Coding for SIMD architectures using ATLAS

Peter Søndergaard

8th September 2001

Contents

1 Introduction.

The goal of this project has been to produce a very fast implementation of single precision matrix multiply for the Intel Pentium III and AMD Athlon architectures. Both architectures have so called SIMD (Single Instruction Multiple Data) instructions, that operate on two or four element at the same time, and the intention was to utilize these instructions. This project relies heavily on the ATLAS project, and it has been possible to built the performance sensitive code as a “plug-in” to ATLAS, and to use all the timers and infrastructure provided by ATLAS to greatly speed up the development process.

I would like to thank Clint Whaley for making the nice framework in ATLAS and Camm Maguire for providing good inspiration and healthy competition.

1.1 Typography.

To describe the examples of the code I have made, I simply use the generated code itself, so at this place I will give a description of how to read it. Since the problem is always doing matrix multiplication, which only involves addition and multiplication, only three different kinds of instructions are needed: Instructions for moving data, adding and for multiplying. They are referred to as `vec_mov`, `vec_add` and `vec_mul`. A suffix is added to this to indicate where the operand to the instruction are placed: m if the operand is in memory and r if it is in a register. Input operands are placed first, and output operands last! This is the way that GNU assembler places the operands, but it is the reverse of the Intel assembler.

So, a sequence of instructions might look like this:

```
vec_mov_mr(pB0,reg4);
vec_mov_mr(pA0,reg2);
vec_mul_rr(reg4,reg2);
vec_add_rr(reg2,reg0);
```

This sequence will load the value pointed to by `pB0` into register 4 and load the value pointed to by `pA0` into register 2. Register 4 is then multiplied by register 2, and the result is placed in register 2. Finally, this is added to register 0, and the result is placed in register 0. The variable of the type `pA0` means a pointer to the first element in the first column of A, `pA1` would be a pointer to the second column of A. The same naming scheme is used for pointers to elements in B and C. Registers are always named `reg0`, `reg1`, `reg2` ... etc, so a name of this type will always refer to a register. Leading dimensions of matrices will be denoted by the variable names `lda`, `ldb` and `ldc`.

Everything else except this will be written in the C programming language.

2 The Problems.

2.1 Pluggin in to ATLAS.

2.2 GEMM, matrix - matrix multiplication.

The problem that we wish to solve is matrix multiplication, and we want to do it as fast as possible. The problem is $C = A \cdot B$, where C is a $m \times n$ matrix, A is a $m \times k$ matrix and B is a $k \times n$ matrix. The symbols will keep the same meaning all through this report, and will be used frequently.

ATLAS provides a method for people to contribute an optimized "kernel" to ATLAS, which ATLAS will use as a basis for building a full general matrix multiplication routine. The advantage is that ATLAS provides a simple interface to the programmer, and will then take care of all the details of doing cache blocking (see later), doing complex precision routines, supplying all variations of the problem: AB, A^TB,AB^T and A^TB^T. All these variants can be covered by writing just one routine, because ATLAS will copy the data to the correct format, so the matrix-transposition will happen during this data copying process. ATLAS has a code generator and timing routines that ensures, that if it can generate faster code than the code you have contributed, it will use its own code. Also, if another person has contributed faster code than you for a specific purpose, it will use his code instead. This way, the final library might end up being composed of ATLAS' automatically generated code, and specialized code from several different contributers to ensure a fast library for all cases of the matrix multiplication.

2.3 GEMV, matrix - vector multiplication.

Matrix - vector multiplication. Even though this seems similar to matrix - matrix multiplication, there is one big difference: The order of computation is now only $O(n^2)$ but the order of data is still $O(n^2)$. This means that data movement and not computation will be the major bottleneck. For each data load the CPU can only do two computations, so the computational speed is limited by the memory bus as shown below:

CPU and memory:	Bus speed: GB / s	Precision:	Peak: Mflops
PIII / SDRam:	1.066	Single	533
		Double	266
Athlon / DDR	2.13	Single	1066
		Double	533
P4 / Rambus	3.2	Single	1600
		Double	800

3 Modern low cost SIMD architectures.

This work has been targeted at the Intel Pentium III and Pentium IV processors, the AMD K6-2, K6-3, Athlon/Duron, Palomino and *Hammer chips and the Motorola G4 chips. They all have so called SIMD, Single Instruction Multiple Data, instructions. These instructions operate on "vector" registers holding two or four elements of single or double precision, and so they are very far from what traditionally has been known as vector computers. The capabilities of the instruction sets can be summarized as follows:

Ins. Set:	Vector length	Nr. registers.	Precision.	muladd
3DNow!	64 bits.	8	Single	separate
SSE1	128 bits.	8	Single	separate
SSE2	128 bits.	8	Double	separate
AltiVec	128 bits.	32	Single	combined

The names in the first column is the names of the instruction sets. AMD uses 3DNow!, but is changing towards SSE1 and SSE2, which is used by Intel. Motorola uses the AltiVec instruction set. The vector length of 3Dnow! allows it to hold two single precision values, SSE1 can hold four single precision values, SSE2 holds 2 double precision values and AltiVec four single precision values. The last column indicates whether the architecture has two different instructions for multiplication, or one big combined instruction capable of doing first a multiplication and then an addition. This can be a huge advantage since it saves both code and registers, and makes the code much easier to optimize.

The architectures has originally been developed for computer games, image processing, DVD encoding/decoding etc. which does not need the accuracy of double precision, and therefore most of the chips only support single precision. The 3DNow! is by far the worst, since it does not even support infinities and NaNs (not-a-number), and it is therefore easy not to spot a programming error, singularity and get seemingly correct results. SSE support both NaNs, infinities and denormalized numbers[1] , while AltiVec has a register to indicate if denormalized numbers should be supported.

The instruction sets have instructions for doing element-wise additions and

[1]Denormalized numbers are numbers so small that the can only be represented at the cost of lower precision, and they are therefore not very usefull.

multiplications, loading/storing instructions from continuous memory, instructions to shuffle elements of the vectors around, instructions for loading and storing single elements and so on.

4 CPU characteristics.

This section will describe some of the different characteristics of modern CPUs.

4.1 Caches.

A modern computer chip can calculate much faster, than it can retrieve the data from memory needed for these calculations. A rough estimate is, that the chip can do 50 - 150 operations while waiting to receive the data for one of them. The solution to this is to introduce some temporary, very fast storage between the processor and the memory, that holds the data you have previously accessed, so that if you need them again soon, they will still be there. Data is fetched from memory into the cache in so called "cache lines", that are the smallest amount of data you can transfer. It is 64 bytes on the Athlon and 32 on the Pentium. So even though you only want one byte, you might actually get 64, but if you access your memory linearly, you will probably need the bytes you just received.

4.2 Pipelines.

In order to speed up the actual calculation of e.g. an addition, the process is split up i many stages. The unit that does the addition will then be able to start on a new addition, as soon as the old addition has finished the first stage. This does not make each addition finish faster, but it makes it possible to work on many additions simultaneously. Modern computer chips can have very long pipelines. The Pentium 4 has a pipeline of 20 stages, meaning that it can start working on an addition each clock cycle, but it will take 20 clock cycles before the first one finishes.

A consequence of this is that if you need a result from an operation, you have to find something else to do while it is in the pipeline, and if you do a lot of things while you wait, you have to have registers enough to store all the temporary results! When you put an operation into the pipeline, you have to specify in which register to put the result, and this register is then unavailable while the operation passes through the pipeline. If the pipeline is long, and you have few registers, you might very soon run out of registers. Both the Athlon and the Pentium III only have 8 available registers, so on these architectures this can be a serious problem.

4.3 Branch prediction.

When you write code you will very soon need a branch in the code, this can be e.g. a conditional statement (if..then..else) or a loop (do ... while). Because of the pipelines, you might not yet know where to go when you reach the branching point. Instead of waiting for the result to pass through the pipelines, the CPU will try to guess where you are going. If it guesses right it will keep its pipelines filled and run fast, if it guesses wrongly it will have to discard all computations, clear the pipelines, and go back to the branching point! This is very time expensive.

Regular loops are easy to predict, and the prediction will probably only fail the very last time when the loop finishes, if it fails at all. Still, the processor might not be able to make a jump with the same ease as if it was sequential code, and one should therefore consider completely leaving out a small loop and just writing everything up several times.

The placement of the target of the jump may also have importance. When you arrive at your destination after the jump, the processor will load a cache line full of instructions from memory. If however, the place you jumped to is placed in the end of a cache line, you will do unnecessary loading of data, and you will soon have to load again. It might therefore be an advantage to place the jump target in the beginning of a cache line.

4.4 Execution units.

The execution unit is where the actual computation takes place, a CPU might have one or more execution units. The Athlon has to units, one for doing addition and one for doing multiplication. The Pentium III has one combined unit for both. However when you have a pipelined architecture, some of the last stages of the pipeline will be concerned with producing the output in the correct format, and these stages might not be needed if you immediately continue to use this result in the next computation. Therefore the processor can speed up calculations if it can find the dependencies and take advantage of them. The Pentium machines can exploit this, and send half-finished data from the multiply unit to the add unit, and this way save a few cycles.

4.5 Virtual registers.

Both the Pentium III and the Athlon only have 8 addressable registers. However they really have many more registers, maybe 60 - 80, I don't know the precise numbers, and the 8 addressable registers are then mapped to the real registers. This is called virtual registers and register renaming. The remapping may happen as soon as you write something to a register without reading from it. In that case you could actually be writing to a whole new register without changing the semantics of the code. The processor will discover this, assign you a fresh register, and change all subsequent accesses to this new register.

4.6 Memory access.

When accessing memory it is important that data is aligned, meaning that it is placed on a memory address divisible by the length of the data. If you for example is accessing a 4 byte single precision word the access will be fastest if the data is placed on a address divisible by 4. If it is misaligned it will take longer time depending on how misaligned the data actually was.

When using vector registers this becomes an even bigger problem, since the data operated on is now larger. On the Pentium III with its 4 elements vector registers, access is fastest if they are aligned on 16 bytes boundaries. The Pentium has two instructions for loading data, a fast one that will only load aligned data, and a slower one that will load misaligned data as well. So in this case, not only should your data be aligned properly, but you should also *know* that it is aligned properly. The problem is not as big on the Athlon, here you can run at full speed if your data is aligned, but without specifically knowing it. On the AltiVec architecture a misaligned load is simply not possible! You must load into several registers and then shift your result into place!

4.7 Indirect addressing.

When coding the matrix multiply, one will very soon need some sort of indirect addressing. This means, that you do not request memory by giving the CPU the address of the memory, but rather by giving a base address and an offset plus another offset times a stride. Example: If you wish to get the element a_{23} of the A matrix you will request the element at the address $pA + 2 + 3 \cdot lda$, where pA is the address of the first element of A, and lda is the number of elements between the first element of two consecutive columns of A, known as the leading dimension of A.

The consequence of this is, that in order to get the correct memory address you need to do two integer additions and one integer multiplication. Modern CPUs have special units for doing this kind of computations, but they are pipelined, so it will take some time before the correct address has been calculated. This greatly increases the total time it takes to calculate one instruction. Fortunately, since matrix multiplication is so regular, many of the computations can be done in advance, thereby reducing the impact of this problem.

4.8 Instruction reordering.

In order for the CPU to execute as much code as possible in parallel in its different execution units, the CPU will look ahead and see if it can execute future instructions now. It will reorder the instructions so that they are executed most efficiently without changing the semantics of the program. Modern processors can have quite large look-ahead buffers. The Athlon can look at up to 72 instructions and determine which ones to execute next.

One could now think that it was no longer necessary to feed the instructions in the most optimal order, but that is still necessary. Now the optimal order just depends on how the processor will most easy recognize the parallelism in the program. This can lead to quite funny, primitive or contra-intuitive layouts of the instructions. Perfectly scheduled code might run slowly, and seemingly huge stalls in the pipeline might execute most efficiently, because the processor is optimized for correcting the pipeline stalls.

4.9 CISC architectures.

CISC is an abbreviation for “Complex Instruction Set Architecture” and it is and old method of designing instruction sets that still exists today for backwards compatibility issues. The general idea in CISC architectures is to have a rich instruction set so that it is possible to express the program in few, possibly rather complicated instructions. Furthermore these instructions can adress the memory in many ways, for example in these ways:

- The data needed is encoded into the instruction.

- The data needed is contained in a register.
- The data needed is pointed to by a register.
- The data needed is pointed to by a register plus an offset that is encoded into the instruction
- The data needed is pointed to by a register plus a leading dimension encoded into the instruction, times the value of a second register.

Since CISC was designed when the memory sizes were small the lengths of the instructions wary according to the complexity of the instructions and the addres mode used.

The modern replacement for this paradigm is called RISC, "Reduced Instruction Set Architecture". When using this design all instructions have the same length.

All modern CISC CPUs use a RISC-like internal instruction set, and decode the CISC instructions into RISC instructions. Some CISC instructions take a long time to decode, but these are easily avoided, so they are not a problem. The real bottleneck of CISC is that the CPU has to partially decode an instruction to determine the length of the instruction, before it can start decoding the next instruction. Therefore a substantial part of a CISC chip is devoted to decoding the instructions fast enough, and a program should be optimized to be as easily decoded as possible.

One way this problem can show up is that it is very difficult to align the instructions correctly. The instruction cache is seperated into cache lines, but an instruction might easily be placed on the boundary of two cache lines, making it much harder to decode, because the chip might have to wait for the cache subsystem in the middle of decoding the instruction. The problem is very hard to avoid since instructions have warying lengths: Even though the first instruction is perfectly aligned in the beginning of a cache line, the alignment is completely lost after just a few instructions.

5 Coding techniques.

This section will describe various techniques to make fast running code.

5.1 Cache blocking.

The main trick to utilize the caches is to block the problem. This mean,that instead of trying to do the whole matrix multiplication at once, the problem is split up in submatrices that will fit in cache. One or more of these submatrices can then be kept in cache, reducing the amount of traffic to the main memory. Often it does not pay of to have a submatrix of C, A and B in the cache at the same time, but only to keep one of them, normally A, completely contained in cache, and then load small panels of C and B into the cache.

When writing a user supplied kernel to ATLAS, all this work has already been done for you, and the submatrices have been copied to continuous memory. This means additional memory transfers, so the problem has to be big enough to justify it, but ATLAS runs tests to determine when this happens, and will call other code if it is not feasible to do the memory copying. You can therefore safely assume that the A and B submatrices are stored in contiguous memory. However, this is not the case for C, for several reasons.

The main reason is that ATLAS emphasizes that you only write an element in C once. This can be done by doing the usual "inner-product" way of doing matrix-multiplication, as it is first formulated in any textbook. As you only write it once it would be of very little advantage to first store it in memory with a known leading dimension. Storing it directly to the correct place is just as fast. The second reason is that you can easily end up in a situation where m and n is big, but k is very small, meaning that A and B are very rectangular, and you are almost doing an outer product. This would make the C matrix very big, and therefore make the memory-copy cost very high, but the actual number of computations would not be high. In that case, making a copy of C would hurt performance badly.

5.2 Looping structures.

ATLAS uses the $A^T B$ case for doing the computations. If other variants are needed, like AB or AB^T, the data from A and B is simply transposed accordingly while being copied. The big advantage is that A and B now have the same leading dimension: k. Data is stored in the same format as the Fortran programming language uses, column major format. That means that the element b_{ij} is followed in memory by $b_{i+1,j}$. Doing an $A^T B$ matrix multiplication kernel to use with ATLAS might look like this:

```
for (j=0;j<NB;j++)
{
   for (i=0;i<MB;i++)
   {
      for (k=0;j<KB;k++)
      {
         C[i+j*ldc]+=A[k+i*KB]*B[k+j*KB]
      }
   }
}
```

There are several things worth noticing here. First of all `MB`, `NB` and `KB` are the numbers of elements along the m, n and k dimension, however, these values are known at the time of compilation of the program. It means that this code, when compiled, will only work for matrices of one specific size. However, it will be an advantage when doing loop unrolling.

This looping structure is not very good, since we can use the variables much better; we do too many and complicated address calculations to index the matrices. There is also the issue of branch prediction. Modern processors try to guess which direction a jump will take, and since loops are common, the processor will predict that a backwards jump will almost always be taken. With these loops however, the check is in the beginning of the loop, and if the compiler is not good enough it might not the optimal jumps. A better looping structure would be:

```
stM=A+MB*KB;
stN=B+NB*KB;
```

```
pA0=A;
pB0=B;
do
{
   do
   {
      for (k=0;j<KB;k++)
      {
         pC0[0]+=pA0[0]*pB0[0];
         pA0+=1;
         pB0+=1;
      }
      pA0+=0;
      pB0+=-KB
      pC0+=1;
   }
   while (pA0!=stM);

   pA0+=-MB*KB;
   pB0+=KB;
   pC0+=-MB+ldc;
}
while (pB0!=stN);
```

This example looks stupid because the problem is so simple, but this is the general skeleton for all the generated codes. The big advantage is that A, B and C can now be addressed much simpler. The loops stop when the pointers `pA0` and `pB0` reach the end of each matrix. The innermost loop now consists of two additional instructions to advance the pointer, and it looks like much in this example, but the costs vanishes as the innermost loop is unrolled. The "+0" statements is just to indicate that they are needed for more complicated loop unrolling and register blocks.

The innermost loop is still a for-loop since all compilers should be able to interpret this to generate the fastest code. However, it can always be exchanged with a do-while loop if needed, since the loop-variable is never used for anything than the loop.

5.3 Register blocking.

Register blocking is a technique used to reuse the data stored in registers as much as possible. This is done by doing the matrix computation with small outer products. The advantage of an outer product is, that if you have m registers holding values of A and n registers holder values of B, you will be able to do $m \cdot n$ multiplications and additions. However, it requires that you can also fit in mn registers to hold the values of C, and also a few registers to hold intermediate values. Unfortunately, both the Pentium III and the Athlon only has eight registers, so it is only possible to use this method with something like 2×2 and 4×1 blocks. This can however still be a gain since at least one register can be reused a number of times.

5.4 Loop unrolling.

Loop unrolling is a technique used for several reasons, where the primary reason is to avoid the cost of jumps. The idea is to simply duplicate the code inside the loop, and then do fewer iterations of the loop. An example:

```
for (k=0;k<40;k+=4)
{
   vec_mov_mr(pB0+k,reg0);
   vec_mov_mr(pA0+k,reg2);
   vec_mul_rr(reg0,reg2);
   vec_add_rr(reg2,reg6);
   vec_mov_mr(pA0+k+KB,reg3);
   vec_mul_rr(reg0,reg3);
   vec_add_rr(reg3,reg7);
}
```

This code describes the innermost loop of what could be the basic building block for an ATLAS matrix-multiply routine. I takes an element of B, multiplies it with two different elements of A and adds the results to some registers, that can be used to accumulate a value for C. In matrix notation:

$$\begin{aligned} c_{11} &= c_{11} + a_{11}b_{11} \\ c_{21} &= c_{21} + a_{12}b_{11} \end{aligned}$$

This is code for the $A^T B$ case, and the leading dimension of A is KB, so the reference to `pA0+k+KB` refers to the element $a_{k,1}$. Unrolling this code by 2 would yield:

```
for (k=0;k<40;k+=8)
{
   vec_mov_mr(pB0+k,reg0);
   vec_mov_mr(pA0+k,reg2);
   vec_mul_rr(reg0,reg2);
   vec_add_rr(reg2,reg6);
   vec_mov_mr(pA0+k+4,reg2);
   vec_mul_rr(reg0,reg3);
   vec_add_rr(reg3,reg7);
   vec_mov_mr(pB0+k+4,reg0);
   vec_mov_mr(pA0+k+4,reg2);
   vec_mul_rr(reg0,reg2);
   vec_add_rr(reg2,reg6);
   vec_mov_mr(pA0+k+4+KB,reg3);
   vec_mul_rr(reg0,reg3);
   vec_add_rr(reg3,reg7);
}
```

The first loop would run for 10 iterations, since we are increasing k by 4 for each iteration. The second loop would only have to run for 5 iterations, and we would therefore have less penalty for the jump. This disadvantage is that we now have more code, and that the original numbers of iterations would have to be divisible by 2. If this is not always the case, one will have to add additional code outside the loop to cope with this situation.

An additional advantage is that we can "roll" the registers. In the above code we used `reg1` and `reg2` to hold an intermediate result. This means that the first and the second part of the loop cannot be carried out in parallel, since they use the same register. The same holds true for `reg0`, that holds a value of B. If we have available registers, this can be changed to:

```
for (k=0;k<40;k+=8)
{
   vec_mov_mr(pB0+k,reg0);
   vec_mov_mr(pA0+k,reg2);
   vec_mul_rr(reg0,reg2);
   vec_add_rr(reg2,reg6);
   vec_mov_mr(pA0+k+4,reg3);
   vec_mul_rr(reg0,reg3);
   vec_add_rr(reg3,reg7);
   vec_mov_mr(pB0+k+4,reg1);
   vec_mov_mr(pA0+k+4,reg4);
   vec_mul_rr(reg1,reg4);
   vec_add_rr(reg4,reg6);
   vec_mov_mr(pA0+k+4+KB,reg5);
   vec_mul_rr(reg1,reg5);
   vec_add_rr(reg5,reg7);
}
```

Now the first and second part uses different registers, so they can be carried out in parallel if the CPU is clever enough. In this code there is now four sections of code that can be executed in parallel, if the compiler is smart enough to see it. The biggest constraint is that we only add to two different output registers, so this can possibly inhibit parallelism.

5.5 Filling pipelines.

Consider again the first example code shown:

```
for (k=0;k<40;k+=4)
{
   vec_mov_mr(pB0+k,reg0);
   vec_mov_mr(pA0+k,reg2);
   vec_mul_rr(reg0,reg2);
   vec_add_rr(reg2,reg6);
   vec_mov_mr(pA0+k+KB,reg3);
   vec_mul_rr(reg0,reg3);
   vec_add_rr(reg3,reg7);
}
```

In order to fully utilize the computing pipelines, the CPU has to have some data ready. In this example we load the data immediately before use. This is not very efficient, since we cannot execute the calculations before the data is loaded, and it will take some time before the data is loaded, so the CPU stalls. In the beginning of the loop all registers and pipelines are empty, then the CPU loads data and does calculations, and in the end of the loop everything is emptied again. This creates a lot of stalls, because everything must be emptied and re-filled shortly afterwards.

If we only had one iteration of the loop, this was the only thing we could do. However, since we have several iterations of the loop, we pull one iteration outside the loop, and use this to rearrange the instructions:

```
vec_mov_mr(pB0+k,reg0);
vec_mov_mr(pA0+k,reg2);
vec_mul_rr(reg0,reg2);
vec_mov_mr(pA0+k+KB,reg3);
for (k=0;k<36;k+=4)
{
   vec_add_rr(reg2,reg6);
   vec_mul_rr(reg0,reg3);
   vec_add_rr(reg3,reg7);
   vec_mov_mr(pB0+k,reg0);
   vec_mov_mr(pA0+k,reg2);
   vec_mul_rr(reg0,reg2);
   vec_mov_mr(pA0+k+KB,reg3);
}
vec_add_rr(reg2,reg6);
vec_mul_rr(reg0,reg3);
vec_add_rr(reg3,reg7);
```

One iteration has now been pulled outside. The data loading instructions are placed in front of the loop, and the rest of the one iteration has been placed after the loop, and the loop now runs one iteration less. The instructions in front loads the data and the instructions in the end takes care of any remaining data, and this allows the loop in the middle to run with pipelines full all the times. The multiply instruction that was pulled in front of the loop also serves as a kind of loading instructions, since it supplies us with data for an addition.

As the code is presented here not much has changed, since we now flush and reload in the middle of the loop, but we now have much higher possibilities for reordering the instructions, since it is no longer critical to load the data. An example of a reordering could be:

```
vec_mov_mr(pB0+k,reg0);
vec_mov_mr(pA0+k,reg2);
vec_mul_rr(reg0,reg2);
vec_mov_mr(pA0+k+KB,reg3);
for (k=0;k<36;k+=4)
{
   vec_add_rr(reg2,reg6);
   vec_mov_mr(pA0+k,reg2);
   vec_mul_rr(reg0,reg3);
   vec_mov_mr(pB0+k,reg0);
   vec_add_rr(reg3,reg7);
   vec_mov_mr(pA0+k+KB,reg3);
   vec_mul_rr(reg0,reg2);
```

```
}
vec_add_rr(reg2,reg6);
vec_mul_rr(reg0,reg3);
vec_add_rr(reg3,reg7);
```

This code loads data into registers as soon as they are empty, so all registers are full all the time. Whether this is an advantage depends on the CPU. This code shows that we can start the loop with different instructions that does not have strong dependencies on each other, and that the registers can be kept full throughout the loop allowing for better scheduling and fewer stalls.

5.6 Scheduling techniques.

The question remains how to actually schedule the instructions, to choose in which order they should be laid out to get maximum performance of the CPU. This is quite difficult to say, since it depends heavily on each CPU. I will try to outline several techniques that can be used depending on characteristics in the CPU.

The most important thing is to create as much distance between references to a register as possible, since it will allow a result of a computation to flow through a pipeline. A typical use of a register is something like this:

```
vec_load_mr(pA0,reg2);
...
vec_mul_rr(reg6,reg2);
...
vec_add_rr(reg2,reg0);
...
vec_load_mr(pA0+4,reg2);
```

First a value is loaded into register 2, then it is multiplied by another register, added to a third and finally reloaded with a new value in the beginning of the next cycle. The idea is then to put as many other relevant instructions in place of the dots. However, depending on the CPU, it is not equally important to insert instructions in all the possible spaces in the instruction sequence, and this is the place where things get funny.

If the CPU has a calculation unit capable of inserting the result of a multiplication efficiently into the following addition, then the multiply and the add should be placed right next to each other. This is the case for the Pentium III. The reason that it is possible to get a speedup from this is, that some of the last stages of the pipeline are devoted to aligning the result correctly and converting from some internal format to the format stored in the register. This work can be saved if the result is passed straight on to the addition unit. It also requires that the CPU is able to look quite a bit ahead, since the combined multiply and addition will take some time to execute.

If the CPU does not have an execution unit capable of this, then the multiply and addition instructions should be placed as far apart as possible, since the multiplication pipeline is often the longest. This is true for the Athlon, which has completely separate multiplication and addition units.

Many CPUs have more registers than can be addressed by the instructions, and then rename the addressable registers to physical registers. This can make the picture look a lot better, since in the case above, when we load into the register the second time, we will actually be loading into a whole new register. This also means that there is no reasons for inserting instructions in between the addition and the load instruction, because there simply is no dependency between the instructions. This strategy has worked very well on the Athlon.

5.7 Other tricks.

A trick to get a longer multiply pipeline is to use the registers used to hold the values of A and B as working registers when the value they contain is no longer needed. As an example consider once again the very small example:

```
for (k=0;k<40;k+=4)
{
   vec_mov_mr(pB0+k,reg0);
   vec_mov_mr(pA0+k,reg2);
   vec_mul_rr(reg0,reg2);
   vec_add_rr(reg2,reg6);
   vec_mov_mr(pA0+k+KB,reg3);
   vec_mul_rr(reg0,reg3);
   vec_add_rr(reg3,reg7);
}
```

Here we use two working registers `reg2` and `reg3`. The trick is, that we don't need the value contained in `reg0` any more, so we could have used that one instead of `reg3`, like this:

```
for (k=0;k<40;k+=4)
{
   vec_mov_mr(pB0+k,reg0);
   vec_mov_mr(pA0+k,reg2);
   vec_mul_rr(reg0,reg2);
   vec_add_rr(reg2,reg6);
   vec_mul_mr(pA0+k+KB,reg0);
   vec_add_rr(reg3,reg7);
}
```

We actually save an instruction because the B value is already in the register. The multiplication is now done from memory instead of from a register, so the method requires that the instruction set can support this. However, if we had had a bigger example both the A and B values would have been kept in memory, and it would not have been a problem.

Sometimes this trick is a benefit, sometimes not. It does create a little bit more irregular code but uses the registers better.

5.8 Decoding optimizations.

This section is only relevant for CISC architectures.

6 Working with vector registers.

The blessing of having vector registers is that you can work on multiple values at the same time, the problem is how to actually fill the vectors with data. I have tried two different methods.

6.1 Vectors along the k-dimension.

The easiest thing to do when working with ATLAS is to place the vector along the k-dimension. An example is:

$$c_{11} = c_{11} + a_{11}b_{11} + a_{21}b_{21},$$

this calculation can be represented by vectors of length two, with the a-vector holding a_{11} and a_{21}, and the b-vector holding b_{11} and b_{21}. An element-vise multiplication will give a vector holding $a_{11}b_{11}$ and $a_{21}b_{21}$. Both these elements must be added to c_{11}, so an instruction or instruction sequence for summing a vector register is needed. The most efficient way to do it is to use a vector register to hold all these partial sums, and then sum this register only once in the end. The working register will contain the values like this:

$$w = \begin{pmatrix} a_{11}b_{11} + a_{31}b_{31} + ... \\ a_{21}b_{21} + a_{41}b_{41} + ... \end{pmatrix}$$

The downside of this method is that a whole vector register is used for storing what is essentially only one value. Furthermore these are the registers holding the values of C, and for an m times n outer-product blocking, this results in $m \times n$ registers not being fully utilized, so for big blocking sizes this might not be the method to use.

A big plus for this method is that it works very well with ATLAS. Since ATLAS always calculates $A^T B$ then the elements in the vectors are stored next to each other in memory, so they can be loaded together. Storing the results is done just one value at the time, which can work well with the way that ATLAS handles the complex case, where all elements of C needs to be stored apart from each other. Vectors can also be summed together, if the block size is correct, that is if the number of vectors that needs to be summed equals the vector length. In that case you can shuffle the elements in the vectors, add them together, shuffle again and so forth so you end up having all the sums stored in just one vector, that can be written in just one operation. This is illustrated on figure 1.

First the values pointed to by the dashed lines are swapped. This is done by making an intermediate copy of one of the two registers, and then using special instructions for moving data around, and finally the vectors are added together. Compared to summing the registers individually, this method only uses about half as many instructions.

6.2 Vectors along the m-dimension.

Aligning the registers along the m-dimension means doing something like this:

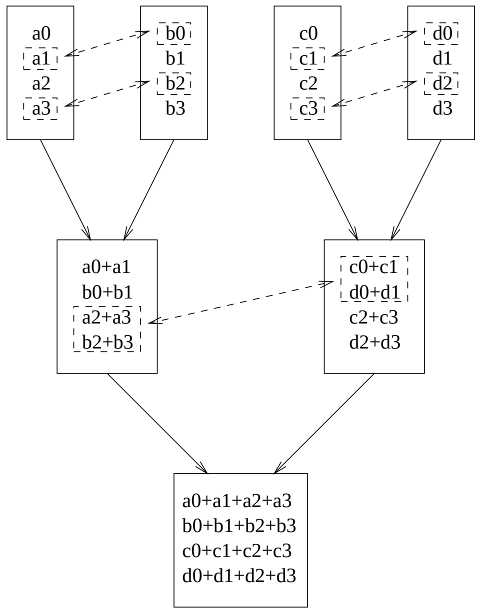

Figure 1: The summing of four vectors into one. The values in dashed boxes are swapped before summing the vectors.

$$
\begin{aligned}
c_{11} &= c_{11} + a_{11}b_{11} \\
c_{21} &= c_{21} + a_{12}b_{11},
\end{aligned}
$$

this is essentially a saxpy-operation. In this case the c-values and the a-values can be stored in vector registers, and the b value must be duplicated on all places in the vector register. The situation is the same as before in that we use a whole vector register to represent a single value, but in this case the number of affected register grows only linearly with the size of the dimension of the block, and not quadratic. A downside is that the duplicating of the elements must be done many times inside the inner loop, where as the summation in the previous example only needed to be done once.

Another serious downside is that this method works badly with ATLAS. Since ATLAS always calculates $A^T B$ then the elements in the a-vector are stored apart from each other, so special instructions must be issued to collect them into a vector. This makes the method hard to use since neither A or B can be read straight from memory. This problem goes away if this method is used for the GEMV $Ax = y$ problem. In this case A has the correct layout.

Storing C in the complex case will also be a problem since the vector needs to be split up, this is however a minor problem.

There are different strategies to load the A data:

1. Use a sequence of instructions that collect the elements needed to fill a vector. This gets overly complicated for long vectors.

2. Load veclen vectors normally to get a small veclen $\times$veclen matrix, and

then transpose this matrix through an instruction sequence. This should be more efficient than method 1, but it also takes up more registers.

Placing the vectors along the N-dimension will just be even worse than this example, the a values would have to be duplicated, the b-values would have to be loaded from different places in the memory, and the c-vector would have to be split up before it could be stored. No benefit at all.

7 Cleanup

What do you do when the problem size is not a multiple of the blocking size? ATLAS solves this problem by simply making smaller blocks along the edges, and then having special routines for handling these cases. This is referred to as the cleanup cases. The three most important ones are one of the m, n or k dimension being smaller than the blocking size and the other two dimensions being equal to the blocking size. The other cleanup cases are the situations where two dimensions at the same time are smaller than the blocking size, and finally where all three dimension are smaller.

The cases can be visualized as the sides of a cube, where the interior of the cube is handled by the normal kernel code, the faces are handle by the codes with one dimension smaller than the blocking size, and the edges and the corner by the remaining cleanup cases. For a big problem it is easily seen that the bulk of the computations are taking up by the kernel code and face cleanup code. For this reason it is only possible to supply codes for these problem to ATLAS. ATLAS has its own code generator to handle the remaining and less important cleanup cases.

The cleanup cases where M or N are smaller than the block size are rather trivial to implement because of the way that ATLAS transposes the A matrix. This results in the leading dimension of both A and B being K, and since K is fixed at the block size it is easy to unroll the innermost loop, and address calculations are also easy because one can use a fixed integer constant instead of a variable. The only difference to the kernel code is that one of the outermost loops must be able to run a variable number of times. This is most easily done by choosing a $mu \times 1$ register blocking for the N-cleanup and a $1 \times nu$ for the M-cleanup. Then the outermost loop will have to trouble with odd sizes blocks.

The resulting codes for M- and N-cleanup are usually almost as fast as the kernel code with a drop of at most 10% in performance.

The K-cleanup is quite different. This time the leading dimension and the length of the innermost loop changes. This has some big consequences. It is now very hard to properly unroll the innermost loop, since one does not know how long it is going to be. This problem is solved in a brutal but effective way. The trick is to generate routines for all possible values of the K-dimension! This results in a number of routines equal to the blocking size where the length of the K-dimension is known in each of the routines, and they can therefore do effective loop unrolling. This approach also has the benefit of making the leading dimension be a constant as it is in the main kernel and the other two cleanup cases. However, our problems are not over.

Since the most succesfull method of using the vector registers is to place them along the k-dimension, it becomes hard to make the cleanup cases where K is not

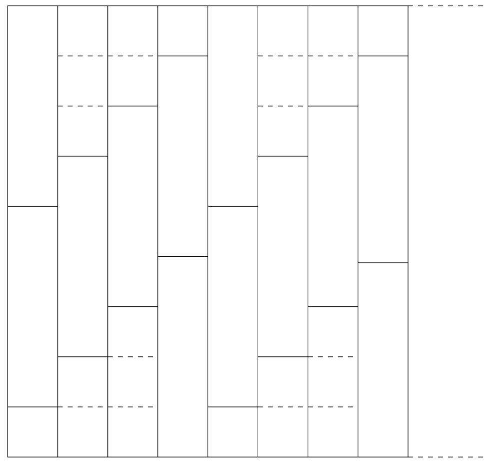

Figure 2: The first column of a K-cleanup of size 9 for a vector length of 4. The data in the large boxes is stored at an aligned position and can be loaded efficiently into vector registers. The data in the small boxes can be loaded into normal (scalar) registers.

a multiple of the vector length. This however can be overcome since the vector architectures usually have some support for using the vector registers with fewer (one) elements in them, or simply using normal floating points instructions for the last few elements along the k-dimension. The real problem is the alignment of data.

ATLAS copies both the A and B matrices to a buffer that is aligned at a multiple of the vector size, so this aligns the first column of each matrix. Unfortunately, since the leading dimension is no longer a multiple of the vector length, the second column will be misaligned! This prohibits the use of aligned loads and stores, and kills the performance so badly that I have not been able to make codes that are significantly better than ATLAS' code using normal instructions and registers.

Even though the columns of the matrices are not aligned, their starting position in memory is still known. Consider an example with a vector length of 4 and a K-dimension of 9. This is illustrated on figure 2.

The figure shows how to find vectors in the matrix stored at properly aligned positions. The remaining pieces of data in the beginning and end of a column will then have to be processed by using normal instructions or registers. Since the picture repeats itself for every 4 columns, one could make a code that processed the columns in an interleaved fashion.

Unfortunately, since both A and B are stored this way, and we need to multiply all columns in A with all columns in B, the alignments in the A-column and B-column will generally not be the same. The only thing to do is to use this scheme only for one of the matrices, and load the other one with unaligned loads. I have not had any great success with this approach, performance seems to be roughly the same as for not using the interleaved model, so I have not continued pursuing this approach because of the rather complex codes it produces.

The result of all this is that I have more or less given up on producing an efficient K-cleanup for the odd-sizes cases, I have only produced K-cleanup with a K-dimension divisible by the vector length. The remaining cases are then produced by ATLAS' internal code generator.

The problem can be solved by a completely different approach. Since the problem is the leading dimension, one could copy the data to a slightly bigger buffer where the leading dimension was a multiple of the vector length, and making sure the extra elements were all zero. That would make it possible to use aligned vector instructions all the way, but the approach involves changing ATLAS itself, so it has not been done or tried yet. When vector instructions become more important than they are today (and when someone has a lot of spare time), this might be the way to go.

8 The scheduling algorithm.

I have developed a simple algorithm to place instructions that use the same register as far apart from each other as possible.

The basic information required for this is to know how the instructions depend on each other. Consider the following table:

Nr: (i)	Instruction	Depends on:	Dist.: $d(i)$
0	`vec_mov_mr(pB0+k,reg0);`	5	2
1	`vec_mov_mr(pA0+k,reg2);`	3	5
2	`vec_mul_rr(reg0,reg2);`	1	1
3	`vec_add_rr(reg2,reg6);`	2	1
4	`vec_mov_mr(pA0+k+KB,reg3);`	6	6
5	`vec_mul_rr(reg0,reg3);`	4	1
6	`vec_add_rr(reg3,reg7);`	5	1

This is the previously seen instruction sequence numbered from 0 to 6. For each instruction, the third column list the previous instruction that this instruction 'depends' on. By this I mean the closest previous instruction that use a common register. This is a rather primitive notion, since instructions can depend on the result from several previous instructions, and two instructions that reads from the same register does not really depend on each other. However its enough to ensure the semantics of the program, if an instruction is never swapped with the instruction it depends on. Note that the dependencies wrap around, since the code is to be executed in a loop. For example, instruction 0 does not really depend on instruction 2, but if the code was executed in a loop, then instruction 0 would have to wait for instruction 2 from the previous iteration to finish using the register.

The last column shows the distance $d(i)$ between instruction i and the instruction it depends on. If this number is low, it means that the sequence will probably stall the instruction pipeline, because the second instruction will have to wait before the first finishes. The best thing one can hope for is that all the distances are more and less equal, that is: Maximizing the minimum distance between the instructions. This is the main algorithm for the scheduling of the instructions.

I have devised a primitive algorithm for maximizing the minimum distance. I cannot prove that the algorithm finds anything else than a suboptimal solution, but it seems to produce nice solutions, so it is a working and adequate solution to the problem.

The basic idea of the algorithm is to search through all permutations where one instruction is moved and choose the one that maximizes the minimum distance. Then the instruction sequence is altered by this permutation, and a new search is done, until a permutation is chosen that cancels the previous permutation. Then the algorithm stops. I have not proven that this stopping criteria will work in all cases, but I have yet to see it fail. The idea is that the algorithm reaches a point where the smartest thing to do is to go back. It might be possible to go back through a cycle of length longer than two.

The algorithm only considers valid permutations, but it is easy to see which permutations are valid from the distance information: A permutation of the instructions i and j with distance $dist = j - i$ is only valid if $d(j) > dist$, which simply means that the second instruction in the swap does not depend on any instructions in between the two swapped instructions or the first instruction.

The algorithm actually works on group of instructions rather than the instructions itself. This is done to be able to group multiply and add's together, and multiply and loads together. Whether this is efficient depends on the architecture, but the goal of the algorithm is to spread dependencies as these as far apart as possible, and the grouping into non-splittable groups is therefore necessary. It complicates the algorithm a little bit, but not much. Instead of maximizing the distance between instruction groups, the algorithm instead instead maximizes a weight function, which is the actual distance between instructions in the groups, and not the distance between the groups themselves.

I have not been able to calculate the asymptotic running time of the algorithm too well, but it seems quite high. Searching through all permutations is of the order seems at first to be $O(n^2)$, but if one considers the above code sequence, there are only two valid permutations $(0, 1)$ and (3,4) so the order is probably much lower. Calculating the minimal distance for each permutation is of the order $O(n)$, so each step of the algorithm is of the order $O(n^3)$, and all together I would guess that it was an $O(n^4)$ algorithm, but this does not seem to be an asymptotically close bound. I have tried simpler and faster algorithms, but they have failed to find an adequate solution for certain cases. One algorithm I have dismissed was to only consider permutations of adjacent instructions, but this algorithm easily winds up in a pretty bad local maximum. The current algorithm is a little more robust, but suffers from the same weakness, and it will sometimes fail to converge to anything sensible. A full blown branch-and-bound solution to the problem might be the only way to go in all cases, but I have yet to implement it.

It can be discussed whether this idea for scheduling the instructions is the correct one. This approach only considers the registers, and there might be more important resources in the CPU that would be a constraint, e.g, the multiplication or addition units. For example, the Athlon has a multiply and addition and a loading unit, so issuing the instructions in groups of three instructions (load,mul,add) to keep the units filled might be more important than spreading out instructions. The pipelines in the CPU could also have different length, so that it was more important placing a multiply and and add instruction far from each other, than placing a loading instruction and a multiply instruction far from each other.

These obstacles could be overcome by building a model of the CPU in the code generator complete with different units and latencies for these units, and then letting this model choose the next instruction in the sequence based on what resources in the model was available, or otherwise waiting a symbolic 'tick' until a resource is available.

I have tried building a model like this, but I had little luck because it generated irregular code, so I abandoned the idea. The current algorithm of spreading the instructions places all irregularities in a nice pattern, so that all trouble spots in the code a placed at some regular distance from each other. I believe that this is a better method than the other even though what might be the the most important aspects (the (load,mul,add) group on the Athlon for example) is obtained as a side effect of the algorithm rather as the goal of the algorithm.

9 Generally about testing.

The parameter space of the code generator is quite large, and therefore it takes quite a big search of the parameter space to find the fastest solution, and it is not always well defined, since many different possibilities will run at nearly the same speed. Furthermore, one cannot really assume that any parameters are independent of each other, or that the parameter space is in any way konvekse, so there seems to be no easy way to search for a solution other that to search all posibillities.

One important simplification can be made however: The loop unrolling of the innermost loop and the block size should be almost independent of the register blocking and instruction schedulling. ATLAS exploits this in its own search routine by alternatingly searching for blocking size and register blockings until they both converge, which should be at most two iterations.

The scheduling of the instructions should also be almost independent of the register blocking, but since I have had to develop a specific method of schedulling for each major platform (AMD, Intel), this does not say much. The schedulling method for the Intel PIII was found by looking a the fastest generated code at that time, and discovering that the speed came from a very perculier arrangement of the instructions, and then trying to implement this scheduling for all cases.

As a result of this I have generally tested all platforms with all possible configurations, or just used my own knowledge of the tested platform to speed up the search.

To present my results and the capabilities/deficiencies of each platform I have assembled a number of illustrative graphs. The graphs have not been used to find the fastest solution, they are only usefull as a demonstration. The problem is that there is so many limiting factors in the computer architecture, both some known to me, and some unkown to me, that it is only meaningfull to plot a section of the parameter space including the fastest solution, since this will eliminate as many limiting factors as possible. All plots therefore contain the fastest solution, and the parameters not shown on the graph will be those of the fastest solution.

10 AMD Athlon Thunderbird 1000 MHz.

10.1 Description.

The code generator was originally developed and tested on this machine, so this is the architecture that has been most well tested. The CPU has three separate floating point units: addition, multiplication and load/store.

The Athlon comes has been produced with three different cores:

- The Classic Athlon (K7): This core has a off-die level 2 cache of 512kb connected to the chip by a dedicated bus.
- The Thunderbird core: This core has an on-die level 2 cache of 256. The cache is exclusive, meaning that data in the L1 cache is not duplicated in the L2 cache.
- The Palomino core: This core has the same caches as the Thunderbird core. It has hardware prefetch and supports SSE instructions.

Hardware prefetch means that the CPU will try to guess which data will be needed next and fetch it before it is needed. It should work well if the data has a regular structure, but it requires a high performance memory bus since the amount of data transferred on the bus will increase.

The Athlon decodes CISC instructions into its own internal format using three separate instruction decodes. They operate in parallel and can decode three instructions per clock cycle under optimal circumstances. The decoders will block if they encounter a complex CISC instructions or an instruction that crosses a cache line boundary and spend a whole cycle decoding just this one instruction. The complex instructions are described in [bib:Athlon] and are easily avoided. The cache line crossing problem is much more serious and hard to avoid since x86 instrutions can have varying lengths. A cache line is only 8 bytes long and a typical instruction is 2-5 bytes long for the x87 double precision instruction set and 4-8 bytes long for the 3DNow! instruction set, so this can be very inhibiting to performance.

10.2 Single precision results.

, so the code runs faster if the multiply is scheduled away from the add, so that it will have time to pass through the multiply pipeline completely.

The fastest code used a 2×2 blocking with a multiply pipeline length of 4. This means that neither the values of A nor B can be copied into registers, there is only room for the 4 registers to hold the values of C and the 4 registers for the multiply pipeline. Both the values of A and B must be loaded several times straight from the level 1 cache, so there is no register reuse at all! The only explanation I can give is that the Athlon must have some unknown logic to recognize the repeated loads of the same value and just reload it from some of its many hidden registers.

The innermost loop is unrolled to a length of 16, with code outside the loop corresponding to a length of 4, so this code can do block sizes of $16n + 4$, $n > 0$, the fastest being a block size of 100. The program was considerably slower if the code outside the loop was of the same size as the code inside the loop, which

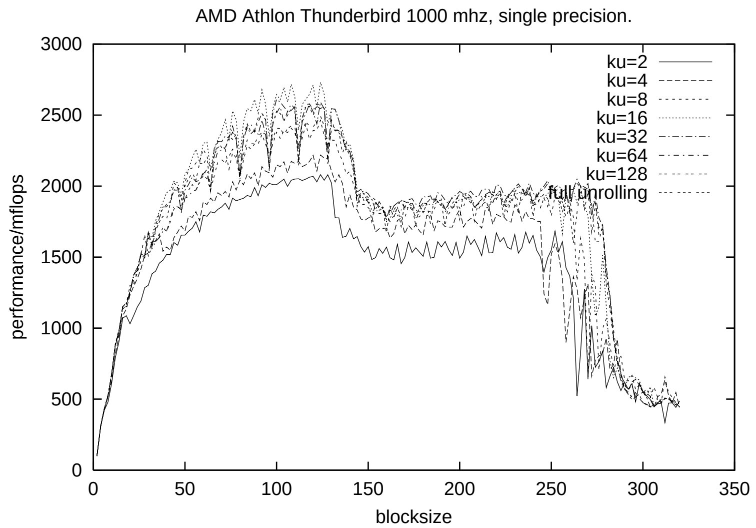

Figure 3: Different blocking and unrollings for the AMD Athlon.

is rather strange. I would guess that it comes from alignment issues with the cache: If the code outside the loop does a block of length 16, then the whole block size must be divisible by 16. Since both A and B are copied by ATLAS to continuous memory in the block size, two adjacent elements in the same row of the matrices (e.g. a_{11} and a_{12}) will end up being placed exactly $nb \cdot 4$ bytes apart in memory, where nb is the block size and the factor 4 comes from the length of the data type 'single'. Since nb is divisible by 16, $nb \cdot 4$ will be divisible by 64, with is the cache line length for this CPU, so one could guess that the CPU would map row-adjacent elements into the same place in the cache.

A plot off all considered unrollings and blocksizes can be seen in figure 3. The structure of the caches is clearly visible. Perfomance first drops dramatically when a whole block is unable to fit into the level 1 cache at a block size of 128. The second drop occurs when a whole block is unable to fit into the level 2 cache, and data has to be loaded from the main memory.

The associativity problems of the cache are also clearly visible: Performance drops for all block sizes that are a multipla of 16, and peaks for modulo 4, 8 and 12. The effect is only really visible when performance exceeds 1500-2000 mflops.

Another strange thing about the code is that the length in bytes of the code seems to give it special properties. If one tries to insert a variable number NOP[2] instructions in front of the innermost loop, the code will drop/rise drastically in performance, as much as 500 mflops. This does not happen quite as drastically for other codes, or from other minor variations of this code. My guess is that the code has the property that all the code is aligned perfectly on the cache lines, so that no instruction cross a cache line, and that the beginning of the loop is also placed in the beginning of a cache line. The latter is insured by

[2]A NOP instruction does nothing at all, and has a length of 1 byte, so they are effective for aligning the code differently. The Athlon processor can execute 5 NOPs per clockcycle, so they take very little time to execute.

putting an assembly macro in front of the loop that aligns the loop correctly.

11 Intel Pentium III Classic 600 MHz.

11.1 Description.

The Pentium III comes in two flavours:

- The Classic PIII: This core has an off-die level 2 cache of 512kb connected to the CPU by a dedicated bus.
- The Coppermine core: This core has an on-die level 2 cache of 256kb.

The CPU has a combined unit for doing additions and multiplication and it has some logic to pipe results from a multiplication straight into an addition there by saving some stages in the pipeline and speeding up the process a little bit.

11.2 Single precision results.

For the Pentium III I found two different solutions peaking in speed under different conditions. The normal way ATLAS tests code is to do extensive flushing of the L2 cache to create a realistic environment, but it is also possible to test the code with no flushing at all, which can be very helpful if one is just testing how good the code is pipelined, and how fast the code can run. Normally the same piece of code wins both tests, but not in this case.

The first code uses a 2×1 block with the k-loop fully unrolled and a pipeline length of 3 using the trick described in section 5.7. Because the vector length is 4 there is not enough elements along the m-dimension to do the reduction sum described in last part of section 6.1, so each register will have to be summed separately. This code is fastest when the cache is flushed all the time.

The second code uses a 4×1 block with the k-loop fully unrolled and a pipeline length of 2 using the same register saving trick. In this case the reduction sum can be applied for a more efficient summing of the vectors. This code is fastest when the cache is not flushed, it can simply attain a higher speed under optimal circumstances than the first code, however the first code seems faster under more realistic conditions.

When handling the cleanup cases where the k-dimension is much smaller than the block size, efficient summing of the vectors becomes much more important. Because the vectors registers are placed along the k-dimension it takes very little code to handle a k-dimension size of 4, 8 or 12 but it still takes the same amount of code to sum the registers. Therefore, in this case, the second code is much faster that the first one, because it only uses half as many instructions to sum the registers. I found the break even to be around 28, for blocksizes bigger than that the first code is the fastest.

A plot of block sizes and unrollings can be seen in figure 4. It is clearly seen that performance peaks with a fully unrolled innermost loop and a block size of 56. For bigger blocksizes the performance drops drastically, indicating that the matrix size exceeds the size of the cache. A block size of 64 corresponds to filling the level 1 cache of 16kb completely. This a very nice architecture in the sense that the fastest solution is well defined.

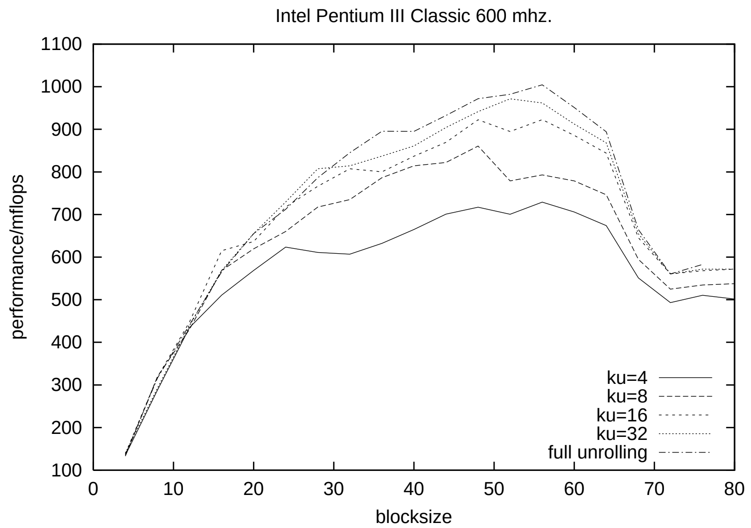

Figure 4: Different blocking sizes and unrolling for the Intel Pentium III

It should be noted that these results are specific for a classic Pentium III that has a separate level 2 cache not integerated with the chip. A Coppermine Pentium III with its integrated L2 will peak with a blocksize of 60 or 64, and would probably show a quite different plot.

12 AMD K6-2 300 MHz.

12.1 Description.

This is an rather old chip from AMD with a poor floating point unit, so it is very unlikely that it will ever be used for scientific computations. The only reason I have testet it is that it is my desktop machine and good testbed for the Athlon since it suffers from some of the same problems.

The CPU has two different computational units: One for doing x87 floating point code, both single- and double precision, and one for doing 3DNow! code. The x87 unit is not pipelined and very slow making it the achilles heel of this chip. The 3DNow! unit on the other hand is quite good considering the age of the chip.

The future for this chip might be in embedded applications where good single precision performance could have some impact.

12.2 Single precision.

The best code found was a 2×2 unrolling with a block size of 68. No for loop was used, the code was completely unrolled. The length of the multiply pipeline is 2, which means that the code keeps a small (2×1) block of both a and b in registers, and copy from this to the two working registers. Altogether 4 registers are used to hold the c-values, 2 registers are used to hold B, and 2 registers are used for the pipeline for a total of 8 registers.

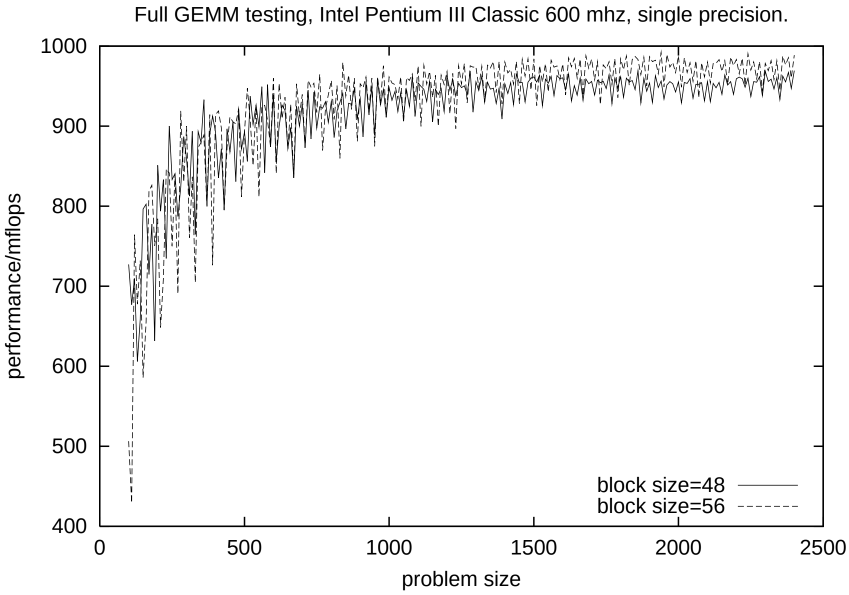

Figure 5: Full GEMM build and testet for differemt block sizes and problem sizes.

Without using vector code the processor has a performance of 83 mflops in single precision and with vector code the performance rises to 470 mflops, a factor of 5.6! This is no doubt due to the fact that two different parts of the chip are used.

A blocking size graph can be seen in figure 5. The maximum is not clearly defined, but a blocksize of 68 with full unrolling is one of the peaks. The bumps in the graph are probably just measuring errors. Performance drop drastically when exceding a block size of 84, which fits nicely with have one full matrix and a few panels of the other matrices. A block size of 90 would correspond to one matrix filling the 32kb level one cache completely.

13 Intel Pentium IV 1500 MHz.

13.1 Description.

The Pentium 4 has a very small and fast level 1 cache, only 8 kbytes, and a much larger level 2 cache integrated with the chip. This result of this is that the block does not fit in the level 1 cache, but instead fills up a part of the level 2 cache. On other architectures, the block usually almost fills up the level 1 cache, just leaving space for a few columns of the other matrices.

This is an instruction cache that contains the already decoded instructions in the order in which to execute them. This means, that after the first pass through the loop, the processor will have decoded the instructions and placed them in optimal ordering, so the second time through the loop they will execute at highest possible speed.

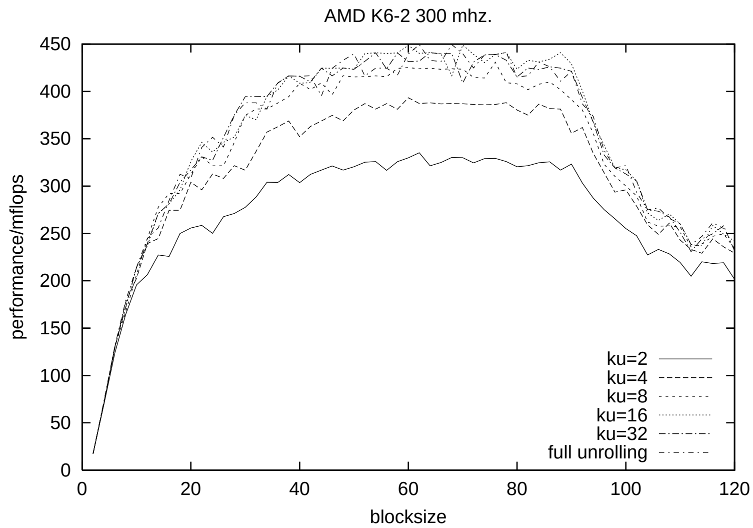

Figure 6: Different unrollings and blocking sizes for the AMD K6-2.

13.2 Thermal throtteling.

I had quite some trouble bechmarking the Pentium 4, until I discovered the real reason for my trouble: A Pentium 4 chip is equipped with a safety device that kicks in if the chip gets too hot, and makes the chip do nothing every other clock cycle, effectivily reducing it to half speed. When I ran a long series of tests, the first tests would run at the correct pace, but then the chip would get warm and the last tests would run at roughly $\frac{3}{4}$ the correct speed. This was fixed by making the chip wait a few second after each test, so the result displayed should be the correct ones. The effect can be seen in figure 6. The figure simply shows the same benchmark repeated 200 times and the $\frac{3}{4}$ slow down after 10 minutes is clearly visible. The problem was later fixed by attaching the heatsink properly, but a test like this should be conducted on any Pentium 4 to test if the cooling system works correctly.

13.3 Double precision.

The fastest code for this machine for double precision was a 1×4 unrolling, with a length of the innermost loop of 8. This is quite different from all other Pentium machines, because they always run best with the innermost loop fully unrolled. For the Pentium IV, a small unrolling is just as good. In fact block sizes ranging from 72 to 130 and unrollings of the innermost loop of 8, 16, 32 and fully unrolled, produced codes that ran at almost the same speed. Therefore I chose an unrolling of 8 and a block size of 80 to reduce the code size and to reduce the need for cleanup. This can be seen on figure 7. Notice the big dip at a block size of 128. This is probably alignment issues in the cache that kills performance. The curves for the four biggest unrollings are placed almost on top of each other. The performace gradually level off instead of dropping suddenly.

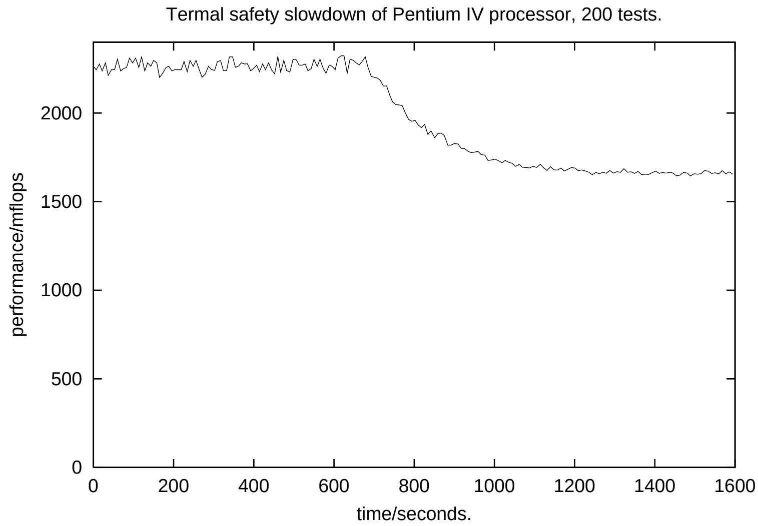

Figure 7: A test repeated 200 times on a Pentium IV processor. The processor gets hot after 10 minutes and starts slowing down. The test is simply benchmarking the fastest double precision code.

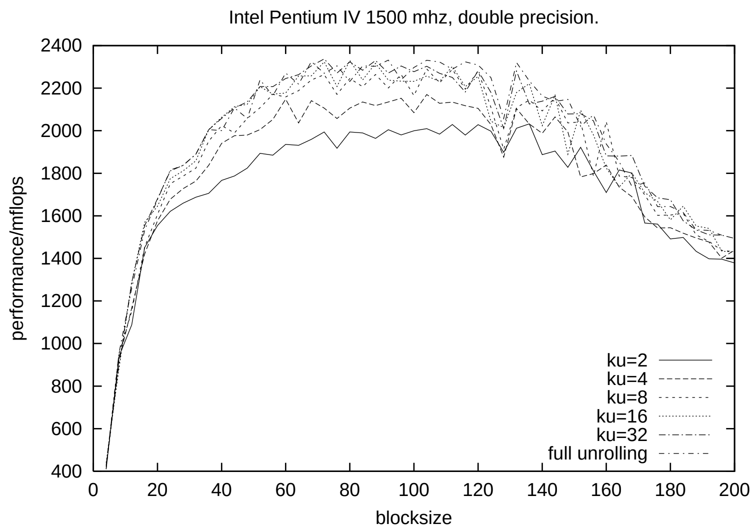

Figure 8: Block size and unrolling for the Intel Pentium 4, double precision.

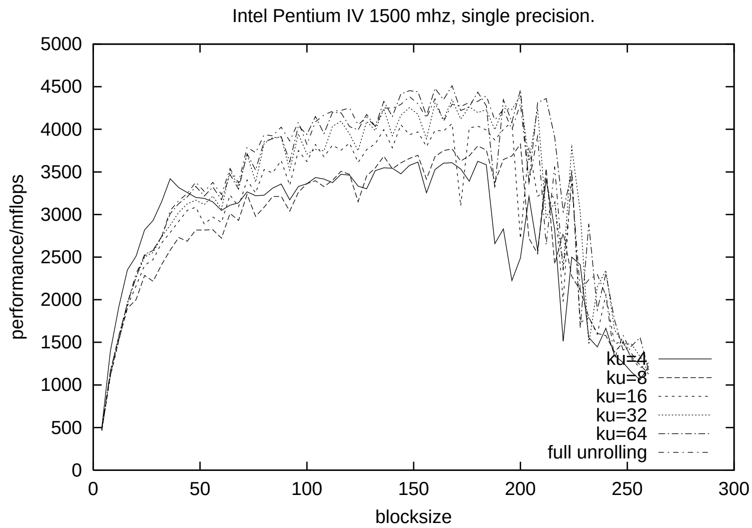

Figure 9: Unrollings and blocking sizes for the Pentium 4, single precision.

13.4 Single precision.

For single precision the picture can be seen in figure 8

The little need of unrolling is probably due to the caches of the Pentium 4, which are quite different from the other testet architectures.

I cannot say exactly why block size from 80 to around 130 is optimal. In that case a block takes up between 20% to 50% of the level 2 cache.

Another exceptional thing about this architecture is, that the scheduling of the instructions seems to have little to no influence on the speed, i.e. it does not matter in which order you issue the instructions, only which instructions you issue. This is very different from any of the other architectures tested. The explanation probably comes from the Pentium IV's so called 'trace cache'. This fits very well with the small unrolling of the k-loop. Processors without a trace cache has to decode instruction in all iterations of the loop.

The trace cache seems to benefit the Pentium IV in making it much easier to program.

14 Summary of test results.

The results for comparison:

Name:	Speed MHz.	Precision:	Block size	Regular Mflops	Vector Mflops
AMD Athlon	1000	Single	100	1250	2600
Intel PIII	600	Single	56/60	470	1000
AMD K6-2	300	Single	68	83	440
Intel P4	1500	Double	80	870	2300
		Single	80	1000	4600

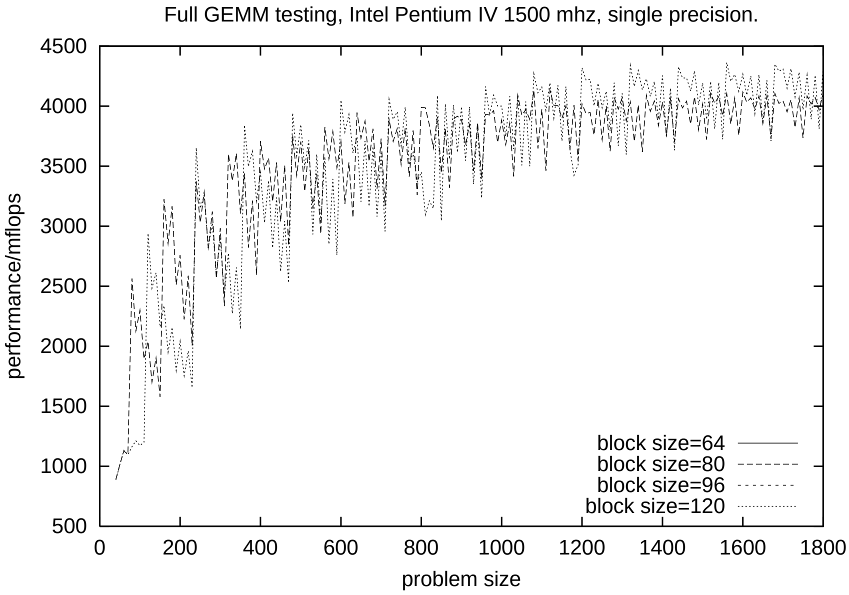

Figure 10: Full GEMM build and testet with different block sizes and problem sizes.

The reason for having two blocksizes for the Pentium III is that the first number applies to the Classic version and the second to the Coppermine version. The 'before' number for the Pentium 4 single precision is an estimate, I don't know the precise number.

Speedups are impressive, from a factor of 2 to a factor of 5! The choosen block sizes are not choosen to give the highest performace based on the tests I have performed, but rather choosen to give overall fastest performance. Therefore, some of the block size are smaller than what would be expected from the figures previously shown. A smaller block size reduces the need for cleanup code, and gives faster performance for small problem sizes.

15 Conclusion.

Using a code generator seems to be the way to go in the absence of optimized compilers for vector instructions. However the speed at which new instruction sets and CPUs are developed seems to be too fast for the compilers to keep up, so this might be the only solution for a long time. Language support is also a bit lacking, it is hard to specify in C or Fortran exactly how you want the vectors used, so using optimized libraries made by codegenerators or experts seems to be the way to go.

Using a code generator has also given me the extra benefit of being able to test the chip by generating programs with a changing parameter in order to test the influence of that parameter.

Coding the code generator in Python has been a rewarding experience. Python is a great language for manipulating text strings and making flexible code. Coding it in C would have been a horrible experience.

I have seen a great speedup in performance from using the vector instructions. The main reason for this is that matrix-matrix multiplication is dominated by the cost of calculations, and not the cost of moving data in memory. To expand the project to BLAS level 2 and 1 operations (matrix-vector and vector-vector operations) will be harder because the dominant cost will be data movement. The potential cure for this is to use the software prefetch instructions present in many modern computer architectures.

The project is far from over.

16 Literature.

The articles by R. Clint Whaley are available from the ATLAS distribution in the ATLAS/doc/ directory. All the technical documentation from Intel are available from their web site www.intel.com, similar all the AMD documentation are available from www.amd.com

For more documentation on the code generator and the source code, surf into www.cs.utk.edu/~soender/atlas/

References

[1] Whaley, R. Clint, "User contribution to ATLAS",atlas_contrib.ps

[2] Whaley, R. Clint et. al., "Automated Empirical Optimizations of Software and the ATLAS project",atlas_over.ps

[3] "Intel Architecture Optimization, Reference Manual", 24512701.pdf

[4] "AMD Extensions to the 3DNow! and MMX Instruction Sets",22466.pdf

[5] "AMD Athlon Processor, Technical Brief",22054.pdf

[6] "Intel Architecture Optimization Manual",24281603.pdf

[7] "IA-32 Intel Architecture Software Developer's Manual",24547001.pdf

[8] "IA-32 Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference (Pentium 3)", 24319102.pdf

[9] "IA-32 Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference (Pentium 4)", 24547101.pdf

[10] "Intel Pentium 3, Processor Optimization, Reference Manual",24896601.pdf

[11] "Intel Pentium 4, Processor Optimization, Reference Manual",24896601.pdf

[12] "3DNow! Technology Manual",21928.pdf

[bib:Athlon] "AMD Athlon Processor, x86 Code Optimization Guide",22007.pdf

[13] “3DNow! Instruction Porting Guide, Application Note”,22621.pdf

[14] “AltiVec Technology Programming Environments Manual”,altivec_pem.pdf