

HPC-II Spring 2009 – Project 2

Optimizing Gaussian Elimination with SSE

Robert van Engelen

Due date: April 23, 2009

1 Introduction

1.1 HPC Account and Login Information

For this assignment you need an SCS account. The account gives you access to `pamd`. You also need an account on the FSU HPC cluster. Contact the instructor if you do not have an SCS account.

To obtain an account on the FSU HPC cluster, do the following:

- If you are an SCS student, then:
 1. Use your favorite web browser to go to <http://www.hpc.fsu.edu/>
 2. Click "Connecting" under "Getting Started" in the left hand menu.
 3. Click "HPC Account Application Details"
 4. Click "Apply" for a owner-based access account
 5. Select "Non-Faculty Account"
 6. Fill in the form:
 - (a) Enter you FSUID
 - (b) Select your unit: SCS
 - (c) Fill in all the rest of the personal information
 7. Click "Request Account"

After receiving your account creation acknowledgment, you can login to the head node from any machine:

```
ssh -Y scs.hpc.fsu.edu
```

- If you are not an SCS student, then:

1. Use your favorite web browser to go to <http://www.hpc.fsu.edu/>
2. Click "Connecting" under "Getting Started" in the left hand menu.
3. Click "HPC Account Application Details"
4. Click "Apply" for an general access account
5. Select "Sponsored Account"
6. Fill in the form:
 - (a) Enter your FSUID
 - (b) Choose your faculty sponsor: van Engelen
 - (c) Fill in all the rest of the personal information
7. Click "Request Account"

After receiving your account creation acknowledgment, you can login to the head node from any machine:

```
ssh -Y submit.hpc.fsu.edu
```

You should edit, save, and compile your files on this head node. The runs are performed on the compute node(s). After logging in on the head node, use the MOAB "msub" command to start an interactive job on one of the compute nodes assigned to the classroom queue:

```
msub -I -l nodes=1:ppn=4 -q scs_classroom
```

You can then run the compiled binaries interactively. How this is done is explained later.

From the head node you can also check the availability of the scs_classroom queue before starting a job:

```
showq -w class=scs_classroom
```

1.2 Setup

In your HPC account create a directory named "Pr2" and copy the files from:

```
http://www.cs.fsu.edu/~engelen/courses/HPC-adv/Pr2.zip
```

To do so, use `wget http://www.cs.fsu.edu/%7eengelen/courses/HPC-adv/Pr2.zip` from the head node.

The package bundles the following files:

- `build.sh`: a build script to use the Intel compiler
- `Makefile`: Makefile
- `make.x`: a set of sub-make files to compile on different platforms
- `rdtsc.h`: for RDTSC-based timing (not used in this project)
- `cputime.h` and `.c`: timing functions
- `config.guess`: guessing the platform

Use `./build.sh` to build the code on the HPC head node and execute `./gauss` on a compute node (i.e. via `msub` in another window).

1.3 Project Aims

In this project we will use SSE intrinsics to optimize the Gaussian elimination algorithm. For this project you should write a report with your findings and submit this to the instructor for grading. Include explanations of your findings in your report. Your report must address all tasks listed in the sections below and also list the source code that you wrote. You do not need to submit your source code(s) separately.

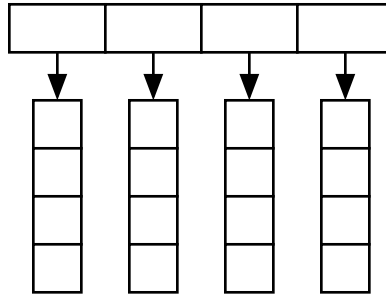
2 SSE Vectorization of the Gaussian Elimination Algorithm

Gaussian elimination solves $\mathbf{A}\mathbf{x} = \mathbf{b}$ given matrix \mathbf{A} and vector \mathbf{b} . Gaussian elimination performs a forward elimination step (reduction to reduced row echelon form) and a backward step (back substitution) to find the solution. Pivoting is used to improve the numerical stability of the algorithm.

The choice of data layout can have a profound impact on the efficiency of memory references. In general, it is best to access consecutively stored array elements in sequence in loops (spatial locality) and reuse data (temporal locality). In general, we should both attempt to rewrite the algorithm's data access patterns and simultaneously improve the data layout that best

fits the access pattern. Sometimes there is no possibility to change the algorithm's data access patterns, i.e. because of data dependences, and we can only change the data layout to fit these patterns.

For Gaussian elimination it appears best to use a column-major layout, such that elements in columns can be accessed consecutively. In C, we can use a jagged column layout by allocating an array per column as is shown in the next diagram.



Now let's move on to our algorithm!

In a previous project in HPC-I we implemented Gaussian elimination with pivoting in file `gauss.c` as follows (changed here slightly for the new project):

```
void gauss(MATRIX a, VECTOR x, int n)
{
    int i, j, k, maxloc;
    float maxval;
    int *idx;
    VECTOR tmp;

    idx = (int*)malloc(sizeof(int) * n);

    /* alloc tmp vector */
    tmp = alloc_vector(n);

    /* Init row index array */
    for (i = 0; i < n; i++)
        idx[i] = -1;

    for (k = 0; k < n; k++)
    {
        /* Find pivot element in k'th column */
        maxval = 0;
        maxloc = -1;

        for (i = 0; i < n; i++)
```

```

{
    if (idx[i] == -1 && maxval < fabs(a[k][i]))
    {
        maxval = fabs(a[k][i]);
        maxloc = i;
    }
}

/* Singular? */
if (maxval < FLT_EPSILON)
{
    fprintf(stderr, "Singular matrix!\n");
    exit(1);
}

/* Relabel row of the k'th pivot element */
idx[maxloc] = k;

/* Forward step: reduce the rows (except pivot row and previous rows) */
for (i = 0; i < n; i++)
{
    if (idx[i] == -1)
    {
        float fac = a[k][i] / a[k][maxloc];
        for (j = k; j < n+1; j++)
            a[j][i] = a[j][i] - fac * a[j][maxloc];
    }
}

/* Row exchanges for A[][] and b[] */
for (j = 0; j < n+1; j++)
{
    for (i = 0; i < n; i++)
        tmp[idx[i]] = a[j][i];
    for (i = 0; i < n; i++)
        a[j][i] = tmp[i];
}

/* Backward step: solve x by backsubstitution */
for (i = n-1; i >= 0; i--)
{
    /* Note: vector b is stored in a[n] */
    float sum = a[n][i];

    for (j = i+1; j < n; j++)
        sum = sum - a[j][i] * x[j];

    x[i] = sum/a[i][i];
}

```

```

    free_vector(tmp, n);

    free(idx);
}

```

Note that in this code:

- Vector b is stored in the column-augmented matrix a at column n , i.e. $b[i]=a[n][i]$.
- The matrix and vectors are indexed from element 0 to element $n-1$.

Most of the work takes place in the forward and backward steps. The innermost loops in these steps iterate over the j columns, while the outermost loop iterates over the rows. This seems contradictory to our earlier claim that we need to find a data layout that fits the algorithm's data access patterns.

However, we should first reorder the operations in the forward and backward steps to optimize for vectorization. Note that the inner loop in the backward step is a reduction, which does not vectorize well. So let's first rewrite the loop nest by looking carefully into the algorithm's operations:

```

/* Backward step: solve x by backsubstitution (b is stored in a[n]) */
for (j = n-1; j >= 0; j--)
{
    x[j] = a[n][j] / a[j][j];
    for (i = 0; i < j; i++)
        a[n][i] = a[n][i] - a[j][i] * x[j];
}

```

Recall that we made a similar change for the OpenMP parallelization of the inner loop to avoid the sum reduction. Instead, we perform multiple sums simultaneously using destructive assignments to vector b (stored in $a[n]$, remember?).

It should be clear now that the column major layout of a helps to vectorize the inner i loop, since we can access consecutive elements of $a[n][i]$ and $a[j][i]$. Note that $x[j]$ is a scalar value in the inner loop.

The forward step will be very inefficient in the way it is currently organized. We should apply loop distribution and loop interchange to reorder the data accesses. Before we can apply loop distribution (loop fission), we need to apply scalar expansion to fac to compute its value first:

```

/* Forward step: reduce the rows (except pivot row and previous rows) */
for (i = 0; i < n; i++)
{
  if (idx[i] == -1)
  {
    tmp[i] = a[k][i] / a[k][maxloc];
    for (j = k; j < n+1; j++)
      a[j][i] = a[j][i] - tmp[i] * a[j][maxloc];
  }
}

```

and then distribute the loops:

```

/* Forward step: reduce the rows (except pivot row and previous rows) */
for (i = 0; i < n; i++)
{
  if (idx[i] == -1)
    tmp[i] = a[k][i] / a[k][maxloc];
}
for (i = 0; i < n; i++)
{
  if (idx[i] == -1)
    for (j = k; j < n+1; j++)
      a[j][i] = a[j][i] - tmp[i] * a[j][maxloc];
}

```

Verify the flow/anti dependences to ensure correctness of this transformation. That is, check that the flow/anti dependences on `fac` and `a` are preserved. The guard `idx[i] == -1` ensures that $i \neq \text{maxloc}$ in the loop operations, so `a[j][maxloc]` is never changed (this is a property of the algorithm).

After interchanging the loops, we get:

```

/* Forward step: reduce the rows (except pivot row and previous rows) */
for (i = 0; i < n; i++)
{
  if (idx[i] == -1)
    tmp[i] = a[k][i] / a[k][maxloc];
}
for (j = k; j < n+1; j++)
{
  for (i = 0; i < n; i++)
    if (idx[i] == -1)
      a[j][i] = a[j][i] - tmp[i] * a[j][maxloc];
}

```

Note that the interchange is safe, because we have no cross-iteration dependences in the loop nest.

Finally, we should make some improvements to get rid of the guard in the innermost loop at some expense of performing more operations:

```
/* Forward step: reduce the rows (except pivot row and previous rows) */
t = a[k][maxloc];
for (i = 0; i < n; i++)
{
    if (idx[i] == -1)
        tmp[i] = a[k][i] / t;
    else
        tmp[i] = 0.0;
}
for (j = k; j < n+1; j++)
{
    t = a[j][maxloc];
    for (i = 0; i < n; i++)
        a[j][i] = a[j][i] - tmp[i] * t;
}
```

The `i` loops are now good candidates for vectorization. The guard in the first `i` loop can be implemented using logical operations on floats in vectors, since it is safe to evaluate `a[k][i]/t` for any `i`.

The changes discussed above are implemented in `gauss.c` that you downloaded. The places where SSE vectorization is applicable or aligned memory allocation is needed are marked by `TODO` comments in the source code. The forward and backward steps are most important, since most of the work is done there. Some other places can be improved with SSE and are marked as `TODO CHALLENGE`. For 16-byte aligned memory allocation/free, use `_mm_malloc` and `_mm_free`.

For this project use SSE intrinsics to optimize the marked loops. Use the techniques and SSE examples discussed in class. What is the speedup obtained after vectorization?

For a detailed SSE intrinsics overview, see
http://www.cs.fsu.edu/~engelen/courses/HPC-adv/intref_cls.pdf

Note that when you compile the project with `./build.sh` we turned automatic vectorization off with option `-vec-`. Otherwise, the Intel compiler automatically vectorizes two loops. We want to compare the performance of the non-vectorized code and the SSE vectorized code. The loops that the Intel compiler vectorized are also not the loops that we are most interested in to optimize. In general, automatic vectorization has certain limitations based on loop structure and when array data dependence testing disproves flow dependences.

End.