GPU-Disasm: A GPU-based x86 Disassembler

Evangelos Ladakis[1], Giorgos Vasiliadis[1], Michalis Polychronakis[2], Sotiris Ioannidis[1], and Georgios Portokalidis[3]

[1] FORTH-ICS, Greece
{ladakis, gvasil, sotiris}@ics.forth.gr
[2] Stony Brook University, USA
mikepo@cs.stonybrook.edu
[3] Stevens Institute of Technology, USA
gportoka@stevens.edu

Abstract. Static binary code analysis and reverse engineering are crucial operations for malware analysis, binary-level software protections, debugging, and patching, among many other tasks. Faster binary code analysis tools are necessary for tasks such as analyzing the multitude of new malware samples gathered every day. Binary code disassembly is a core functionality of such tools which has not received enough attention from a performance perspective. In this paper we introduce GPU-Disasm, a GPU-based disassembly framework for x86 code that takes advantage of graphics processors to achieve efficient large-scale analysis of binary executables. We describe in detail various optimizations and design decisions for achieving both inter-parallelism, to disassemble multiple binaries in parallel, as well as intra-parallelism, to decode multiple instructions of the same binary in parallel. The results of our experimental evaluation in terms of performance and power consumption demonstrate that GPU-Disasm is twice as fast than a CPU disassembler for linear disassembly and 4.4 times faster for exhaustive disassembly, with power consumption comparable to CPU-only implementations.

1 Introduction

Code disassemblers are typically used to translate byte code to assembly language, as a first step in understanding the functionality of binaries when source code is not available. Besides software debugging and reverse engineering, disassemblers are widely used by security experts to analyze and understand the behaviour of malicious programs [8,12], or to find software bugs and vulnerabilities in closed-source applications. Moreover, code disassembly forms the basis of various add-on software protection techniques, such as control-flow integrity [24] and code randomization [16].

Most previous efforts in the area have primarily focused on improving the accuracy of code disassembly [9,13,24]. Besides increasing the accuracy of code disassembly, little work has been performed on improving the speed of the actual disassembly process. As the number of binary programs that need to be analyzed is growing rapidly, improving the performance of code disassembly is vital for coping with the ever increasing demand. For instance, mobile application repositories contain thousands of applications that have to be analyzed for malicious activity [17]. To make matters worse, most of these applications are updated quite frequently, resulting in large financial and time costs for binary analysis workloads. At the same time, antivirus and security intelligence vendors need to analyze a multitude of malware samples gathered every day from publicly available malware scanning services and deployed malware scanners.

In this work, we focus on improving the performance of code disassembly and propose to offload the disassembly process on graphics processing units (GPUs). We have designed and implemented *GPU-Disasm*, a GPU-based disassembly engine for x86 code that takes advantage of the hundreds of cores and the high-speed memory interfaces that modern GPU architectures offer, to achieve efficient large-scale analysis of binary executables. GPU-Disasm achieves both inter-parallelism, by disassembling many different binaries in parallel, as well as intra-parallelism, by decoding multiple instructions of the same binary in parallel. We discuss in detail the challenges we faced for achieving high code disassembly throughput.

GPU-Disasm can be the basis for building sophisticated analysis tools that rely on instruction decoding and code disassembly. We chose to focus on the x86 instruction set architecture for several reasons. First, x86 and x86-64 are the most commonly used CISC architectures. Second, building a disassembler for a CISC architecture poses more challenges compared to RISC, due to much larger set of instructions and the complexity of the instruction decoding process. Third, it is easier to apply the proposed GPU-based design decisions to a RISC code disassembler than the other way around.

We have experimentally evaluated GPU-Disasm in terms of performance and power consumption with a large set of Linux executables. The results of our evaluation demonstrate that GPU-Disasm is twice as fast compared to a CPU disassembler for linear disassembly, and 4.4 times faster for exhaustive disassembly, with power consumption comparable to CPU-only implementations.

In summary, the main contributions of this paper are:

1. We present the first (to our knowledge) GPU-based code disassembly framework, aiming to improve the performance of the instruction decoding process.
2. We present techniques that exploit the GPU memory hierarchy for optimizing the read and write throughput of the decoding process. Such memory optimizations can be applied in tools with similar memory I/O operations.
3. We evaluate and compare our GPU-based disassembly library with a CPU-based approach in terms of performance, cost, and power consumption.

2 Background

2.1 General Purpose Computing on GPUs (GPGPU)

While GPUs are traditionally used for computer graphics, they can also be used for general-purpose computation. Due to the massive parallelism they offer, they can achieve significant performance boosts to certain types of computation. GPUs typically contain hundreds (or even thousands) of streaming cores, organized in multiple *stream multiprocessors* (SM). GPU Threads are divided in groups of 32, called *warps*, with each core hosting one warp. Each warp executes the same block of code, meaning that the threads within a warp do not execute independently, but all of them run the same instruction concurrently. Consequently, code containing control flow statements that lead to different threads following divergent execution paths, cannot fully utilize the available cores. When some threads within a warp diverge, because a branch follows a different path than the rest of them (*branch divergence*), they are stalled. Consequently, the tasks that can truly benefit from the massively parallel execution of GPUs are the ones that do not exhibit branch divergence. Among many domains, GPUs have been used in scientific computing [2], cracking passwords [1], machine learning [5], and network traffic processing [20–22].

GPUs have a distinct memory model. Each multiprocessor has a set of 64K registers, which are the fastest GPU memory component. Registers are assigned to threads and are privately scoped. The scheduler is responsible for ensuring that register values are saved and restored during context switches of threads. Each multiprocessor has its own Level 1 (L1) cache and shared memory, which are shared by all the threads running on it, and are part of the same physical memory component. This allows for choosing at run time (before spawning the GPU threads) how to distribute memory between cache and shared memory. The L1 cache is organized in data cache lines of 128 bytes. Shared memory is as fast as L1 cache but is programmable, which means that it can be statically allocated and used in GPGPU programs.

GPUs also include *global memory*, which is equivalent to the host's RAM. It is the slowest memory interface, but has the largest capacity. Global memory is available to all SMs and data from the host to the device and vice versa can be transfered only through this part of memory. Interestingly, global memory also hosts *local memory*, which is used by threads to spill data when they run out of registers or shared memory. Finally, global memory also includes constant memory, a region where programs can keep read-only data, allowing for fast access when threads use the same location repeatedly.

A Level 2 (L2) cache is shared between all SMs and has a larger capacity than L1. Every read/write from and to the global memory passes through the L2 cache. A GPU multiprocessor can fetch 128 byte lines. The driver keeps this alignment in global memory and in cache lines to achieve increased throughput for read and write operations. The maximum transfer throughput to global memory is 180 GB/s.

There are two frameworks commonly used to program GPUs for general purpose computations, both using C API extensions. The first is CUDA [14], a programming framework developed by NVIDIA (which we use in this work), and the second is OpenCl [19], which is a generic framework for programming co-processors with general purpose computational capabilities.

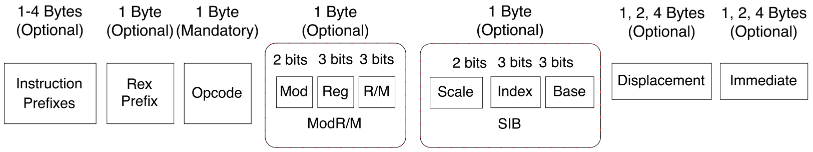

Fig. 1: x86 Instruction format.

2.2 `x86` architecture

The `x86` and `x86-64` architectures are probably the most widely used `CISC` (Complex Instruction-Set Computing) architectures [7]. Their instruction sets are rich and complex, and most importantly they support instructions of varying length. Instruction lengths range from just *one* byte (i.e., instructions comprising just an opcode) to 15 bytes. Generally, instructions consist of optional prefix bytes, which extend the functionality of the instruction, the opcode, which defines the instruction, the ModR/M and SIB bytes, which describe the operands, followed by an immediate value, that is also optional. The overall format of an `x86` instruction is depicted in Figure 1.

Due to the extensive instruction set and the variable size if its instructions, it is very easy for disassemblers to be confused, decoding arbitrary bytes as instructions [3], e.g., because data may be interleaved with instructions, or because the beginning of a block of instructions is not correctly identified.

2.3 Code Disassemblers

There are two widely used code disassembly techniques, *linear* and *recursive* disassembly [6]. In linear disassembly, a segment of bytes is disassembled by decoding instructions from the beginning of the of the segment until the end is reached. Linear disassembly typically does not apply any heuristics to distinguish between code and data, and consequently, it is easy to get "confused" and produce erroneous results. For example, compilers emit data and patching bytes for function alignment, which a linear disassembler decodes as instructions, along with the actual code. Thus, when disassembling the whole text segment of a binary, the output of linear disassembly is likely to contain erroneous parts that correspond to embedded data and alignment byte Binaries may also contain unreachable functions that are included during compilation, e.g., due to the static linkage of libraries, which will also be included in the output of linear disassembly.

Recursive disassemblers use a different approach that eliminates the erroneous assembly produced by linear disassembly, but with its own disadvantages. The decoding process starts from an address out of a set of entry points (exported functions, entry points) and linearly disassembles the byte code. Whenever the disassembler encounters control flow instructions, it adds all targets to the set of entry points. The disassembly process stops when it finds indirect (computed) branches which cannot be followed statically. The process continues recursively by decoding from a new target out of the set of entry points. They main drawback of recursive disassembly is that it cannot reach code segments that are accessible only through indirect control flow transfer instructions.

3 Architecture

In this section, we describe the overall architecture of our system. Our aim is to design a GPU-based disassembly engine that is able to process a large number of binaries in parallel. The key factors for achieving good performance are: (i) exploit the massively parallel computation features of the GPU, (ii) optimize PCIe transfers and pipeline all components for keeping the hardware utilized at all times, and (iii) design optimization heuristics for exploiting further capabilities of the hardware.

The basic operations of our approach include: (i) *Pre-processing*: loading of the binaries from disk to properly aligned buffers of the host's memory space, (ii) *Host-to-device*: transfer of the input data buffers to the memory space of the GPU, (iii) *Disassembly*: the actual parallel code disassembly of the inputs on the GPU, and storage of the decoded instructions into pre-allocated output data buffers, (iv) *Device-to-host*: transfer of the output buffers to the host's memory space, and finally (v) *Post-processing*: delivery of the disassembled output and initialization of the pointers to the next chunk of bytes of each binary, if any, that will be fed to the GPU for disassembly. Once processing of all binaries has completed, input buffers are loaded with the next binaries to be analyzed.

3.1 Transferring Input Binaries to the GPU

The operation to consider is how input binary files will be transferred from the host to the memory space of the GPU. The simplest approach would be to transfer each binary file directly to the GPU for processing. However, due to the overhead associated with data transfer operations to and from the GPU, grouping many small transfers into a larger one achieves much better performance than performing each transfer separately. Thus, we have chosen to copy the binary files to the GPU in batches. In addition, the input file buffer is allocated as a special type of memory, called page-locked or "pinned down" memory, in order to prevent it from being swapped out to secondary storage. The copy from page-locked memory to the GPU is performed using DMA, without occupying the CPU. This allows for higher data transfer throughput compared to the use of pageable memory, e.g., using traditional memory allocation functions such as `malloc()`.

3.2 Disassembling x86 Code on the GPU

Instruction Decoding and Linear Disassembly. Linear disassembly blindly decodes a given sequence of bytes from the beginning to the end without applying any further heuristics or logic. Initially, the GPU decoder dispatches the instruction prefixes (if present), which always come before the opcode of x86 instructions. Afterwards, the decoder dispatches the next byte of the instruction which is the actual opcode we are interested in. The decoder shifts the opcode bytes to bring them in a form that it can easily use them as an index for a look-up table. After decoding the opcode, we determine if the instruction has operands or not, by decoding the ModR/M byte. The operands can be registers or immediate values. If the operands are registers, they can be either implicit, as part of the instruction, or explicit, defined by the following bytes. If the instruction uses indexed addressing, then the next decoded byte corresponds to the SIB (Scale Index Base) which determines the addressing mode of the array. Lastly, the disassembler decodes the displacement and immediate bytes.

The disassembly process can fail while decoding an instruction. Depending on the failure reason, the disassembler handles it in a different way. When more bytes than available are expected based on the last decoded opcode, the instruction decoding process stops and an appropriate error is reported. When invalid instructions are encountered, the disassembler marks them and continues the decoding process from the following byte.

Each GPU thread is assigned to disassemble a single chunk of an input binary at a time. Consequently, the total GPU kernel execution time is equal to the time of the slowest (last finished) thread. Note that the overall performance would drop in case some threads remained under-utilized, i.e., they were assigned smaller workloads. To avoid this, we assign fixed-sized input buffers (chunks) to all threads, which minimizes the possibility of having idle threads. However, as all input binaries do not have the same size, some imbalance unavoidably happens as the processing of smaller input files completes. Our current prototype does not handle such imbalances, but their effect can be minimized by selecting input file batches based on file sizes, so that each batch includes files of similar sizes.

Having fixed size chunks leads to more complex data splitting, when a binary may not fit inside the buffer all at once. Therefore, we have to divide the binary in several chunks and perform the disassembly process on batches. Due to the nature of the x86 instruction set (Section 2.2) we have to carefully choose the starting point of the next chunk of bytes for decoding, otherwise any split instructions will generate incorrect disassembly.

Exhaustive Disassembly. We have also implemented an exhaustive disassembly mode, which applies linear disassembly by starting from each and every byte of the input, i.e., by decoding all possible (valid) instructions contained in the input. Further analysis of the output can be then performed to identify function boarders, basic blocks, and even obfuscated code constructs. For instance, Bao et al. [4] use exhaustive disassembly to generate all possible outputs, and then apply machine learning techniques to find instruction sequences that correspond to function entry and exit points. Other approaches [8,13] disassemble the same regions of a binary from different indexes and apply heuristics to identify basic blocks and reconstruct the the control flow graph.

For exhaustive disassembly, we transfer the input buffer to the GPU memory space and spawn as many threads as the bytes of the binary. Each thread starts the decoding of the same input from a different index. Although each thread decodes only one instruction, this approach is effective in quickly extracting all possible instructions contained in the input.

3.3 Transferring the Results to the Host

After an instruction is decoded, the corresponding data is stored in the GPU memory. As storing extensive data for all decoded instructions from all threads can easily deplete the memory capacity, we chose to save only basic information about each decoded instruction, which though is enough for further analysis. Specifically, we store the relative address of the instruction within the input file, its opcode, the group to which it belongs (e.g., indirect control flow transfer, arithmetic operation, and so on), and all explicit operands such as registers and immediate values. The above extracted information can fully describe each decoded instruction, and can be easily used for further static analysis, compared to more verbose storing of raw fields, such as ModR/M bits. Information such as implicit operands and the size of the instruction mnemonic can be easily extracted from the stored metadata. For example, the size of the instruction can be calculated from the distance between the relative addresses of the current and the next instruction.

The decoded instructions are stored in a pre-allocated array with enough space for all instructions of the input. As shown in Figure 4 (discussed in more detail in Section 5.1), only less than 20% of the encountered instructions on average are a single-byte long, so the number of decoded instructions in typically much smaller than the size of the input in bytes. Consequently, we safely set the number of slots in the array as half the size of the input buffer in bytes.

The GPU disassembly engine saves the decoded instructions on GPU memory and transfers them back to the host for further analysis. After the device to host transfer has completed, the system evaluates the extracted information as part of a post-process phase. This includes checks for errors due to any misconfiguration of the GPU threads, and for each thread, whether there are pending bytes for disassembly for the current input binary being processed. Then, the pointer for the next chunk to be processed is set according to the last successfully decoded instruction, so that the disassembly process is not corrupted. If a thread has finished disassembling an input binary, the pointer is set to NULL so that a new binary will be assigned to it, after the processing of the whole batch is completed.

3.4 Pipeline

After optimizing the basic operations, we have to design the overall architecture in such a way that will keep every hardware component utilized. The GPGPU API supports running computations using streams. Thus, we can parallelize data transfers with the disassembly process and eliminate idle time for the PCIe bus and the GPU multiprocessors. We use double buffers for both input and output, so that when the GPU processes a buffer, the system can transfer the output data and fill the next input buffers with new binaries for disassembly. With the proper usage of streams, we can keep the CPU, the PCIe bus, and the GPU utilized concurrently at all times.

The GPU can handle the synchronization of GPU operations internally. However, before the host proceeds with output analysis, it needs to synchronize the GPU operations. The host is unable to know if the device has finished processing until the driver receives a signal from the GPU that denotes completion. Ideally, we would like to keep the GPU utilized without blocking for synchronization. The architecture can be designed so that synchronization is kept to a minimum, just for one of the operations. By placing all input values (binaries, sizes, memory addresses) and all output data into a single buffer, as described above, requires invoking the synchronization process only after the copy of the output from device to host, eliminating in this way any intermediate serialization points.

4 Optimizations

4.1 Access to Global Memory

Due to the linear nature of the disassembly process, we enforce both reads and writes to the input and output buffers to be performed only once for each decoded instruction. As mentioned, the instruction sizes of the `x86` ISA vary significantly, ranging between 1 and 15 bytes. According to the alignment property that GPUs follow for the memory accesses, different sequences of instructions with different sizes may result in misaligned accesses, consequently resulting in degraded memory access throughput.

We describe the improvement of the reading process in Section 4.3. Regarding the improving the write throughput of the disassembly output to global memory, GPU best practices [15] propose that data structures on the GPU should be placed as structs of arrays. In most cases, this results in improved data throughput from global memory. However, in our case we observed lower performance due to the drop of the writing throughput back to global memory. We tackled this issue and achieved a better throughput by having a struct with the decoded information per instruction, instead of separate arrays for each field.

4.2 Constant Memory

A crucial part of the disassembler are the look-up tables with the decoding information that are hardcoded in the instruction decoder. These tables are used as dispatchers for the decoding process. They hold information about each instruction, such as the opcode, whether there are operands and how many to expect, the type of the instruction, the group of the architecture extension of an instruction, and so on. The look-up tables are constants and shared through all threads. Therefore, we can use the constant memory of the GPU in order to have fast access to these tables. The constant memory though is limited in size, and the look-up tables can easily exceed the available memory. To strike a balance between performance and accuracy, we measured the most used tables and placed them to the GPU constant memory, and kept the more rarely used tables in the (slower) global device memory. Furthermore, global variables such as function pointers that are being assigned by the initialization process, are placed to the shared memory of each multiprocessor, which can be initialized at run time.

4.3 Access to L2 Cache

Read and write data accesses pass through the L2 cache, which is a shared memory interface for all multiprocessors as the global memory. The L2 cache memory is n-associative [23], which means that data lines are placed depending on the least significant bits of the accessed address. When assigning large input buffers to each thread, memory divergence increases, and consequently, line collisions inside the L2 cache occur more frequently as well. On the other hand, having small input buffers will result in under-utilization of the GPU threads, and an overall drop in performance.

Taking in consideration this trade-off, we sought a solution that combines the benefits of both approaches. Each read access to the global memory from a multiprocessor fetches a 128-byte line of data. Consequently, we chose to divide large buffers into smaller ones (as shown in Figure 2) with a size aligned to the access line of the GPU, and place them within the larger buffer in such a way that threads access the buffer as a group. We evaluated buffer sizes of 16, 32 and 64 bytes, and the results of our experiments showed that beyond 32 bytes, the L2 hit ratio from the L1 cache dropped due to line collisions (Table 1). For every 32 bytes of the input buffer, we place in the first 16 bytes the previous 16 decoded bytes, and in the following 16 bytes the new bytes that have to be decoded. The repeated bytes are needed for correcting the decoding alignment, in case of out-of-bounds errors of a previous disassembly. In that case, we continue the decoding process from the byte where the previous disassembly stopped at, until the end of the 32 bytes. Furthermore, this optimization forces the disassembler to make fixed read accesses to global memory, which achieves better throughput.

4.4 Data in GPU Registers

We take advantage of the GPU registers to store statically allocated data that is frequently used by the decoder. Typically, instruction operands are dynamically allocated for each instruction, due to the fact that the number of operands am `x86` instruction uses is not known in advance. We changed the list of operands to a static array, which eventually the compiler keeps in registers. As mentioned earlier, operands may be either explicit or implicit. Due to memory capacity limitations, we decided to keep in registers only the explicit operands (three or less). Implicit operands depend on the instruction opcode, and therefore can be easily inferred.

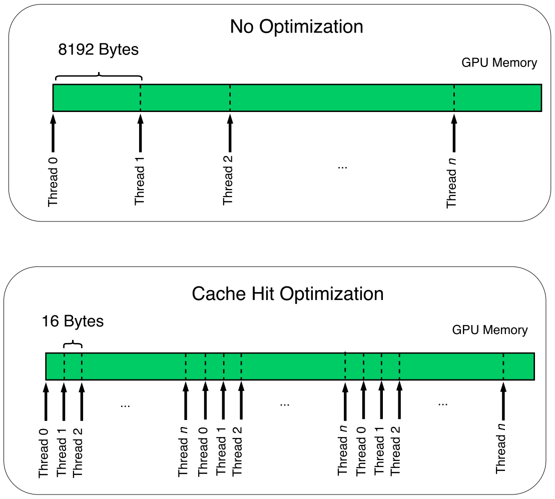

Fig. 2: Reading inputs from GPU global memory with L2 cache optimization.

Keeping operands into registers instead of shared memory is preferable because the latter would affect the L1 cache of each multiprocessor, which corresponds to the same hardware, and therefore would drop the read access throughput of the input binaries. Also, the shared memory would have to be divided according to the number of threads for each multiprocessor, imposing an upper-bound on the number of threads that could be spawn due to the size of temporary list of operands for each thread.

Another use of registers is related to improving the read throughput of the input buffers. Traditionally, read requests pass from global memory through the L2 cache, and finally the data are fetched to the L1 cache of the corresponding multiprocessor. In order to avoid reading from the L1 cache, or even worse to overwrite the cache line where decoded bytes are stored, we save the 32 byte lines into a *uint4_t* statically declared array, which is translated at compile time in register storage. Although excessive use of registers can result in register spilling to local memory, any incurred latencies can be hidden by spawning more threads. Our experiments show that stall instructions due to local data accesses are rare.

5 Evaluation

In order to evaluate our GPU-based disassembler, we create a corpus of 32,768 binaries from the `/usr/bin/` directory of a vanilla Ubuntu 12.04 installation, allowing duplicates to reach the desired set size. The sizes of the binaries vary between 30 KB and 40 KB. Our testbed consists of a PC equipped with an Intel i7-3770 CPU at 3.40GHz and 8 GB of RAM, and an NVIDIA GeForce GTX 770 GPU with 1536 cores and 4 GB of memory.

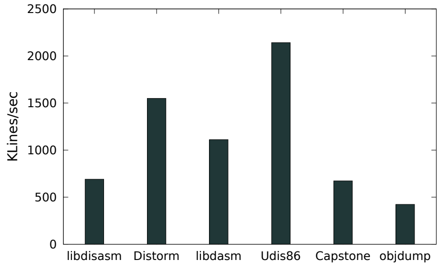

Fig. 3: Performance evaluation of various open-source linear disassemblers.

5.1 Performance analysis

The performance evaluation examines both the system as a whole, as well as its sub-parts (e.g., the decoding engine and data transfers). We also test existing CPU-only disassemblers for comparison. We report the throughput of the disassembly process as the number of assembly lines (or decoded instructions) produced per second. As the size of instructions in `x86` ISA varies, it would be misleading to measure the number of bytes processed per time.

Performance analysis of open-source disassemblers As a first step, we evaluate several popular open-source linear disassemblers to estimate the throughput of conventional CPU-based disassemblers. In order to eliminate any I/O overhead, we redirect the output of the tools to `/dev/null`. Figure 3 depicts the average disassembly rate for various disassemblers in thousands of assembly lines (KLines) per second, when utilizing a single CPU thread. The faster disassembler is *Udis86*, which achieves a throughput of 2142.2 KLines/sec and the slower is the *objdump* utility, which processes 423.664 KLines/sec. The differences in throughput are mostly due to the data produced for disassembled instruction; the more information generated by a disassembler, the lower its throughput. For instance, some tools record only the opcodes and the corresponding operands for each instruction, while others include information such as its instruction group, relative virtual addresses, etc.

Data Transfer costs In this experiment, we measure the data transfer rate between CPU and GPU over PCIe for different block sizes of data. Figure 5 shows the results in GB/sec including standard error bars for transferring data from host to GPU memory and vice versa. The maximum theoretical transport bandwidth for PCIe 3.0 is 16 GB/s, however, in this experiment the maximum achieved rate is 12 GB/s, when transferring blocks of 16 MB.

GPU Instruction-Decoding Performance In this section, we evaluate the decoding performance of the GPU, excluding any data transfers, and pre- and post-processing occurring on the CPU (e.g., opening files and preparing data

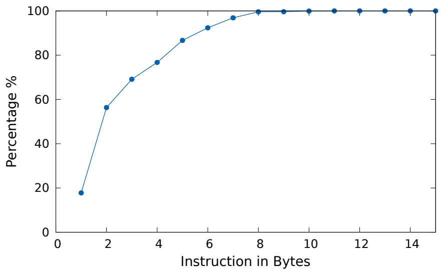

Fig. 4: CDF of the x86 Instructions sizes found on GNU Binutils.

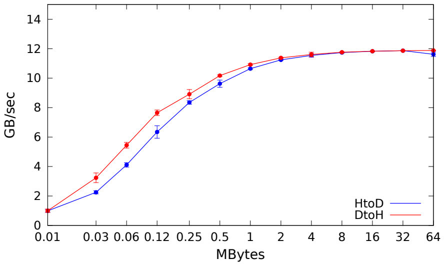

Fig. 5: PCIe 3.0 Transfer Throughput.

exchanges). In this experiment, we use three different inputs: (i) linear disassembly of synthetic binaries, (ii) linear disassembly of binaries corpus, and (iii) exhaustive disassembly of a subset of the corpus.

Cache Hit Rate in L2	
Buffer optimized size	Average Hit Rate %
16 Bytes	58.70
32 Bytes	53.65
64 Bytes	45.26

Table 1: Average hit rate at L2 cache for all read requests from L1 cache, when decoding 2-byte instructions in the GPU.

Optimization	MLines/Sec.	performance Gained %
No Optimization	52.05	-
Improve Cache Hits	65.51	+25.85 %
Structs of Arrays	43.85	-15.75 %

Table 2: Impact of *Access To Global* optimizations, when decoding 2-byte instructions in the GPU.

Instruction Size	MLines/Sec. CPU	Performance Dropped CPU %	MLines/Sec. GPU	Performance Dropped GPU %
1	35.90	-	100.91	-
2	14.12	60.6	66.67	33.93
4	12.63	64.81	59.53	41.00
8	9.96	72.25	46.32	54.09

Table 3: Effect of instruction sizes in decoding.

Synthetic Binaries In this experiment, we aim to evaluate our various optimizations and the effect of instruction-size. First, we generate buffers including 2-byte instructions, which is the most common instruction length (about 38.54% in our dataset, see Fig. 4), and measure how the buffer size used in decoding affects the L2 cache hit rate, when using 4096 threads. Table 1 shows that the optimal buffer size is 16 bytes. Table 2 shows the performance gained in accessing global memory by each of the optimizations described in Sec. 4.

Exhaustive Disassemble Results	
Description	Speedup
Average	4.411
Standard Deviation	0.928
Maximum	7.122
Minimum	2.729

Table 4: Exhaustive disassembly of 101 binaries from the corpus. GPU speedup results compared to the CPU. (Includes only instruction decoding.)

Threads	Performance MLines/sec
512	3.096
1024	4.857
2048	9.335
4096	17.548
8192	28.053
16384	28.085
CPU performance: 13.933	

Table 5: End-to-end disassembly of binaries corpus. Overall Performance in MLines/sec. (Data transfer buffer is 8192 Bytes)

As mentioned in section 2.2, the size of an x86 instruction can be between 1-15 bytes. Figure 4 shows the cumulative distribution function (CDF) of instruction sizes in the binaries used in the evaluation. In order to understand how binaries containing a mix of instructions with different sizes will affect performance, we decode files containing instructions with different sizes, with each file containing only a single length of instructions. We also compare decoding throughput by running our decoder both on the CPU and GPU, and try different numbers of threads on the GPU. We again use 4096 threads in the GPU, as we found that is is optimal in the synthetic binaries scenario. Table 3 lists the results of this experiment.

Linear Disassembly of Binaries In this experiment, we evaluate the GPU performance on disassembling the binaries in our corpus. This is likely to be the common use case of our prototype on large scale binary analysis. Each thread is assigned a different binary for disassembling. In Figure 6, we plot the speedup gained when offloading the disassembling process to the GPU. We evaluate several configurations, i.e., bytes per thread and number of threads, in order to find the best configuration. We can see that the GPU reaches maximum performance on different number of threads (8192) than with the synthetic binaries (4096). We also observe that the performance on different binaries drops to 28.4 *MLines/sec* compared to decoding all 8-byte instructions in the GPU (46.32 *MLines/sec*). This performance loss happens due to the different memory stalls that occur to each thread at a given moment. Threads decode different sizes of instructions when disassembling binaries, as a consequence they do misaligned accesses to the global memory and the cache misses increase. Still, by increasing the threads per multiprocessor we can hide some of these stalls and therefore the disassembler scales up to 8,192 threads. From the other hand, just spawning threads is not enough for hiding all the stalls. Spawning more threads arises more races to the caches and more cache misses for the concurrent cache lines. Lastly, by decoding different instructions, we slightly increase the branch divergence that also creates stalls. As we can see in Figure 6 the GPU was ≈ 2 times faster on the disassembly process than a relevant high-end CPU. Performance stops scaling after 8192 threads which we can safely state that this is the optimum configuration for the disassembly process.

Exhaustive Disassembly of Binaries In this experiment we disassemble each binary starting from each byte in order to find all possible instructions included in the binary. The evaluated prototype is the one described in Section 3.2. We evaluate the prototype using several number of threads in order to find the optimal for this case. The best performance is reached when we spawn 131,072 threads. Therefore, the exhaustive prototype, shall perform better, if we disassemble binaries of size bigger than the threads we spawn. In case the binary is smaller than the optimal amount of threads we spawn as many threads as the size of the binary. As we saw the disassemble performance differs among different sizes of instructions. In order to be accurate, we exhaustive disassemble a set of 101 binaries and evaluate the achieved performance. In Table 4 we can see the results of the experiment described, on disassembling binaries exhaustively. The average speedup we gained is 4.411 with a standard deviation of 0.928.

Overall Performance In this section, we evaluate our prototype in an end-to-end scenario. As mentioned in Section 3, we use streams in order to pipeline the operations and hide communication costs. We measure the time spent for each component in isolation. For all subsequent experiments with use 8,192 threads, as this configuration achieves the best performance, as we have shown in Section 5.1. In Figure 7 we can see the raw times of the corresponding components stacked in the order they execute in a given stream, pipelined with the current disassembly process of the previous stream. When the number of threads is lower than 1024 we can see that the bottleneck operation is pre-processing. However, after 1024 bytes per thread we can see that the disassembly component becomes the bottleneck of the whole process. Therefore, pipelining does not reduce performance. In Table 5 we demonstrate the raw performance in MLines/sec of the GPU in several threads with the size of the input buffer at 8192 bytes per thread.

Hybrid Model: We also evaluate the performance of utilizing all CPU cores and the GPU to massively disassemble binaries. Despite the fact that the GPU is an independent processing system, it still requires interaction with the CPU for transferring data, spawning the GPU kernel for execution, etc. Therefore, when we over-utilize the CPU with workload, we increase the probability of having threads stalled due to context switching. At the evaluation process, by overloading the CPU we experience an increase in the pre- and post-processing overhead and so, we wasted time by having idle the GPU and decrease the overall performance. In order to evaluate properly the hybrid model we assigned one CPU thread to the GPU processes (pre, post, GPU invocation and interaction) and the rest for disassembly on the CPU. In Figure 8 we can see the performance on different devices and the hybrid model as described. The hybrid model achieved the performance of 37.336 *MLines/sec* which is 2.67 times faster than having only the CPU utilized and 1.32 times faster than the GPU implementation. The divergence of the hybrid model from the ideal performance is due to the assigned thread to the GPU controlling processes.

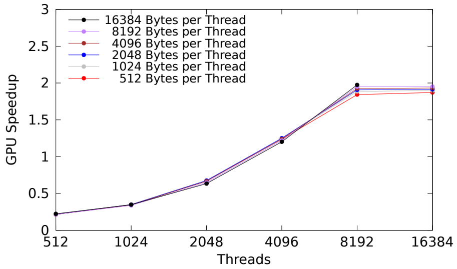

Fig. 6: GPU-Disassembler speed up compared to the CPU on different set ups. Comparing only the disassemble process without the transfers.

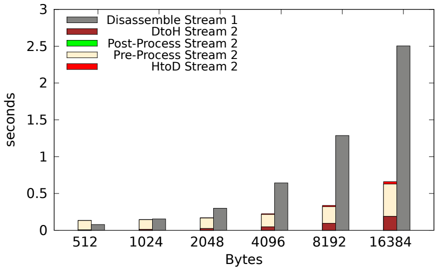

Fig. 7: The disassembly components of the GPU pipelined using streams. Focused on 8192 Threads.

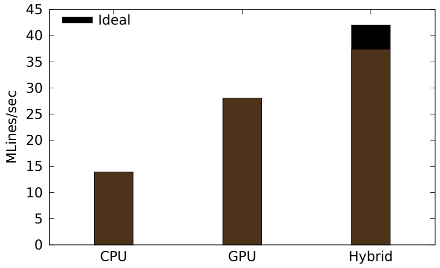

Fig. 8: The overall performance on CPU, GPU and on both processors. Focused on 8192 Threads for the GPU.

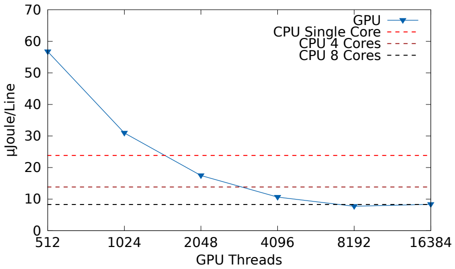

Fig. 9: Power Consumption per decoded line.

5.2 Power Consumption and Cost

Power Consumption per line In this experiment we measure the power consumption of our prototype at a given moment, with the components pipelined, when disassembling binaries. For the comparison we define the metric Joules consumed per decoded line. We evaluate the watts consuming per second and the performance of the tool as defined in previous sections (Lines/sec). By dividing these values we come up with Joules consumed per decoded line (Joules/line). In Figure 9 we demonstrate the power consumption efficiency for the GPU and CPU in different threads. For the measurement of the power consumption we used sensors that can measure the power consumption of the CPU, the PCI bus, the RAM and peripherals. For each set up, we sum up the power consumed at a given moment and then we calculate the power consumed per decode line. Both of the devices perform similar in terms of power consumption per decoded line. GPU consumes 8.34 μJ at the best configuration for decoding an instruction.

Lines per Dollar For our hardware setup, we have selected relatively high-end devices; for the CPU we used an Intel(R) i7-3770 which costs around $305, and the NVIDIA GTX 770 graphics card with similar cost at $396.[4] [5] These are

[4] Cpu benchmarks: Intel core i7-3770 @ 3.40ghz. `http://www.cpubenchmark.net/`

[5] Videocard benchmarks: Geforce gtx 770. `http://www.videocardbenchmark.net/`

the prices at the time this work was published. The total system cost is around $1120 with the current values of the components. Our prototype performs with an overall cost of 23.36 KLines/$.

6 Related Work

The improvement of the disassembly process for the `x86` and `x86-64` architecture is still an open issue. There are various publications that address disassembly correctness and effectively differentiate code from alignment patching bytes inside the text section of the binaries. Most of these publications, are based on a similar approach. They use the targets of control-flow instructions in order to recognize the regions of basic blocks and functions borders. They make several disassembly passes on these code regions until the given conditions of correctness are satisfied. Finally, they construct the final call graph and discard the unreachable regions [9, 13, 18, 24]. However, there is also a dynamic approach that leverages machine learning techniques [4, 11]. This approach uses decision trees, that are constructed by feeding binaries, compiled from various compilers and optimization flags as training sets. They perform exhaustive disassembly on the binary to produce all the possible assembly output. Lastly, they use the constructed tree to match and recognize the entry and exit points of functions.

GPUs continuously become more powerful and with extended computational capabilities that can support more applications. In the scientific community, there are several security analysis tools that exploit the parallelism offered by GPUs for fast processing such as network packet processing [10, 20, 21].

7 Limitations

The implementation of our prototype comes with limitations. The size of the decoded instructions for all the threads can be enormous and as a result, we can easily run out of memory. Also, memory constrains occur on the fast memory interfaces such as constant memory, shared and register usage per thread. Furthermore, GPU limitations with regards to dynamic memory allocation, forces us to use static allocation and requires rewriting of the dynamic parts of the disassembler.

We are unable to further exploit GPU parallelization due to memory stalls that occur at decoding time. GPU threads, make arbitrary accesses to memory at the decoding process which under-utilize the access throughput. Although, we can hide memory stalls by spawning more threads, there is a limit on how the cuda-process scales. The GPU hides stalled threads by context-switching to threads that are ready to execute. However, complex programs, that have high needs in resources and frequently access memory, can generate more stalls when excessively utilizing threads. Thus, it is not trivial to determine the optimal number of threads for a GPU-Disassembler; it really depends on the implementation and the disassembly algorithm (linear, exhaustive, etc.).

8 Conclusion

GPUs are powerful co-processors, which we can use to accelerate computationally intensive tasks like binary disassembly through parallelization. In this work we have built a GPU based x86-disassembler that exploits the hardware features offered from GPUs to accelerate disassembly. We evaluate our GPU-based x86-disassembler in terms of performance and cost. Our prototype performs two times faster in linear disassembly and 4.4x faster in exhaustive disassembling of the same binary compared to a CPU implementation. In terms of performance over power consumption; GPU performs similar with a full utilized CPU at 8.34 $\mu J/Line$.

9 Acknowledgments

We want to express our thanks to the anonymous reviewers for their valuable comments. This work was supported by the General Secretariat for Research and Technology in Greece with the Research Excellence grant GANDALF, and by the projects NECOMA, SHARCS, funded by the European Commission under Grant Agreements No. 608533 and No. 644571. This work was also partially supported by the US Air Force through contract AFRL-FA8650-10-C-7024. Any opinions, findings, conclusions or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government or the Air Force.

References

1. New 25 GPU Monster Devours Passwords In Seconds. `http://securityledger.com/new-25-gpu-monster-devours-passwords-in-seconds/`.
2. {GPU} accelerated monte carlo simulation of the 2d and 3d ising model. *Journal of Computational Physics*, 228(12):4468 – 4477, 2009.
3. Gogul Balakrishnan and Thomas Reps. Wysinwyx: What you see is not what you execute. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 32(6):23, 2010.
4. Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. Byteweight: Learning to recognize functions in binary code. *Proceedings of USENIX Security 2014*, 2014.
5. Bryan Catanzaro, Narayanan Sundaram, and Kurt Keutzer. Fast support vector machine training and classification on graphics processors. In *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, pages 104–111, 2008.
6. Chris Eagle. *The IDA pro book: the unofficial guide to the world's most popular disassembler*. No Starch Press, 2008.
7. Intel Intel. and ia-32 architectures software developer's manual, volume 3b: System programming guide. *Part*, 1:2007, 64.
8. Aditya Kapoor. *An approach towards disassembly of malicious binary executables*. PhD thesis, University of Louisiana at Lafayette, 2004.

9. Johannes Kinder. Static analysis of x86 executables. 2010.
10. Lazaros Koromilas, Giorgos Vasiliadis, Ioannis Manousakis, and Sotiris Ioannidis. Efficient software packet processing on heterogeneous and asymmetric hardware architectures. In *Proceedings of the 10th ACM/IEEE Symposium on Architecture for Networking and Communications Systems, ANCS*, 2014.
11. Nithya Krishnamoorthy, Saumya Debray, and Keith Fligg. Static detection of disassembly errors. In *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on*, pages 259–268. IEEE, 2009.
12. Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. Polymorphic worm detection using structural information of executables. In *Recent Advances in Intrusion Detection*, pages 207–226. Springer, 2006.
13. Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. Static disassembly of obfuscated binaries. In *USENIX security Symposium*, volume 13, pages 18–18, 2004.
14. NVIDIA. *CUDA C Programming Guide, Version 5.0.*
15. CUDA NVidia. C best practices guide. *NVIDIA, Santa Clara, CA*, 2012.
16. Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Proceedings of the 33rd IEEE Symposium on Security & Privacy (S&P)*, May 2012.
17. Thanasis Petsas, Antonis Papadogiannakis, Michalis Polychronakis, Evangelos P Markatos, and Thomas Karagiannis. Rise of the planet of the apps: A systematic study of the mobile app ecosystem. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 277–290. ACM, 2013.
18. Edward J Schwartz, J Lee, Maverick Woo, and David Brumley. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *Proceedings of the USENIX Security Symposium*, page 16, 2013.
19. John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(1-3):66–73, 2010.
20. Giorgos Vasiliadis, Spiros Antonatos, Michalis Polychronakis, Evangelos P. Markatos, and Sotiris Ioannidis. Gnort: High performance network intrusion detection using graphics processors. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection (RAID)*, September 2008.
21. Giorgos Vasiliadis, Lazaros Koromilas, Michalis Polychronakis, and Sotiris Ioannidis. GASPP: A GPU-accelerated stateful packet processing framework. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, June 2014.
22. Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis. MIDeA: A multi-parallel intrusion detection architecture. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)*, October 2011.
23. Henry Wong, M-M Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. Demystifying gpu microarchitecture through microbenchmarking. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, pages 235–246. IEEE, 2010.
24. Mingwei Zhang and R Sekar. Control flow integrity for cots binaries. In *USENIX Security*, pages 337–352, 2013.