something

Devin Bayly

Abstract

State of the art computational tools are in especially high demand in the field of Neuroscience. However, bottleneck exists in terms of how much data can be transferred between hard disk and memory for computation. As research increasingly relies on processing huge volumes of data, this issue demands attention. One strategy which addresses this is memory compression of data in memory (DRAM). Algorithms that are effective do a good job of decompressing exactly as much of the data as are needed for the calculations, so that they still minimize the memory footprint of the program without significant speed drops.

This project exists to address this general bottleneck in the context of the Neuromapp program created by members of the Blue Brain Project team. The project was is split into research and implementation of in-memory compression along the lines of compression library selection, interface algorithm development, and block data structure design. The resulting compression mini-app is intended to relieve this bottleneck, and provide accelerated calculation capacity for the suite of associated mini-apps that come with Neuromapp. It appears that the current implementation doesn't resolve the problem, but could present benefits under other performance metrics than runtime and bandwith which were considered here. (??)

Introduction

The compression library is nothing without a data object to apply it to, so let us briefly consider the block container. This data structure is Timothee Ewart's brain child, and i'll attempt to do just to the implementations description. At its core the block is a representation of a contiguous piece of memory. By adopting a convention that a unique block address can be described by the row position multiplied by the column position, we can transform a large 1 dimensional array of memory into a 2 dimensional representation; In this way the block may be treated like a general matrix. Worth noting upfront is the decision to embrace a policy design for the block, and its essential components. This means that a block particulars – compressor, allocator, or type – can be specified in the instantiation. There are two varieties of allocator: cstandard is for regular malloc and free memory allocation; align is for posix memory allocation. Zlib is our currently implemented compression policy, but others can be added using zlib policy as a template. The numeric value stored in the block may be any one of the c++ numeric types, and this is accomplished through standard templating terms. These items make up the foundational aspects of the block.

With this understanding in place, its worth discussing the functional capabilities of the block. The block supports basic IO using the c++ stream redirection operator << for output via ostream types, and input >> operations in relation to ifstream types. Continuing with STL related capabilities, the block also has a nested random access iterator class which enables more straightforward access to other tools in the STL, such as sorting. The compression capabilities of the block allow for one shot full block compression/uncompression using the zlib library. It is possible to to interrogate the block for information relating to its size, or current compression state. This layer is essential for using the block in higher level programs like those discussed below.

The block doesn't exist in a vacuum, therefore lets now discuss the various tools included in the compression mini-app that leverage the block. Following the established pattern, the compression mini-app features a command line program: **use code markdown** ./app compression –compression –file {file_arg} –split –sort –benchmark –stream_benchmark –kernel_measure are all options. The –compression flag runs a standard single file routine on the –file {file_arg} if provided. The –split routine may be added in to parse the double, or float numeric type into its binary representation sorted by category: sign, sign, … ,exponent, exponent, … ,mantissa, mantissa … . The –sort option orders all of the columns in the block based on the values specified in a particular row of the block. The –benchmark is the default, and follows through with a compression, split,and sort combo run on a default specified file. The –stream_benchmark option initiates the block hybrid of the STREAM_BENCHMARK bandwith measurement test designed by John McCalpin at University of Virginia. Last but not least, we have the –kernel_measure option which compares compression and non-compression performance changes as a function of increasing levels of computational complexity. At this point we have done an upper level tour of all of the functionality –both old and new– that comes with the block, and the compression mini-app.

techniques

This project has been a massive, and I mean massive learning experience. Tim joked with me once before the submission of proposals that “most self labeling proficient programmers are only fooling themselves.” Where I'm concerned, I would have to agree. This project has exposed me to whole new programming paradigms, and lower and higher level computational tasks than I had ever tangled with before. It makes sense to briefly visit these items in some detail before looking at the results they yielded.

In the life cycle of an implementation we must start with the build script, CMake, which was a completely unfamiliar tool for me. Currently the build is setup to disable the compression app by default because the Blue Queen (??) will likely experience problems with it. When enabled, Cmake for the compression mini-app can create a subdirectory in the ${Binary Root} (out of source build directory root) named neuromapp/compression/*. This file contains the compiled binaries necessary for running the compression mini-app, and sourcing input files from the neuromapp/test/block_data/* path.

Many of the methods used to create these binaries were also unfamiliar. Tim introduced me to policy design, and although it isn't second nature for me to think in those terms, I can appreciate the code-reuse options it presents. Now we have a framework for adding in additional compression libraries in the future which will be discussed later on. In implementing the splitting tool I had the chance to practice trait classing. Here we needed to express two varieties of union conversion structs: the uint32 variety corresponding to float type decimal conversion; and the alternative uint64 variety for teh double type decimal conversion. This is a natural example of where we benefit from defining a struct that at compile time has information relating to the constants, and types needed for the program. Before this project I was also unaware that you could convert decimals into binary number pieces for the sign, exponent, and mantissa. As discussed later, this was an important tool to design.

Time is often our enemy, but it was crucial to have it working for us in this project. I designed a timer tool that is part of the compression mini-app that is very useful for profiling small chunks of code. I was unfamiliar with the chrono library for c++, but now I understand how to calculate ranges, and output results with meaningful units.

The areas which made extensive use of the timer tool were the compression, kernel measure, and stream benchmark (WTW) routines. The zlib library is a classic piece of programming, and like an ancient sacred text, its documentation is dense. I only went so deep as to pick out the utility functions "compress/uncompress" **code** which are one-shot meaning they compress, and uncompress entirely. The stream benchmark is similar to the compression in that it deals with operations on a relatively low level. In the stream benchmark bandwith is composed of four canonical computations involving numeric containers (labeled A,B,C): the copy just asigns the contents of A to B; scale asigns scalar multiples of each element in A to B; add sums A and B elementwise and assigns to C; the triad assigns to C the elementwise sum of A and B multiplied by a scalar. The block hybrid uses A,B,C vectors containing 640 blocks of 8000 bytes each to bring the actual transfer sizes to reasonable levels. In each of the computations described above we loop over all of the vector positions using the timer to calculate run times for the whole set. Each calculation is performed on blocks that involve the compression routine, and those that don't for comparison. I'm glad to have had a chance to learn about the STREAM BENCHMARK (McCalpins version) because it appears to be a useful starting metric of system performance. The kernel measure is another example of comparing performance on calculations that involve between the compression routines and those that do not. Here three complexity levels are used for comparison: first level is a simple addition of ints, meant to be the least complex computation which is still a computation; the second level is meant to resemble the calculation that is performed in updating synapses PSP's using the Tsodyks-Markram model treating the blocks data as parameters for the main formulae; the third level is a Euler method for solving a differential equation where the equation has been specified as **latex** y^2 + 30*t arbitrarily. We use a vector of size 100 (??) containing file content based blocks, and report the times for each level, and compression/no-compression routine variety per block. This program (WTW) gave me experience writing functors with the added twist of passing them as arguments to ensure execution at the right functional level in the program.

In this project I also came face to face with the creature known as BOOST, and that has its own ineffable consequences. The program_options tool is used to great avail in the main function of our compression mini-app. It allows us to provide program help options, and parse the arguments provided by the user in a reasonable way. There's still plenty about the program_options package that I can't fully comprehend, but its usefulness is not lost on me. BOOST also came in handy in our unit testing. We initially started testing to demonstrate that none of the values in the block are modified permanently as a result of the compression/uncompression routines. We also applied it to ensure that the process of reading into a file created the same block representations as if we had hand crafted the block, or string version output.

]