Contents

1 Introduction

This small paper will describe the various configuration options of the codegenerator. The codegenerator is invoked by the call `codegen.print_program(pars)`, where `pars` is a python mapping variable containing all the information on how to configure the program. The format is `pars['key']=value`, where `key` is always string and `value` can be different kinds of values.

2 Configuring the type of problem.

The key 'problem' determines which kind of problem can be solved. The following type exist.

2.1 pars['problem']='gemm'

This is the standard GEMM kernel for ATLAS. All dimensions are fixed at compile-time.

2.2 pars['problem']='gemm_m'

This is the GEMM kernel usefull for m-cleanup. All dimension are fixed except m.

2.3 pars['problem']='gemm_n'

This is the GEMM kernel usefull for n-cleanup. All dimension are fixed except n.

2.4 pars['problem']='gemm_k'

This is the GEMM kernel usefull for k-cleanup. This will generate an interleaved way of calculating the matrix product to try and use the data in an aligned manner.

2.5 pars['problem']='gemvt'

Level 2 BLAS GEMV transpose kernel.

2.6 pars['problem']='gemvn'

Level 2 BLAS GEMV no-transpose kernel.

3 Choosing the architecture.

The following architectures are supported.

3.1 pars['arch']='standard'

Setting the architecure to this will generate stardard c program code with separate multiply and add, and a vector length of 1. It will simply generate instructions like "`reg5 *= reg8;`", so it should work on any architecture with any c compiler.

3.2 pars['arch']='3dnow'

Support for AMD K6-2, K6-3 and Athlon 3DNow! ISA extensions. The datatype is single precision, 8 vector registers of length 2 are available. This architecture is now IEEE compliant to say the least, it has no support for Inf and NaN!

3.3 pars['arch']='sse'

Support for Intel PentiumIII and PentiumIV SSE ISA extensions. The datatype is single precision, 8 vector registers of length 4 are available.

3.4 pars['arch']='sse2'

Support for Intel PentiumIV SSE ISA extensions. The datatype is double precision, 8 vector registers of length2 are available. If you are working on a Pentium IV and want to work in single precision choose the 'sse' instead.

4 Choosing the method to use.

There are several different way of using vector registers when doing matrix computations, and the idea of this configuration options was to be able to choose between them. However, only open possibility has proven to be succesfull, so this one should always be used.

Both methods only work for separate multiply and add, combined multiply and add has yet to be implemented.

4.1 pars['method']='acc'

This option positions the data in the vector registers along the k-dimension, so the basis of a method will always be a $m \times n \times veclen$ building block. This has proven to be the must simple and fastest method, and therefore it is probably the only one fully implemented. The name comes from that the method requires the vector registers to be summed at the enddd of a computation, using fx. the pfacc instruction on the Athlon.

4.2 pars['method']='splay'

This method positions the data in the vector registers along the m-dimension. It is probably not fully implemented, and has yet to prove that it is usefull. From a theoretical point of view it could be very usefull when going to machines with more vector registers. For now, don't use it. The method requires the same piece of data to occupy all the positions in a vector register, which can be done with a 'splay' instruction on the AltiVec.

4.3 pars['method']='acc1'

This is a very inefficient method that is only used for cleanup. It is exactly the same as the 'acc' method, except it only uses one element in the vector registers.

5 Parameters describing the method.

These parameters describe the size of the basic building block. The reason why 'ku' is not in this list is that both methods have fixed 'ku' settings. The 'acc'-method uses a 'ku' equal to the vector length, and the 'splay'-method uses a 'ku' of 1.

The different methods will try to keep working copies of the A and B values in registers. However, if the parameters request a to long pipeline for all the things to stay in registers, the methods will try to load some values from memory directly if the architecture permits it. This might kill register reuse, but might also be efficient. If the the configuration requires to many registers, and there are no way around it, the program will throw a `MyError`.

If fewer register are requested than the number available, then the extra registers will not be used.

5.1 pars['mu']=n

The size of the building block along the m-dimension. For pars['problem']='gemm_m' this must be 1 and for pars['method']='splay' this must be dividable by the vector length. Otherwise, any whole number greater that zero will do.

5.2 pars['nu']=n

The size of the building block along the n-dimension. For pars['problem']='gemm_n' this must be 1, otherwise any whole number.

5.3 pars['pipelength']=n

The length of the multiplication pipeline. The length of the addition pipeline is determined by 'mu' and 'nu'. For pars['method']='acc' it is $mu \times nu$ and for pars['method']='splay' it is $\frac{mu}{veclen} nu$.

5.4 pars['loadfirst']='a' or 'b'

This parameter is only relevant for the pars['method']='acc' method. It describes which values are to be loaded first into the working registers, the values from A or from B. This has proven to be very effective to produced 'transposed' codes. For example the to codes produced with 'mu'=2 'nu'=1 'loadfirst'='a' and 'mu'=1 'nu'=2 'loadfirst'='b' will be very much alike and probably run at the same speed.

5.5 pars['lastuse']=0 or 1

This parameter is only relevant for the pars['method']='acc' method. When true it allows for some register optimizations. If f.x. pars['loadfirst']='a' and the values of a are kept in registers for register reuse, then the values of a are copied to the working registers before every multiplikation. If this parameter is true, then the method will use the a-register directly the last time it is to be used, instead of copying to the pipeline registers. This will make the pipeline longer, but it will also complicate the code of he structure and prohibit preloading of the a-register. Sometimes it can be a huge benefit.

6 Controlling the loop strucure.

The following are available for controlling the shape of the k-loop (the innermost loop).

6.1 pars['k_loop']='for','do' or None

This will determine whether the innermost loop is a for-loop or a do-loop. If the value is None, there will be no loop.

6.2 pars['ku']=n

This is only relevant if there is a k-loop. The parameter determines how much the innermost loop is unrolled. The value n must be divisable by the size of the basic building block along the k-dimension. This is usually the vector length.

6.3 pars['outside_len']=n or 'ku'

This parameter can either by a number or the string 'ku'. If the parameter has the value 'ku' it simply means the same value as pars['ku'].

This is the length of the k-dimension placed outside the loop. If there is no k-loop, then this must account for the whole unrolling along the k-dimension. The value can not be zero, since it includes the pipeline start and pipeline flush before and after the k-loop. The minimum value is therefore usually the vectorlength.

6.4 pars['align_jumps']=0 or 1

This will place an align() macro just before the innermost loop to align it to a certain boundary, so that the jump to the beginning of the loop will be to a nicely aligned adress.

7 Controlling the schedulling and loop splitting.

These parameters determine how the instructions are layed out, but do not produce instructions. They are therefore independant of the problem and the method.

7.1 pars['split']=0,1 or 2

This determines which instructions in the loop are used for pipeline startup and flusing. Three heuristics exists:

0: All instructions that load data are placed before the loop starts.

1: All instructions that load data, and half of the instrcuctions that process these data are placed before the loop starts.

2: All instructions that load data, and all instructions that operate on the data are placed before the loop starts.

For case 2 it probably sound like all instructions will be placed before the loop, but it is only the loading instructions and the instructions that use the loaded data, and not the instructions that uses the calcuted data, so it is only the two first steps in the dependency chain, and not the third step.

Generally heuristic 1 seems to be good, but there is rarely any big difference.

7.2 pars['sched']

This is a list of strings, where each string tell the scheduler to behave in that way. The options are

'spread' This tell the scheduler to use an algorithm for spreading instructions that depend on each other as far apart as possible.

'group' This tell the sceduler to group instructions that read from a register and subsequently write a value to the same register that is not dependant upon the previous value in the register. These groups of two instructions are not split apart by the option 'spread'. This reason for this option is that when the CPU uses virtual registers, then the second write operaton will write to a whole new register, and therefore the two instructions are independant of each other.

'fuse' This will collect add multiply and add instructions using the same register together in groups. This is efficient if the CPU has some internal mechanism to pipe the result from the multiplication directly into the addition unit.

Some combinations will probably not work, and the 'spread' scheduller is nowhere perfect so it may give bad results.

8 Special parameters.

These parameters are only valid for certain problems.

8.1 pars['m_cleanup']

This is only valid for the 'gemm_m' problem. This is a mapping type containg elements with the same name as in pars. If pars['mu'] > 1, then the 'gemm_m' class will produce some cleanup code to handle the remaing elements. The parameters stored in pars['m_cleanup'] will then override the regular values in pars.

For example, pars['mu']=4, pars['m_cleanup']={'pipelength':4} so cleanup must be used to handle the possible remaining last elements along the m-dimension. The value of 'method' stored in pars['m_cleanup'] will override the value stored in pars['method'], and the same situation with the 'pipelength' value. The cleanup code will therefore use a pipelength of 4 and all other parameters will be the same.

8.2 pars['n_cleanup']

This is exactly the same as pars['m_cleanup'], instead only relevant for the 'gemm_n' cleanup code.

8.3 pars['k_cleanup']

Same story as with the other cleanup cases, just a bit more complicated. Some additional parameters exists:

pars['k_cleanup']['mustfit']=n

This indicates the size of the k-dimension that the code must fit. It might also be usefull for other sizes, but it is guaranteed to fit this blocksize.

pars['k_cleanup']['compile']=0 or 1

This determines whether the size of the k-dimension will be a compile time constant or a run time variable. It is probably very inefficient setting this to 0.

8.4 pars['prefetch']=0 or 1

This determine if you want to use prefetching when doing GEMV. One additional parameter exist for this:

pars['fecthahead']=n

Determines how many cachelines to fetch ahead.

9 Miscellaneous parameters.

9.1 pars['atlasname']

This is a string indicating where to place the output files. The values can be e.g. 'Linux_PIII', 'Linux_ATHLON3DNow2'