Acc++: Accelerator for C++

Chen Chen, Yifan Hong, Yunhan Hu, Zirui Tao

Abstract

C++ is one of the most popular general-purpose programming languages in the world. Large amount of famous applications(OpenCV, MySQL, etc.) has been developed based on C++. Some of them use C++ and its standard library heavily. Thus, performing hardware-level acceleration on C++ STL function could improve the performance of those applications greatly. Our project proposes hardware acceleration on C++ STL via thorough OpenCV profiling analysis, measures the ideal performance gain from heap accelerator, which shows potential to eliminate the runtime taken up by memory management in real applications.

keywords: C++, Accelerator, STL, Heap Manager

1 Introduction

C++ is one of the most popular general-purpose programming languages due to its extension of C language with additional programming features such as object-oriented programming generic and functional features in addition to low-memory manipulation. Recently, there are ongoing research interests for building hardware-level optimization on popular software languages. Gope et al. [1] examined the inexpensive hardware accelerator options for PHP applications via thorough profiling analysis. Meanwhile, there are emerging works that start looking into the hardware accelerator option on some specific C++ standard functions [2]. After exploring the methodology of [1] and report from [2], one natural question is to explore whether there exist opportunities to discover the overhead of existing C++ standard library (STL) and further optimize in hardware level. Thus, the goal of this project is to build the accelerator for C++ STL.

1.1 Background

Wicht [3] analyzed the run time of C++ containers with the most commonly used functions: Insertion, Iteration, Search, Sort, etc., by varying the number of elements and the type of data. It showed that *list* had the slowest iteration but fast insertion. While *deque* had good iteration but slow insertion. The weakness of each container implied chances to mitigate them by implementing specialized accelerators.
Gope et al. [1] explored the opportunities to improve the performance of realistic, large-scale PHP applications. It analyzed the potential optimizations from conventional micro-architecture and concluded that increasing the size of the branch target buffer and cache was the only method with limited benefit. It then found the four most popular fine-grained PHP activities: hash table accesses, heap management, string manipulation and regular expression handling, and designs a specific accelerator for each of them.

1.2 Basic idea

The basic assumption of designing an accelerator is that certain instructions can be run on customized hardware units instead of the normal architecture for higher resource utilization. In fact, there exist hardware accelerators for varying computing demands (such as GPU for graphics computing and TPU for AI-related computation).
Briefly, our work is to find out the high-frequency functions, select one or two suitable functions for optimization, analyze their running path and operational characteristics, try to find out redundant parts and design special operating units for them.
First, we will start with running several C++/opencv benchmarks, analyze the code and performance to find the distribution of CPU cycles of functions in order to select several major overheads to optimize. We then would be able to observe the most frequent performance/resource bottleneck.
Then we will perform hardware acceleration to those activities, design our accelerating units based on the behaviors of the functions. Our design will be simulated on gem5 to evaluate performance comparing to the original situation. This part will be mainly programmed in C++.

2 Profiling

As mentioned above, our first step is to analyze the benchmarks to find the frequency distribution of functions. This process is called profiling.

2.1 Profiling tools

2.1.1 GNU gprof

The GNU profiler gprof [4] uses a hybrid approach of compiler assisted instrumentation and sampling. To gather profiling information at runtime, the program counter is probed at regular intervals by interrupting the program with the operating system interrupts. As sampling is a statistical process, the resulting profiling data are not exact but are rather a statistical sampling approximation by gprof with induced bias. In addition, as we found that, it has been reported as sometimes losing track of parent function. Nevertheless, since gprof is relatively stable and widely used, we select it as the primary profiling analysis tool in this project. Other than these two drawbacks, gprof provides a clean representation of the histograms of time spent on each function based on its total samples.

2.1.2 Gperftools

Gperftools [5] from Google provides a set of tools aimed at analyzing and improving the performance of multi-threaded applications. We focus on their sampling-based CPU profiler, which has a very little runtime overhead, provides some nice features like selectively profiling certain areas of interest and has no problem with multi-threaded applications. Unfortunately, when we profile with gperftools, the profiler shows raw hex addresses instead of function names, which doesn't profile analysis friendly.

2.1.3 Pin

The pin tool [6] is a software instrumentation tool by Intel that allows easy, programmable profiling tools. It provides a variety of profiling such as, all of which are implemented through a C++ class API, which handy to customize. This allows Pin to be a very powerful programming instrumentation tool to analyze almost any program behaviors.

2.1.4 Valgrind

Valgrind is a powerful software development tool that can be used to analyze memory errors, function calls, cache usage, stack and heap management, and multi-threaded contention. In this project, we use the callgrind, recording function calls and generating call graphs, and cachegrind, counting the cache misses and hits, in the Valgrind. The function of callgrind overlaps with the function of the previous tools, so it is only used to be a reference. Cachegrind is the main tool used. For the benchmarks we interest, we will record the cache usage. For the benchmarks with high hit rates, we will give up because they are hard to be improved.

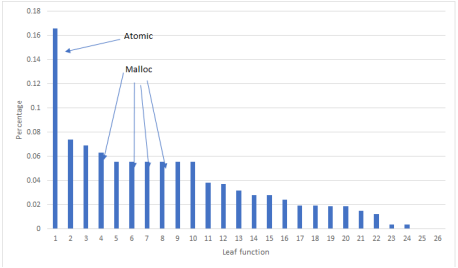

Figure 1: Normalized Runtime percentation of leaf function across OpenCV sample applications

2.2 Benchmark

We started the with the isolated C++ STL container benchmark [3]: it consists of a single container function across testbenches to observe the container function level opportunity. Following from Nghia Ho [7], we performed another profiling of pseudoinverse [7] application compiled by a set of numerical C++ libraries, to determine suitable C++ application candidates. Then we moved to profile the an subset of OpenCV sample applications [8] as for our real application suite, under the OpenCV source sample directory, the OpenCV benchmark consists of four main categories: Image Processing, Algorithm, Video and IO and spans to total of 12 applications.

2.3 Profiling results

To identify the utilization rate over the entire leaf function, we aggregated the utilization rate (percentages of total execution time) of each leaf function across all applications and normalized plot the histogram in terms of the normalized scale. Figure 1 shows the histogram accumulated results.

We first notice the spike appeared on the top leaf function, after examining it as only an automic operation for thread safety, we concluded that it may be the profiling inaccuracy such that it treats such as function call. However, leaf function 4,6,7,8 are std::allocator based operation such as allocation and deallocation. Therefore, we concluded that an heap manager based solution might be the possible acceleration opportunity we could do. Even though optimization over malloc operation in software level has been well exploited in literature [9], [10], as well as in hardware level for PHP web-server application [1] focusing on C++/STL hardware optimization is indeed original.

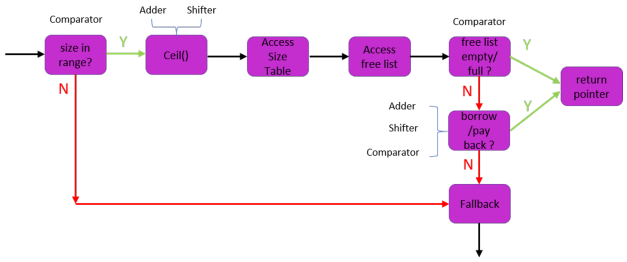

Figure 2: Flow chart for accelerator execution

Figure 3: A illustration of designed type size configuration of the free list

3 Optimization Strategy

Inspired by Kanev Et al. [2], the flow chart of accelerator consists of steps that checks the requested size, rounds up to the closest size class that the accelerator supports, then accesses the associated free list which consists of a list of entries to "cache" available pointers the size of which it pointers to (type size) is of the requested size. Instead of using two arrays, as adopted by them, to tune the mapping of requested size and actually allocated sizes in software implantation, the hardware only simply takes the explicit round-up size and use it as the index to the corresponding free list. The tuning in software is helpful since it can change the type size as program runs. They control the degree of round-up according to requested size's history request frequency : the more frequent request size was requested the closer the requested size is to the size the software aims to accelerate. The purpose of their design is to be both specific (accelerating fixed set of size classes) while not losing too much extensiveness (their accelerator being applicable, to some extent, to other size classes) as well. However, on the hardware level perspective, we conclude that this two-way mapping is not necessary as we follow to propose our alternative design that could achieve the same objective.

We utilize the fact that serving a larger type size pointer would also suffice to serve a smaller type size pointer. Especially, when the larger type size is an integer multiple of the smaller type size, the larger pointer could be used as two smaller pointer, each serving individual malloc requets. This is helpful when one particular size is being requested more often than others. In this case, it is more likely that the associated free list would be empty than other free lists, cousing the OS fallback. If we are able to implement functionality that whenever the free list is empty, the free list instead tries to first look for entries on other free lists and use other free lists space to serve allocation request, then the accerlator would benefit from some cases avioding to call OS to fallback. We call this logic module as borrow logic. our designs adopts this and Fig. 3 demonstrates an configuration of free lists: each free list has equal size, but the type size of the pointer follows some power of 2 .Suppose free list A holds pointer of type size 16 bytes and free list B holds pointer of type size 32 bytes. Whenever the request size is 16 bytes and A is empty, A then tries to borrow a available space from freelists with entries of larger type size whenever there is one. Suppose B is not empty, then A would borrow B a entry of pointer with 32 bytes type size. Then, field burrow address A.b_addr = B.head and burrow count A.b_cnt = 32/16, would suffice to boo keep how much what is the return address and how many requests A can utilize entries in B. Then B also updates its header pointer and size. In essence, the size request is forwarded to B and the rest of memory is all used to serve for the same type size request by bookeeping the A.b_addres and A.b_cnt. As shown in 2, the hardware implementation for bookkeeping does not involve complicated logics and can be finished within few cycles as compared to fallback to OS heap manager.

List. 2 shows the pseudocode for borrow logic for a single borrow event:

```
1
2  def borrow(header_i):
3      for j in range(0, i): # suppose index
       position alighted from large type_size to
       small type_size
4          diff = i - j
5          if header_j.size < header_j.capacity:
6              # borrow memory pointer from
       header_j to header_i:
7              header_i.b_address = header_j.head
8              header_i.b_cnt = 1 << diff
9              header_j.size +=1
10             header_j.head = (header_j.head +
       1)%(header_j.capacity)
11             return (header_i.type_size *)
       header_j.head
12
13     return None
14
15
```

Listing 1: Pseudocode for simple borrow logic

The pseducode given above only shows the case where

each free list only borrows once. When more than one borrow is allowed, the field variable b_addr and b_cnt needs to correspond to each borrow, making the implementation slightly less efficient. Instead of defining them to be field variable, we further propose an b_cnt matrix and b_addr matrix both with dimention $n \times n$, where n is the to number of free list. Entry of b_cnt matrix b_cnt[i,j] represents how many blocks free list j is borrowed from i and entry of b_addr matrix b_addr[i,j] records the "base address" of arrays of smaller pointers, as space reserved for a larger object could be alternatively broken into multiple sequential spaces reserved for smaller objects, each with a pointer address. The borrowing logic avoids the OS fallback as much as possible the by utilizing available resources from other free lists, and it also naturally handles the memory alignment as each larger type size is an multiple of smaller type size by the design.

Similarly, when the current free list is full while trying to push and an recycled space from de-allocation, the space could also be utilized by multiple smaller, pointer from other free list(s). In this case, no bookkeeping logic is needed as deallocations directly changes the metadata record by the current free list (head, size). We call this forwarding of memory space from one free list to another free list whose entries holds smaller pointer type size as"Pay (back)".

As for the comparison from Kanev Et al., their tuning allows to improve utilizatoin by serving allocated size close to requested size for those important type size. Instead, the burrow and pay back logic would allow better utilization at the allocated size level. Thus, our originality arises from the burrow and pay back logic to better utilization of free list space, and thus from the hardware perspective. Gope Et al. [1] did not include the borrow and pay back logic and thus treats each free list space as totally independent. Because of the time constraint we did not conduct follow-up experiments for the comparison, but it would be an intersting comparison of better utilization from hardware and software perspectives.

4 Experimental Analysis

To measure how specialized accelerators will improve the performance, we analyze the ideal performance gain from *heap accelerator* and *iterator update accelerator*. The experiment implementation requires to modify on both source code and simulator. More details will be shown on the following 2 subsections.

4.1 C++ Side

On C++ Side, for the function we are hoping to accelerate, we modify the library to execute the accelerator instruction instead. First we'll dive into the selected leaf function, replacing the inside with our specific accelerator instruction. For an example of our heap manager, this could be done like:

```
void* mystd::my_malloc(int size)
{
    ...
    asm("fsubr %st, %st(1)");
    int* ptr;
    asm("mov %%rax, %0;":"=r" ( ptr ));
    return (void*)ptr;
}
```

In our rewrite malloc function, the input parameters must be kept the same to allow program using library to compile. Then, we replace contents of the actual function with keyword asm to mix inline assembly code with original C++ code. Instruction *fsubr* was an unused instruction but still available in gem 5. Here we modify fsubr in gem5, decoded to invoke our accelerator (more details will be discussed on gem5 side later), then the connection between source code and our accelerator is successfully created. Moreover, in the case of our heap manager accelerator, our malloc function has to actually return valid values back for gem5 to continue correct execution of the program. In this case, fsubr was responsible in gem5 emulation for calculating what the next pointer should be, and placing the value back into the 'rax' register, the default where the return value should be based on function call conventions.

After we successfully implement the leaf function, another problem is to make sure it is able to be correctly called by top function. Unfortunately, simply overload standard library leaf function is not allowed by C++ convention. For example, if we would like to accelerate std::sort with leaf function std::swap, only modifying std::swap is not feasible. Here we have to perform a bottom-up rewrite, i.e. we start from bottom, rewrite leaf function in our new namespace, then rewrite its caller to call the new leaf function, then rewrite its caller's caller... Due to the complexity of C++ standard library and OpenCV library, this rewriting part is one of the most time-consuming work in our project.

4.2 Gem5 Side

We use gem5 [11] to simulate the operation of original function as baseline. We simulate each specialized accelerator's behavior and compare their performance with the baseline. We modify fsubr instruction [12] implemented on gem5 to enable accessing the data structure of accelerator. We apply checkpoint [13] to measure the runtime of *region of interest* (ROI),

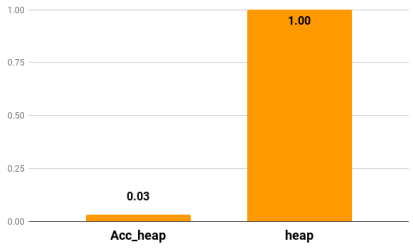

Figure 4: Normalized Runtime of Heap Management

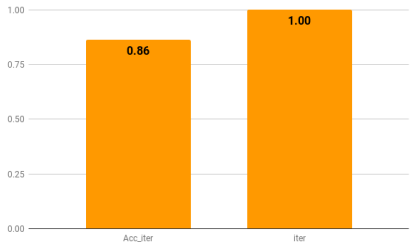

Figure 5: Normalized Runtime of Iterator Update

which bypasses the accelerator construction and deconstruction operations. The PseudoCode to simulate accelerator behaviors is shown below:

```
int main()
{
    // Initialize data structure
    Acc_DataStruct.init();
    ckpt.enable();
    // ROI begin
        ...
        asm("fsubr %reg, %reg(1)");
    // ROI end
    ckpt.end();
    ~Acc_DataStruct();
}
```

Analyze heap accelerator. We modify *fsubr instruction* to simulate pointer calculation process. We assume the fsubr instruction is as fast as a simple *add* instruction. We assume accelerator finishing memory allocation and deallocation out of the simulation space. We assume accelerator having *infinite size* and operating *Max_size()* in accelerator. We use benchmark that contains heap management operation only.

Fig.4 compares the runtime of heap accelerator with the baseline. The accelerator achieves huge performance improvement, which only takes 3% runtime of the baseline. The reason is because a large portion of heap management's runtime is taken up by memory allocation and deallocation, which is pre operated by accelerator. The pointer calculation process brings runtime consumption to accelerator which only takes 3% in overall heap management.

The performance improvement for real application benchmarks depends on the weight of heap management. If a benchmark uses 20% runtime to process heap management, it can achieves up to 20% performance improvement by implementing accelerator for heap management.

Analyze iterator update accelerator. We haven't proposed accelerator strategy for iterator update, such as *iter++*, but we want to analyze the opportunities to improve performance by designing a specialized accelerator. We apply similar implementation method and replace iterator update operation to a simple *add* instruction. Fig.5 shows that iterator update accelerator can maximally achieve limited performance upgrade (13%), and since iterator update process doesn't take a large portion of runtime (less than 1%) in real benchmarks, it will get less benefits to design a specialized accelerator for iterator update.

5 Future Work

Modify OpenCV benchmark. We need to move the data structure related with targeted accelerator out of the *region of interest* and measure the performance improvement achieved by heap accelerator. The challenge to modify OpenCV benchmark is the method to modify their libraries because they are existing in binary method.

Look for more opportunities. The iterator structure contains other leaf functions with high frequency, such as std::operator[14], excepted iterator update process. We can look opportunities to design targeted accelerator for those leaf functions.

RTL synthesis heap accelerator strategy. We need to write *Verilog* to analyze the area and power consumption of our accelerator scenario.

6 Conclusion

We use profiling tools to analyze the leaf functions with the highest frequency distribution. We propose our heap accelerator strategy. We measure the ideal performance gain from heap accelerator, which shows potential to eliminate the runtime taken up by memory management in real applications. We analyze the idea performance gain from a specialized *iterator update* accelerator. We think it brings negligible benefits to implement such a kind of accelerator. We would

like to use profiling tools to look more optimizing opportunities in future.

References

[1] Dibakar Gope, David J Schlais, and Mikko H Lipasti, "Architectural support for server-side php processing," in *ACM SIGARCH Computer Architecture News*. ACM, 2017, vol. 45, pp. 507–520.

[2] Svilen Kanev, Sam Likun Xi, Gu-Yeon Wei, and David Brooks, "Mallacc: Accelerating memory allocation," *ACM SIGOPS Operating Systems Review*, vol. 51, no. 2, pp. 33–45, 2017.

[3] Baptiste Wicht, "C containers benchmark: vector/list/deque and plf::colony," May 2017.

[4] Susan L Graham, Peter B Kessler, and Marshall K Mckusick, "Gprof: A call graph execution profiler," in *ACM Sigplan Notices*. ACM, 1982, vol. 17, pp. 120–126.

[5] "gperftools," `https://github.com/gperftools/gperftools`, [Online; accessed 22-November-2019].

[6] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Acm sigplan notices*. ACM, 2005, vol. 40, pp. 190–200.

[7] "Opencv vs. armadillo vs. eigen vs. more! round 3: Pseudoinverse test," Nov 2012.

[8] Gary Bradski and Adrian Kaehler, "Opencv," *Dr. Dobb's journal of software tools*, vol. 3, 2000.

[9] Richard L Hudson, Bratin Saha, Ali-Reza Adl-Tabatabai, and Benjamin C Hertzberg, "Mcrt-malloc: a scalable transactional memory allocator," in *Proceedings of the 5th international symposium on Memory management*. ACM, 2006, pp. 74–83.

[10] Dave Dice and Alex Garthwaite, "Mostly lock-free malloc," in *ACM SIGPLAN Notices*. ACM, 2002, vol. 38, pp. 163–174.

[11] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al., "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[12] "Cs 752: Advanced computer architecture i (fall 2015 section 1 of 1) homework 2," `http://pages.cs.wisc.edu/~david/courses/cs752/Fall2015/wiki/index.php?n=Main.Homework2`, [Online; accessed 11-December-2019].

[13] "Cs 752: Advanced computer architecture i (fall 2015 section 1 of 1) homework 3," `http://pages.cs.wisc.edu/~david/courses/cs752/Fall2015/wiki/index.php?n=Main.Homework3`, [Online; accessed 11-December-2019].

[14] "libstdc++: Iterators," `https://gcc.gnu.org/onlinedocs/gcc-4.6.3/libstdc++/api/a01201.html`, [Online; accessed 11-December-2019].

7 Contribution

We agree with the following ratio of contribution for every member of our group.

	Profiling analysis (100%)	Design (100%)	Simulation (100%)	Documentation (100%)
Chen Chen	5%	0%	70%	20%
Yunhan Hu	10%	25%	0%	30%
Yifan Hong	40%	0%	30%	15%
Zirui Tao	45%	75%	0%	35%

Table 1: Contribution Table