1. Attached.

2. (a) The sufficient condition for loop unrolling is that the permutation that moves the unrolled loop innermost be valid. So we cannot unroll t, because this will make the first dependence vector lexicographically negative. We can unroll j, because it is always valid to unroll an inner loop. i can be unrolled because moving i to the inner most loop will still leave us with lexicographically positive dependence vectors $(1,-1,0),(1,1,0),(2,0,1)$.

   (b) There are six permutations possible for each vector. All permutations of the last two vectors are lexicographically positive, so we only need to concern ourselves with the first vector. Its permutations are:
   (1, 0, -1), (1, -1, 0), (0, 1, -1), (0, -1, 1), (-1, 0, 1), (-1, 1, 0) We see that only the valid permutations are tij, tji, itj.

   (c)
```
for(int i = 0; i < 1024; i++)
    for(int j = 0; j < 1024; j++)
        for(int t = 0; t <= i; t++)
            S(t, i, j)
```

   (d) Tiling a set of loops is valid when that set of loops is fully permutable. As we saw in (b) the loop is not fully permutable. The two inner loops are permutable, however, so 2D tiling is possible for the i, j loops.

3. Consider the pair A_{ij} and $A_{i-1,j}$. δ^f?

$$i_w < i_r \text{ or } i_w = i_r,\ j_w < j_r$$
$$i_w = i_r - 1,\ \ j_w = j_r$$
$$i_r - i_w = i_r - (i_r - 1) = 1$$
$$j_r - j_w = j_r - j_r = 0 \implies \delta^f\ (1,0)$$

   Consider the pair A_{ij} and $A_{i,j-1}$. δ^f?

$$i_w < i_r \text{ or } i_w = i_r,\ j_w < j_r$$
$$i_w = i_r,\ \ j_w = j_r - 1$$
$$i_r - i_w = 0\ \ j_r - j_w = j_r - (j_r - 1) = 1 \implies \delta^f(0,1)$$

   Consider the pair A_{ij} and $A_{i+1,j+1}$. δ^f?

$$i_w < i_r \text{ or } i_w = i_r,\ j_w < j_r$$
$$i_w = i_r + 1,\ \ j_w = j_r + 1$$
$$i_r - i_w = i_r - (i_r + 1) = -1,\ \ j_r - j_w = j_r - (j_r + 1) = -1$$
$$\implies \text{No } \delta^f, (-1,-1)$$
$$\delta^a?$$
$$i_w > i_r \text{ or } i_w = i_r,\ \ j_w > j_r$$
$$i_w - i_r = i_w - (i_w - 1) = 1,\ \ j_w - j_r = j_w - (j_w - 1) = 1$$
$$\implies \delta^a(1,1)$$

   There is also an output dependence as the loop will overwrite A_{ij} during each loop of t.

4. (a)
```
for i = 1, N, 2
        t1 = b(i)
        t2 = b(i+1)
        for j = 1, i-1
                t1 = t1 - a(i,j)*x(j)
                t2 = t2 - a(i+1,j)*x(j)
        x(i) = t1/a(i,i)
        t2 =  t2 - a(i+1, i)*x(i)
        x(i+1) = t2/a(i+1,i+1)
```

Also, here is a little Python script I wrote to verify that I was correct, along with the runtime compared to the Saday code and the Python built in linear solver.

```
from numpy import *
from numpy.random import *
from numpy.linalg import solve
from time import time

N = 20
A = tril(random((N,N)))
x1 = zeros(N)
x2 = zeros(N)
b = random(N)

# Given by hw assignment
t0 = time()
for i in range(N):
    temp = b[i]
    for j in range(0, i) :
        temp = temp - A[i][j]*x1[j]
    x1[i] = temp/A[i][i]
t1 = time()
# print x
print "Saday Code", t1 - t0

# python built in solver
t0 = time()
y = solve(A,b)
t1 = time()
# print y
print "Python Solver", t1 - t0

# Now I will test the modified code.
t0 = time()
for i in range(0,N,2):
    t1 = b[i]
    t2 = b[i+1]
    for j in range(0,i):
        t1 = t1 - A[i][j]*x2[j]
```

```
        t2 = t2 - A[i+1][j]*x2[j]
    x2[i] = t1/A[i][i]
    t2 = t2 - A[i+1][i]*x2[i]
    x2[i+1] = t2/A[i+1][i+1]
t1 = time()

# print x
print "Unrolled time", t1 - t0
print where(x1!=x2) # The results for rolled and unrolled are identical.
# I havent included the results comparing the x and y arrays,
# because they differ due to rounding error.
#(The error become large for large N).
```

Here are the code results:

```
melvyn@melvyn-Satellite-C655:~$ python unroll.py
Saday Code 0.000399112701416
Python Solver 0.000165939331055
Unrolled time 0.000373125076294
(array([], dtype=int32),)
```

(b) The ji form of the loop is as follows:

```
for j = 1, N
        x[j] = b[j]/A[j][j]
        for i = j+1,N
                b[i] -= A[i]][j]*x[j]
```

Again, I made a Python script tp verify, and my result checks out against the built in solver.

```
from numpy import *
from numpy.random import *
from time import time
from numpy.linalg import solve

N = 10
A = tril(random ((N,N)))
x = zeros(N)
b = rand(N)
b2 = array(b) # This is an easy way to make a deep copy.

for j in range(N):
    x[j] = b[j]/A[j][j]
    for i in range(j+1, N):
        b[i] = b[i] -  A[i][j]*x[j]

print x
print solve(A,b2)
```

And here are the results, which show the arrays are the same:

```
melvyn@melvyn-Satellite-C655:~$ python unroll2.py
[  4.89106596e+00  -5.84569282e+00   1.54170227e-01   2.11669445e+02
  -2.22506331e+02   3.83789822e+01  -2.67779209e+02   5.62652140e+02
   2.75639278e+01   4.89768849e+02]
[  4.89106596e+00  -5.84569282e+00   1.54170227e-01   2.11669445e+02
  -2.22506331e+02   3.83789822e+01  -2.67779209e+02   5.62652140e+02
   2.75639278e+01   4.89768849e+02]
```

(c) We should expect the SAXPY solver to perform worse, since the inner loop accesses different rows of A, and will result in a cache miss for every iteration. Since we are likely going to be solving large systems of equations, all of A accessed by the inner loop will not fit in the cache. I think that A will not stay in the cache between outer loop iterations because I think that most caches are not highly associative, and thus we will likely have many evictions as we loop through the inner loop. Of course, the number of outer loop iterations which cause cache misses will depend upon the cache size and array dimensions.The number of inner loop misses is:

$$\sum_{1}^{N} i = \frac{N(N+1)}{2}$$

The algorithm presented by the homework assignment will iterate across rows of A in its inner loop and have fewer cache misses in this inner loop. Every iteration of the outer loop will cause a cache miss.

This analysis is not thorough, because as we saw in hw1, runtime and miss rates are not always correlated as we expect due to varying miss penalties. Also, without knowledge of associativity and matrix, block and cache size, we cannot give a definitive answer. Also, I neglected to mention the misses for $\vec{x}$ and $\vec{b}$. This is because the inner loop for both the ij ad ji versions of the code miss once per B iterations, where B is the blocksize, due to the x and b vectors, respectively.

Again, I compared the two algorithms and confirmed my conjecture.

```
from numpy import *
from numpy.random import *
from time import time
from numpy.linalg import solve

N = 1500
A = tril(random ((N,N)))
x = zeros(N)
x1 = zeros(N)
b = rand(N)
b2 = array(b) # This is an easy way to make a deep copy.

t0 = time()
for i in range(N):
    for j in range(0, i) :
        b[i] -= A[i][j]*x1[j]
```

```
    x1[i] = b[i]/A[i][i]
t1 = time()
print "ij time", t1 - t0

t0 = time()
for j in range(N):
    x[j] = b2[j]/A[j][j]
    for i in range(j+1, N):
        b2[i] = b2[i] -  A[i][j]*x[j]
t1 = time()
print "ji time: ", t1 - t0
print where(x1 != x) # Nowhere

melvyn@melvyn-Satellite-C655:~$ python unroll2.py
ij time 2.2582681179
ji time:  2.33067584038
(array([], dtype=int32),)
```