VECTORIZED REGISTER TILING

Abstract:

In the last years, there has been much effort in commercial compilers (icc, gcc) to exploit efficiently the SIMD capabilities and the memory hierarchy that the current processors offer. However, the small numbers of compilers that can automatically exploit these characteristics achieve in most cases unsatisfactory results. Therefore, the programmers often need to apply by hand the optimizations to the source code, write manually the code in assembly or use compiler built-in functions (such intrinsics) to achieve high performance. In this work, we present source-to-source transformations that help commercial compilers exploiting the memory hierarchy and generating efficient SIMD code which can be applied in an automated way. Results obtained on our experiments show that our solutions achieve as excellent performance as hand-optimized vendor-supplied numerical libraries (written in assembly). Our source-to-source transformations are based on the tiling, strip-mining, scalar replacement and unroll and jam transformations. In particular, we apply these transformations to loop nests and show their effectiveness; the tiling transformation permits us to exploit the reuse at the register bank, the strip-mining transformation helps us applying outer loop vectorization, the unroll and jam transformation permits unrolling vectorized outer loops and finally, the scalar replacement concept is applied to vectorized loops to obtain what we call vector replacement (scalar replacement applied to vector registers). We have compared the performance obtained with these transformations against what MKL and ATLAS get and concluded that it is possible to achieve high performance in numerical applications applying only source-to-source transformations and letting the compiler to do the low-level optimization work.

VECTORIZED REGISTER TILING

Alejandro Berna, Marta Jiménez and José M. Llabería

Departamanent d'Arquitectura de Computadors. Universitat Politècnica de Catalunya.
Barcelona, Spain. {aberna, marta, llaberia}@ac.upc.edu

introduction

Comercial compilers do not fully exploit the new characteristics of the current processors

→ they need to exploit → SIMD + memory hierarchy

→ programmer can help the compiler → exposing explicitly the different optimizations; using pragmas and keywords provided by the compilers

objective

Perform the following transformations to matrix product:

- Tiling at the register level
- Outer loop Vectorization

using

with pragmas; with keywords

Source to source transformations

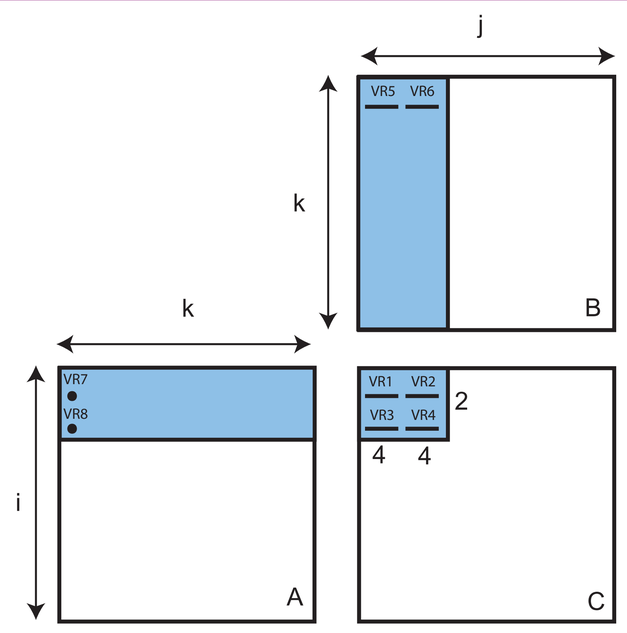
no intrinsics; no ASM; no shared libraries

Target architecture: Nehalem
Compiler: icc

desired resulting pseudocode

```
Vector Register = VR1, VR2, VR3, VR4, VR5, VR6, VR7, VR8;
for(ii=0;ii<dimi;ii+=2)
    for(jj=0;jj<dimj;jj+=8)
    {
        VR1 = C[ii*dimj+(jj:jj+3)];
        VR2 = C[ii*dimj+(jj+4:jj+7)];
        VR3 = C[(ii+1)*dimj+(jj:jj+3)];
        VR4 = C[(ii+1)*dimj+(jj+4:jj+7)];                    C LOADS
        for(k=0;k<dimk;k++)                                  register tile
        {
            VR5 = B[k*dimj+(jj:jj+3)];
            VR6 = B[k*dimj+(jj+4:jj+7)];
            VR7 = 4 copies of A[ii*dimk + k];
            VR8 = 4 copies of A[(ii+1)*dimk + k];            A & B LOADS

            VR1 = VR1 + VR5*VR7;
            VR2 = VR2 + VR6*VR7;
            VR3 = VR3 + VR5*VR8;
            VR4 = VR4 + VR6*VR8;                             C+=A*B COMPUTATION
        }
        C[ii*dimj+(jj:jj+3)] = VR1;
        C[ii*dimj+(jj+4:jj+7)] = VR2;
        C[(ii+1)*dimj+(jj:jj+3)] = VR3;
        C[(ii+1)*dimj+(jj+4:jj+7)] = VR4;                    C STORES
    }
```



icc behaviour

we observe that icc:

only performs inner loop vectorization

does not unroll strip-mined vectorized loops

does not apply vector replacement* (VR) in unrolled loop body

we solve by applying:

strip-mining to outer loops to expose the vector statements as inner loops

unroll & jam the strip-mined loop in the source code

identification of adjacent array references with pointer variables to expose vector register reuse

example:

```
for (i=0; i<dimi; i+=VL)
    for (j=0; j<dimj; j++)
        for(vi = i; vi < i+VL; vi++)
            A[vi]=A[vi]+B[j];
                                  vector statement
```

```
for (i=0; i<dimi; i+=2*VL)
    for (j=0; j<dimj; j++)
        #pragma vector always
        for(vi = i; vi < i+VL; vi++)
        {
            A[vi]=A[vi]+B[j];
            A[vi+4]=A[vi+4]+B[j];
        }
```

```
float *A1, *A2;
A1 = A; A2 = A1 + VL;
for(i = 0; i<dimj; i+=2*VL)
{
    for(j=0; j<dimi; j++)
        #pragma vector always
        for(vi = 0; vi < VL; vi++)
        {
            A1[vi]=A1[vi]+B[j];
            A2[vi]=A2[vi]+B[j];
        }
    A1+=2*VL; A2+=2*VL;
}
```

*Scalar replacement applied to vector registers

matrix product

VL = vector register length

Scalar | SIMD

icc does not apply tiling at register level

→ Apply explicity register tiling

```
register float C1, C2, C3, C4, C5, C6, C7, C8, C9, C10, C11, C12, C13, C14, C15, C16;
for (ii = 0; ii < dimi; ii+=BI)
    for (jj = 0; jj < dimj; jj+=BJ)
    {
        C1 = C[ii*dimj + jj]; C2 = C[ii*dimj + jj+1]; … C8 = C[ii*dimj + jj+7];
        C9 = C[(ii+1)*dimj + jj]; C10 = C[(ii+1)*dimj + jj+1]; C16 = C[(ii+1)*dimj + jj+7];
        for (k = 0; k < dimk; k++)
        {
            C1 += A[ii*dimk + k]*B[k*dimj + jj];             register tile
            C2 += A[ii*dimk + k]*B[k*dimj + jj+1];
            …
            C8 += A[ii*dimk + k]*B[k*dimj + jj+7];
            C9 += A[(ii+1)*dimk + k]*B[k*dimj + jj];
            C10 += A[(ii+1)*dimk + k]*B[k*dimj + jj+1];
            …
            C16 += A[(ii+1)*dimk + k]*B[k*dimj + jj+7];
        }
        C[ii*dimj + jj] = C1; C[ii*dimj + jj+1] = C2; … C[ii*dimj + jj+7] = C8;
        C[(ii+1)*dimj + jj] = C9; C[(ii+1)*dimj + jj+1] = C10; ... C[(ii+1)*dimj + jj+7] = C16;
    }
```

Code A: Register Tiling (BI=2, BJ=8), Scalar execution, Scalar Replacement.

icc does not vectorize outer loops

→ Expose the vector statement*

→ Unroll & jam the strip-mined loop

```
for (ii = 0; ii < dimi; ii+=BI)
    for (jj = 0; jj < dimj; jj+=BJ)
        for (k = 0; k < dimk; k++)
            #pragma ivdep
            for(vj = jj; vj < jj + 4; vj++)
            {
                C[ii*dimj + vj] += A[ii*dimk + k] * B[k*dimj + vj];
                C[(ii+1)*dimj + vj] += A[(ii+1)*dimk + k] * B[k*dimj + vj];
                C[ii*dimj + vj+4] += A[ii*dimk + k] * B[k*dimj + vj+4];
                C[(ii+1)*dimj + vj+4] += A[(ii+1)*dimk + k] * B[k*dimj + vj+4];
            }
```

Code B: Register Tiling, SIMD

*Strip-mining already applied by the tiling transformation

icc does not apply vector replacement

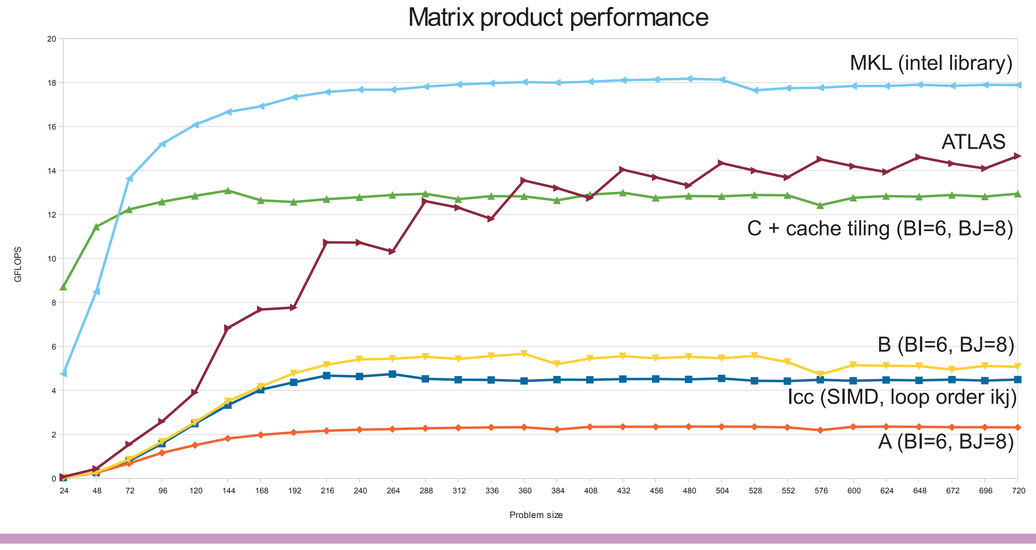
→ Identify individual array references with pointers*

```
float *C1, *C2, *C3, *C4;
const float *B1, *B2, *A1, *A2;
for (ii = 0; ii < dimi; ii+=BI)
{
    A1 = &A[ii*dimk];
    A2 = &A[(ii+1)*dimk];
    for (jj = 0; jj < dimj; jj+=BJ)
    {
        C1 = &C[ii*dimj+jj];
        C2 = &C[(ii+1)*dimj+jj];
        C3 = &C[ii*dimj+jj+4];
        C4 = &C[(ii+1)*dimj+jj+4];
        for (k = 0; k < dimk; k++)
        {
            B1 = &B[k*dimj+jj];
            B2 = &B[k*dimj+jj+4];
            #pragma ivdep
            for(vj = 0; vj<4; vj++){
                C1[vj] += A1[k]*B1[vj];
                C3[vj] += A1[k]*B2[vj];
                C2[vj] += A2[k]*B1[vj];
                C4[vj] += A2[k]*B2[vj];
            }
        }
    }
}
```

Code C: Register tiling, SIMD, Vector Replacement

*Also works with vectors

Matrix product performance



conclusions

To achieve high performance compilers should:

1. Generate efficient SIMD code
2. Exploit efficiently the memory hierachy

Source to source transformations help the compiler to generate efficient SIMD code.

references

A. J.C. Bik. "The Software Vectorization Handbook. Applying Multimedia Extensions for Maximum Performance". Intel Press. 2004.

A. V. Aho, M. Lam; R. Sethi, J. D Ullman. "Compilers Principles, Techniques and Tools", Addison Wesley 2008

D. Callahan, S. Carr, K. Kennedy. "Improving register allocation for subscripted variables". PLDI'1990, pp. 53-65, June 1990.

D. Nuzman, A. Zaks. "Outer-loop vectorization: revisited for short SIMD architectures." PACT'2008. pp.2~11.

R. C. Whaley, A. Petitet, J. Dongarra. "Automated Empirical Optimization of Software and the ATLAS project". Parallel Computing, 27(1-2):3-35, 2001.

This research has been supported by an Intel-UPC Research Grant, the Spanish Ministry of Education (contract no. TIN2007-60625), and the European Union (under the HiPEAC-2 Network of Excelence, FP7/ICT 217068).