Annex C

Journal of Development

C.1 Environmental Routine for Effective use of Cache, Pipelining and Registers

C.1.1 Introduction

It is well known that effective use of cache and registers can make a substantial difference in the performance of codes written to do the core algorithms of linear algebra. Existing computer languages could make it possible to write portable code without losing efficiency by providing a set of functions to probe the system parameters. A user would then construct a program to use the parameters to develop an optimal implementation on the target computer. With many optimizing compilers, a preprocessor could be used to get much of these benefits. It would not be expected to do as good a job, and would still require some selection of parameters for each new machine. The ideas apply to PC's, workstations, and to the code running on a single node of most new parallel machines.

Computers have been evolving in such a way that arithmetic speeds far exceed the rate with which operands from the main memory can be obtained for processing. Because it is cost effective to do so, many new parallel machines are using microprocessors for which this is the case. Since it is frequently the case that when something is used, it is going to be used again reasonably soon, systems are designed to use one or more levels of cache, (i.e. fast memory) where data used recently can be reused more quickly. This approach works quite well for most users, but when working with large dense matrices, data is never in the cache when needed. This can easily cost a factor of two or three in performance and in many cases significantly more.

We regard a computer's registers as a special case of a very small cache, since operations done on registers proceed more quickly than those using an operand that is in the fastest cache. (RISC architectures require that operands be in the registers.) But the differences between registers and cache memory require that different mechanisms be defined to make most effective use of both.

At every level of cache (including the registers), one wants to do as much computing as possible before the data in the cache is flushed for other data references. There is great scope for cleverness here in the design of algorithms. Since matrix multiplication is simple to understand we use it as an example to clarify how the ideas presented here might be used.

We believe that with the kind of features described here, it would not be difficult for those who know about such things to write compilers that would make it possible for those who know about such things to write portable code with no significant loss of efficiency. In addition it would be possible to write applications that can take advantage of these parameters and enhance the performance on a wide range of applications.

The most notable effort to deal with these problems is the Level 3 BLAS,[24] (it should be noted that we make no pretense to discussing all the factors that may be important to obtaining high performance on modern processors.

C.1.2 Language Extensions for the Cache

The cache system of a computer can be characterized by the following:

We expect $\sigma_L > \sigma_{L-1}$ and $\tau_L > \tau_{L-1}$, $L = 1, 2, \ldots, kache$, where we adopt the convention that L_0 corresponds to these values for the registers. We define σ_{kache} to be the largest amount of memory available, which in the case of virtual memory includes disk space. We make no further reference to the τ's, although there are certainly cases when such information might be of use.

In the case of Fortran, these values could be provided by environmental intrinsic functions, which take an argument of the type for which space is desired. In the case of C, these could be provided as part of the standard *math.h* header file. In the case of compilers being used in environments with different cache characteristics, it would be useful to have some means to specify

kache	The number of levels in the cache architecture.
σ_L	The size of the L^{th} cache.
τ_L	The access time to get data from the L^{th} cache divided by
	the time to copy one floating point register to another.
w_L	Number of consecutive items that get replaced when a new
	item is fetched to cache L.

this information in a configuration file. Finally if this were done using a preprocessor, one would provide the information for the system being used to the preprocessor.

Extensions that would allow one to gain the efficiency using the BLAS without writing machine specific code.

C.1.3 For Efficient LA Software

There are at least two things that might be done. Provide information through an interface about the cache structure of the machine, number of caches, their sizes, their access times, the size of the cache line, issues connected with pipelining, etc. Allow the programmer to declare variables and arrays (very small arrays, but still arrays) that are to be kept in registers. It is up to the compiler to pick the size of the arrays, and to unwind loops that refer to such "register" arrays. The compiler makes available the size of the arrays that it selects so that the programmer can reference them for purposes of writing loops, or for any other purpose.

C.1.4 Advantages of this approach

Code need only be written once, no dependence on computer vendors, or when a machine first comes out efficient implementations of important software will be available. If one has the stomach for it, it should be possible to write algorithms exactly as you would like to have them. Thus for example, one doesn't have to organize things to use a matrix multiplication if that is not the most effective way to do things.

C.1.5 Disadvantages of this approach

Code is significantly more difficult to write. It may require some cooperation from compiler vendors. What is proposed here can be done by a preprocessor if one can supply it with the cache information (probably not too hard) and the size that would be appropriate for the register arrays. This can be determined on a machine by machine basis by trying a few possibilities.

Characterizing a Cache System
The number of levels in the cache architecture.
The size of the L^{th} cache.
The access time to get data from the L^{th} cache divided by the time to copy one floating point register to another.
Number of consecutive items that get replaced when a new item is fetched to cache L.
Cache mapping: set associatively, direct

Machine Characteristics
Number of floating point registers
Number of floating point units
Number of caches
Cache size
Type of cache e.g. 2 way set associative, least recently used
Cache line size
Cycle time
Page size
Size of TLB (translation look-aside buffer)
Cycles for floating point operations, add, multiply, division
Number of processors
Fused floating point ops; multiply/add
Cycles from memory to stages in the cache
Pre-fetch and how many outstanding requests
Bandwidth from/to memory
Latency from memory/cache
Number of instructions issued per cycle

Five Parameters Associated with Memory Hierarchy
Access Time: Time for the CPU to fetch a value from memory – including delays through any intermediate levels.
Memory Size: The amount of memory of a given type in a system.
Cost Per Unit (byte): Cost per unit times size roughly equals total cost.
Transfer bandwidth: Units (bytes) per second transferred to the next level.
Unit of transfer: Number of units moved between adjacent levels in a single move.

Memory Hierarchy
Desired - no cost, very fast, large non-volatile
Actual Registers, cache, DRAM, Disk, tape, CD, Flash & EPROM
Registers - fast, local, volatile, VERY expensive, very small
Cache - fast, expensive, volatile, small
DRAM - medium size & speed, volatile
Distributed memory
Disk - slow, low cost per byte, non-volatile
Tape - very slow, very low cost, durable, non-volatile
CD - slow, “read only”, good data density, non-volatile
Flash & EPROM - small, not as fast a DRAM, non-volatile