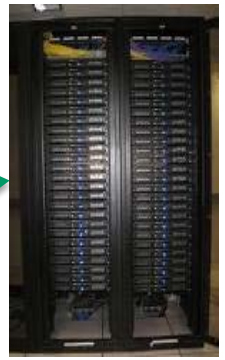
algorithm

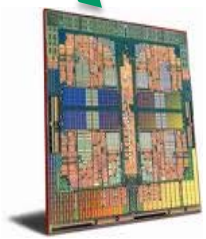
OpenCL

VHDL

µTC

MPI/OpenMP





High **P**roductivity

High **P**erformance

High **P**ortability

Declarative Sales Pitch

HERIOT WATT UNIVERSITY

all effects are known !

no notion of memory !

Magic Potion!!!

What if.....

- no low-level fix!
- no cost intuition!

it all seems to hinge on the code generator...

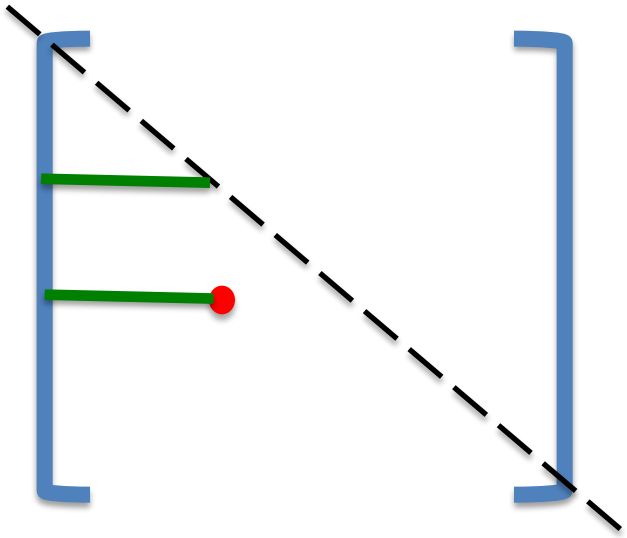
Cholesky Decomposition

$$A = LL^T$$

$$= \begin{pmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{31} & l_{33} \end{pmatrix} \begin{pmatrix} l_{11} & l_{21} & l_{31} \\ 0 & l_{22} & l_{31} \\ 0 & 0 & l_{33} \end{pmatrix}$$

Direct Solution

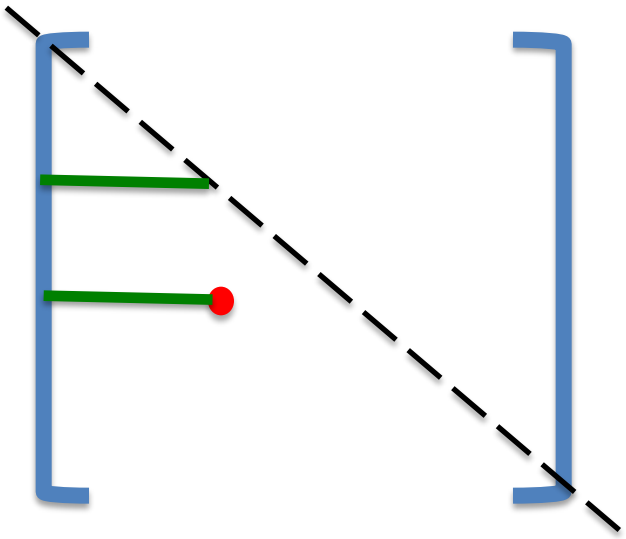
$$l_{ij} = \begin{cases} \sqrt{a_{ij} - \sum_{k=1}^{j-1} l_{ik} l_{jk}} & i = j \\ \frac{1}{l_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} l_{jk} \right) & i > j \end{cases}$$



- **highly sequential!**
- **irregular memory pattern!**

Direct Solution

$$l_{ij} = \begin{cases} \sqrt{a_{ij} - \sum_{k=1}^{j-1} l_{ik} l_{jk}} & i = j \\ \frac{1}{l_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} l_{jk} \right) & i > j \end{cases}$$

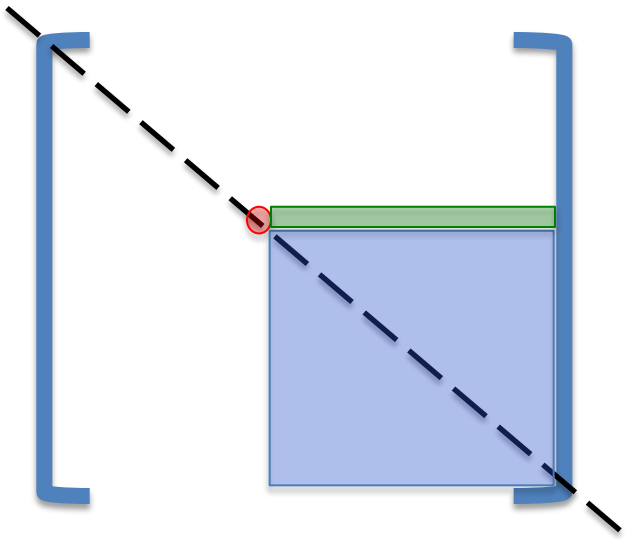


```
for (i = 0; i < n; i++)
   for (j = 0; j < i + 1; j++) {
      s = 0d;
      for (k = 0; k< j; k++)
        s +=A[i,k] *A[j,k];
      if (i == j)
        A[i,j] = sqrt (A[i,i] − s);
      else
        A[i,j] = (1.0d / A[j,j]) * (A[i,j] − s);
```

A Data-Parallel Solution

$$\begin{pmatrix} \alpha_{11} & a_{21}^T \\ a_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} \lambda_{11} & 0 \\ l_{21} & L_{22} \end{pmatrix} \begin{pmatrix} \lambda_{11} & l_{21}^T \\ 0 & L_{22}^T \end{pmatrix}$$

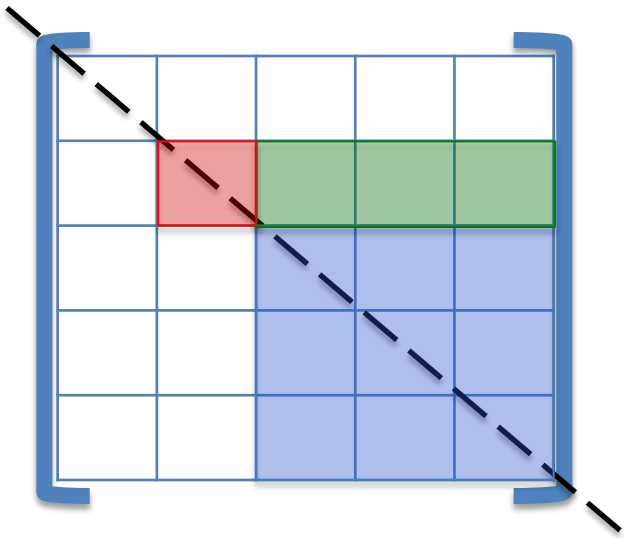
$$= \begin{pmatrix} \lambda_{11}^2 & \lambda_{11} l_{21}^T \\ \lambda_{11} l_{21} & l_{21} l_{21}^T + L_{22} L_{22}^T \end{pmatrix}$$



```
for (i = 0; i < n; i++) {
  A[i,i] = sqrt( A[i,i]);
  A = with {
      ([i,i+1] <= iv < [i+1, n]) : A[iv] / A[i,i];
    } : modarray( A);
  A = with {
      ( [i+1,i+1] <= [j,k] <  [n,n] ) : A[j,k] - A[i,j] * A[i,k];
    } : modarray( A);
}
```

$$\left(\begin{array}{cc} \lambda_{11} = \sqrt{\alpha_{11}} & l_{21}^T = \frac{a_{21}^T}{\lambda_{11}} \\ l_{21} = \frac{a_{21}}{\lambda_{11}} & L_{22} = \text{cholesky}(A_{22} - l_{21} l_{21}^T) \end{array} \right)$$

Blocking for Locality



$$\begin{pmatrix} A_{11} & A_{21}^T \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} L_{11}^T & L_{21}^T \\ 0 & L_{22}^T \end{pmatrix}$$

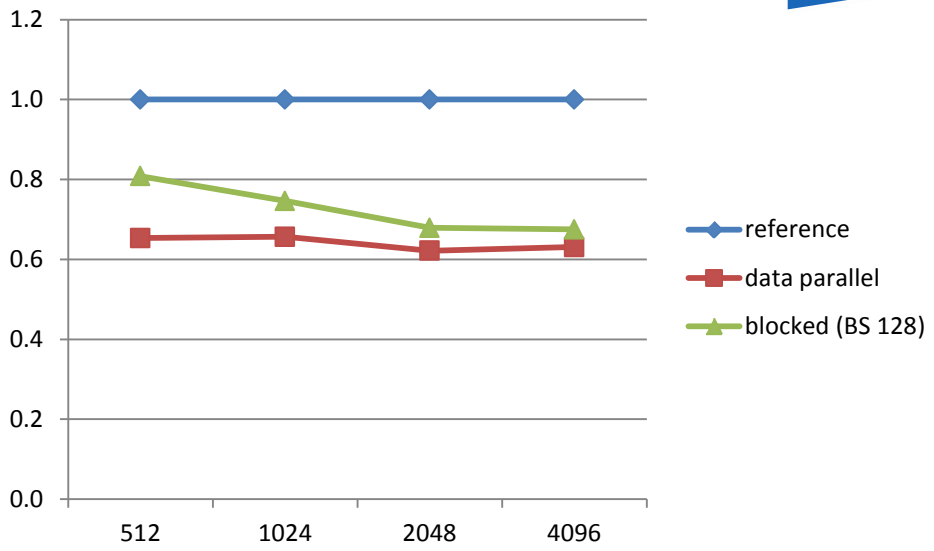
$$= \begin{pmatrix} L_{11}L_{11}^T & L_{11}L_{21}^T \\ L_{21}L_{11}^T & L_{21}L_{21}^T + L_{22}L_{22}^T \end{pmatrix}$$

$$\begin{pmatrix} L_{11} = \text{cholesky}(A_{11}) & L_{21}^T = L_{11}^{-1}A_{21}^T \\ L_{21} = A_{21}L_{11}^{-T} & L_{22} = \text{cholesky}(A_{22} - L_{21}L_{21}^T) \end{pmatrix}$$

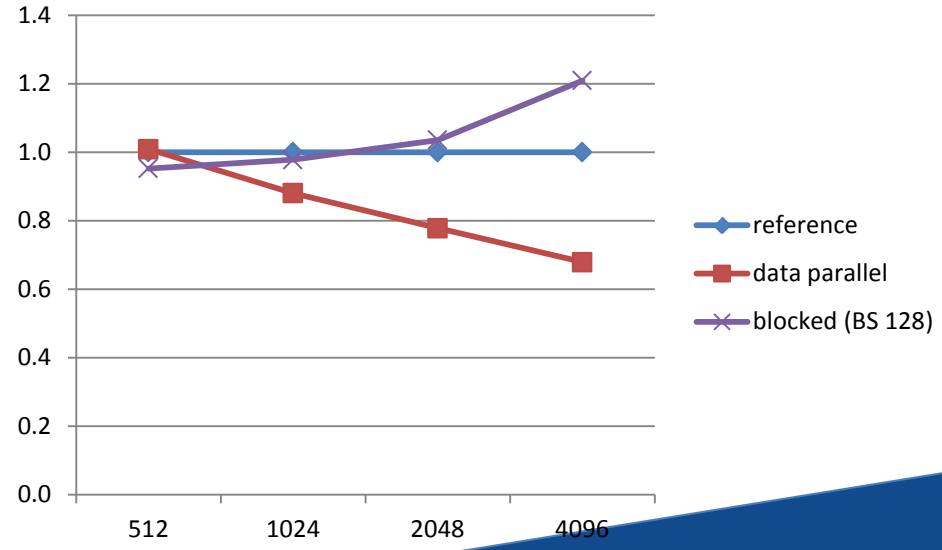
```
for (bi = 0; bi < bn; bi++) {
  A[bi,bi] = cholesky( A[bi,bi]);
  A = with {
      ([bi,bi+1] <= iv < [bi+1, bn]) :
                        trisolve( A[bi,bi], A[iv]);
    } : modarray( A);
  A = with { // dh
      ( [bi+1,bi+1] <= [bj,bk] <  [bn,bn] ) :
                        A[bj,bk] –
          matMul( transpose( A[bi,bj]), A[bi,bk]);
    } : modarray( A);
}
```

Sequential Runtimes

SaC implementations

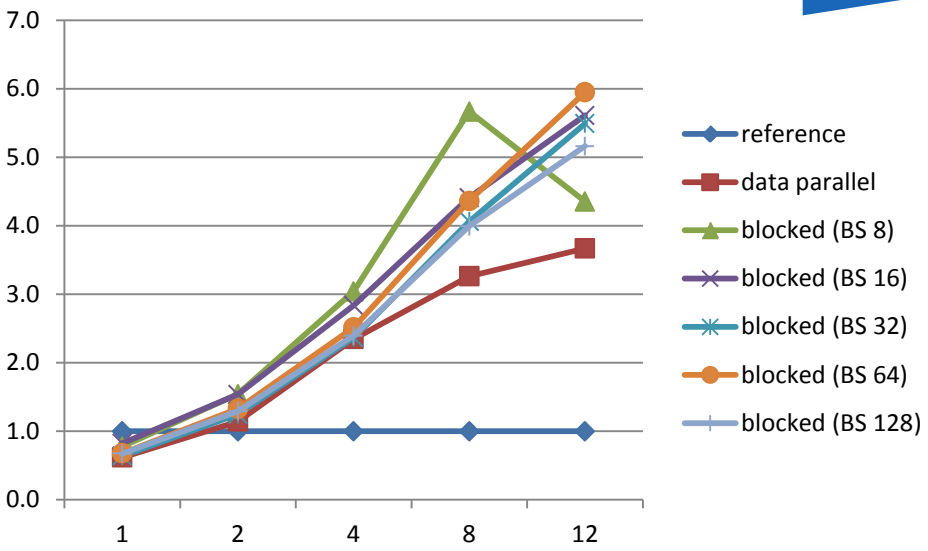


C implementations

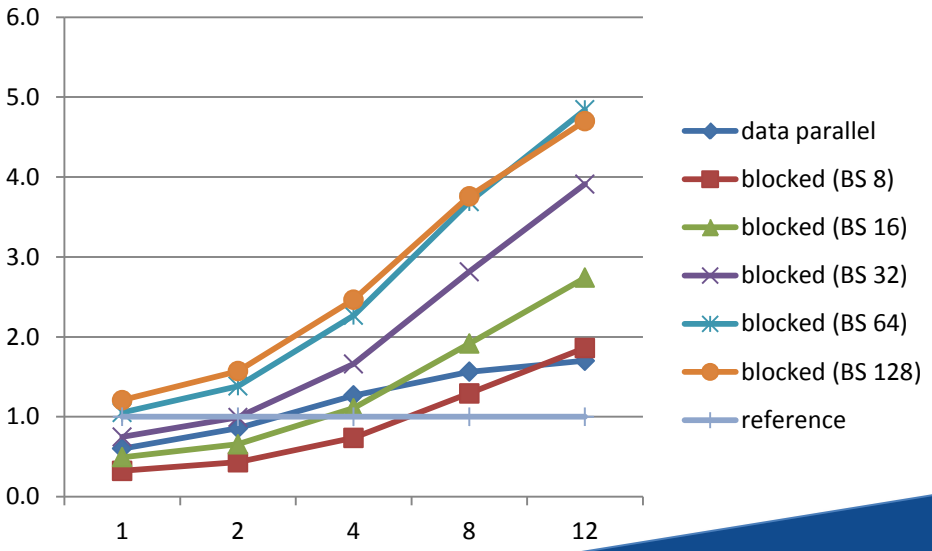


SMT Scaling

SaC implementations

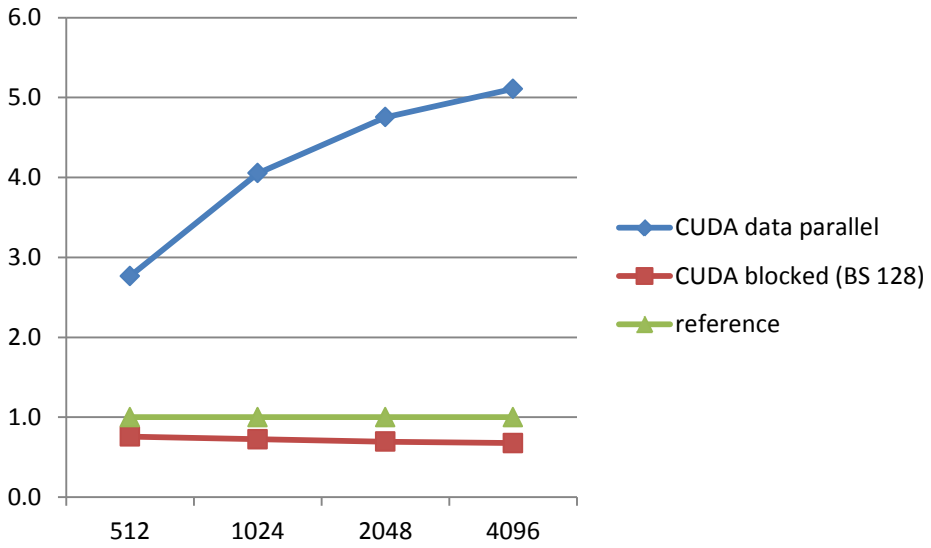


C implementations



CUDA Runtimes

SaC implementations



Lesson Learnt

Choice of algorithm depends on

- target architecture
- problem size
- application context
- compiler capabilities

Declarative Solution

HERIOT WATT UNIVERSITY

Fixed Choice???

Equality - Declaration

```
double[.,.] cholesky( double[.,.] A)
{...implementation 1...}

double[.,.] cholesky( double[.,.] A)
{...implementation 2...}

double[.,.] cholesky( double[.,.] A)
{...implementation 3...}
```

non-deterministic choice by the compiler

Parameterised Equalities

```
double[.,.] cholesky( double[N,N] A)
{ ...}

double[.,.]
cholesky[int bse | bse > 2 && bse < log2(N) ]  (double[N,N] AA)
{
  A = block (AA, 2^bse);
  ...
  return ( unblock(A));
}
```

Open Questions

- How to choose?
  - cost model?
  - context?
  - performance prediction?
  - machine learning?
  - …
- Can such a blocking be generalised and then staged?
- How far can such an approach bridge the gap to DSLs?