A Consumer Application for GPGPUs: Desktop Search

Ateeq Sharfuddin
Department of Computer Science
The George Washington University
Washington, DC, 20052
pikachu@gwmail.gwu.edu

Xiaofan Feng
Department of Computer Science
The George Washington University
Washington, DC, 20052
xiaofan@gwmail.gwu.edu

Abstract

To date, the GPGPU approach has been mainly utilized for academic and scientific computing, for example, for genetic algorithms, image analysis, cryptography, or password cracking. Though video cards supporting GPGPU have become pervasive, there do not appear to be any applications utilizing GPGPU for a household user. In this paper, one consumer application for GPGPU is described: utilizing GPGPUs for Desktop Search. Though the implementation is somewhat rudimentary, it still demonstrates a sizable performance gain.

1 Introduction

Most video cards manufactured today support OpenCL[1, 2], CUDA [3], or DirectCompute [2, 4]. These mechanisms allow a video card's graphics processing units (GPUs) to be utilized for general purpose computing, an approach which is termed GPGPU. The GPGPU approach has been used extensively in the scientific and academic community for evolutionary algorithms [5], image analysis [6, 7], or brute-force password cracking [8, 9]. Though video cards supporting GPGPUs have become pervasive, to a common household user, video cards still only provide image rendering support. In this paper, we demonstrate one consumer application to which the GPGPU approach could be applied: Desktop Search. There are quite a few works detailing applications of GPGPU, such as for fluid simulation [10], object detection [7], and linear genetic programming [5], to name a few. Similarly, there are quite a few works on Desktop Search [11, 12, 13]. However, to our knowledge, no research has been performed that applies GPGPUs for Desktop Search.

Given that a majority of the consumers use Microsoft Windows, we specifically tailor this paper towards the NT File System (NTFS). First, the standard approach to finding files on Microsoft Windows is described. Then some intricacies of the NT File System (NTFS) are explained. Following that, an approach tailored to utilize these intricacies to find files faster is described. Finally, experimental results are provided, comparing the performance of this approach utilizing GPGPU to a CPU implementation.

The implementation for the experiment described in this paper is rudimentary, and only the performance of case-insensitive filename substring queries is demonstrated. The goal of this paper is to demonstrate a consumer application of GPGPUs, and to expand upon this idea further at a later date, by supporting multiple simultaneous queries on multiple attributes. A preliminary version of this Desktop Search, though marginally fast, won an Honorable Mention Prize at the AMD OpenCL Innovation Challenge last year. We describe this version and also how improve this particular approach.

2 Querying for a File in Windows

The Windows API provides a set of functions (e.g., `FindFirstFile`, `FindNextFile`, `FindClose`, as well as a few others), which one can use to query for files in a volume. For example, to query for *all* files having a case-insensitive `.txt` extension in the `C` drive, one might call a recursive function similar to the one described in Procedure 1 with "c:\" and ".txt" as the first and second arguments, respectively.

This approach is relatively simple. Regardless of any compiler or hand-coded optimizations performed for Procedure 1, for obvious reasons (such as the small read requests and the number of ring-0 transitions), this is not the fastest approach to finding files, particularly on NTFS. One can simulate this procedure by performing a

```
dir C:\*.txt /S
```

from the command line.

3 NTFS

The native file system format for Windows is NTFS, and a thorough examination of this file system is provided by Mark Russinovich and David Solomon in Windows Internals [14]. NTFS has a metadata file called the Master File Table (*$Mft*), which is a *logical* array of file references. This *$Mft* contains a file reference for each file on the volume, including the *$Mft* itself and other metadata

Procedure 1 find ($path$, $query$, $result$)

```
data ← {0}
more ← TRUE
(f, data) ⇐ FindFirstFile( path + "*.*" );
if f = INVALID_HANDLE_VALUE then
  return result
end if
repeat
  if data.cFileName = "." then
    continue
  end if
  if data.cFileName = ".." then
    continue
  end if
  if isDirectory(data) then
    result.add (find(path + data.cFileName + "\",
    query, result))
  else if stristr(data.cFileName, query) ≠ 0 then
    result.add(path + data.cFileName)
    {assume stristr is case-insensitive strstr}
  end if
  (more, data) ⇐ FindNextFile(f)
until more = FALSE
FindClose(f)
return result
```

files. Each file reference record in the *$Mft* has a fixed-size of 1024 bytes, and if a file has a large number of attributes, there may be more than one file reference in the *$Mft* for that file. The first 42 bytes of a file reference is the header (see FILE_RECORD_SEGMENT_HEADER in Figure 1). Immediately following these first 42 bytes is the update sequence array. And the remaining bytes in the file reference are utilized for *attributes*.

Each unit of information associated to a file, such as its name, owner, timestamps, content, etc. are implemented as *attributes*, and each type of *attribute* follows its own structure. In this paper, we only perform file name queries– as such, only the `FILE_NAME` attribute structure is of interest. More details regarding these structures and other NTFS structures are available at [16, 17, 18, 19, 20, 21].

4 Large Read Requests

Given that the *$Mft* is an array consisting of 1KB fixed-size file references, a large chunk of the *$Mft* could be read at once. For example, a one megabyte read request from the *$Mft*, would contain 1024 file references. As a consequence, this approach also minimizes the number of ring-0 transitions.

```
struct MULTI_SECTOR_HEADER
{
  UCHAR   Signature[4];
  USHORT  UpdateSequenceArrayOffset;
  USHORT  UpdateSequenceArraySize;
};

struct FILE_RECORD_SEGMENT_HEADER
{
 MULTI_SECTOR_HEADER    MultiSectorHeader;
 ULONGLONG              Reserved1;
 USHORT                 SequenceNumber;
 USHORT                 Reserved2;
 USHORT                 AttributeOffset;
 USHORT                 Flags;
 ULONG                  Reserved3[2];
 FILE_REFERENCE         BaseFileRecord;
 USHORT                 Reserved4;
 ...
};
```

Figure 1: FILE_RECORD_SEGMENT_HEADER and MULTI_SECTOR_HEADER structures

5 A CPU-based Approach

The CPU-based approach reads the entire *$Mft* by performing large asynchronous read requests. The approach consists of a main thread and a pool of worker threads. The main thread performs read requests, and the worker threads are in a waitable state. When the operating system completes a requested read, a worker thread is awoken and provided the data. This worker thread analyzes this provided data and, when complete, returns to a waitable state (unless new work is readily available). The CPU-based approach's worker thread is described in Procedure 2. Certain details such as security on NTFS, error-handling, "patching," the "in use" flag, resident vs. non-resident attributes, attribute traversing, etc. have been abstracted away from the pseudocode of this paper; the problem and solution described in this paper can be introduced without detailing these concepts.

Procedure 2 findCPU ($query$, $records$)

```
results ← {0}
for all i in records do
  filename ⇐ nameAttribute(i)
  if existsAttribute(filename) then
    results.append(matched(filename, query))
  end if
end for
return results
```

A file reference could have multiple `FILE_NAME` structures. The function `nameAttribute` re-

turns the longest file name when available. The `existsAttribute` function ensures that *filename* was found. And the `matched` function uses a case-insensitive character-based Boyer-Moore-Horspool (BMH) [22] implementation to find the *query* in the *filename*. This CPU approach achieves much better results than the "dir" approach shown in Section 2. The reasons are quite simple: large read requests (so fewer ring-0 transitions); reads are asynchronous; and reads are aligned at sector boundaries, which allow us to perform reads without intermediate buffering by the operating system. The CPU approach can be modified slightly for the GPGPU approach.

6 A GPGPU Approach

If we take the example from the previous section, where a one megabyte read request contained 1024 file references, on a video card with 1024 GPUs, all 1024 file references could be analyzed simultaneously. Let us say that the cost is $C(R_i)$ to process a file reference R_i. Assume that we have n file references labeled $R_1, R_2, R_3, \ldots, R_n$, and n GPU cores labeled $G_1, G_2, G_3, \ldots, G_n$. We can assign core G_i to process file reference R_i. The cost for concurrently processing all n file references would be $max\{C(R_i) \mid i = 0, 1, 2, \ldots, n\}$ [15]. This GPU-based approach is described in Procedure 3. Procedure 3 first enqueues a write of file references (or *records*) into video-card memory. Then this procedure enqueues an execution of `match`. Following that, Procedure 3 enqueues an execution of reduce, and waits for *results* to be read back into main memory. The first element of *results* contains the number of matches found in this set of records. The remaining elements (from 1 up to and including *count*) contain the indices of the file references which matched the query.

Procedure 3 findGPU1 ($query$, $records$)

```
results ← {0}
enqueueWrite(records)
enqueueKernel(match)
enqueueKernel(reduce)
event ⇐ enqueueRead(results)
waitFor(event)
count ← results[0]
for i = 1 to count do
    filename ⇐ nameAttribute(record[i])
    results.append(filename)
end for
return results
```

The kernel `match` in Procedure 4 is identical to the **for**-loop's body in the CPU approach (Procedure 2): it identifies if a file reference is in use and locates the `FILE_NAME` attribute with the longest file name, and uses a GPU implementation of `matched` to find the user's query in the file name. In the event that a match occurs, `match` updates the shared global memory array $error$. The function `get_global_id` (which is an OpenCL function) returns a unique work item identifier id. For our purposes, assume that this id indexes into the file reference *records* array.

Procedure 4 match ($query$, $records$, $error$)

```
id ⇐ get_global_id(0)
filename ⇐ nameAttribute(records[id])
if existsAttribute(filename) ∧ matched(filename,
query) then
    error[id] ← 0
else
    error[id] ← 1
end if
```

The kernel `reduce` in Procedure 5 iterates through this shared global memory array *error*, and stores the number of matches in *results[0]*. The subsequent entries in the *results* array index into the file reference *records* array which matched the query. Assume that the call `atomic_inc`(*results*) atomically increments *results[0]* and returns the prior value held by *results[0]*.

Procedure 5 reduce($error$, $results$)

```
id ⇐ get_global_id(0)
if error[id] = 0 then
    k ⇐ atomic_inc(results)
    results[k + 1] ← id
end if
```

This approach was entered into the AMD OpenCL Innovation Challenge last year. This was a fully parallel GPU implementation: in addition to matching and reducing, attribute traversing and in-use checking were also performed in the GPU. The main performance bottleneck was data-transfer: large read requests were being copied from main memory to video-card memory. As a result, for file-name queries, performance was only marginally better than the CPU-based approach described in Section 2.

7 An Improved GPGPU Approach

A file name can be at most MAX_PATH (which is defined as 260) characters. Therefore, for queries specific to file names, the entire 1KB file reference does not need to be transferred into video-card memory. If a *request* contains n file references, an array of size n is created, with each element being MAX_PATH characters large. This array is populated with the largest file name of each

file reference when available. In the event that the file reference is not in use or does not have `FILE_NAME` attributes, the respective element's first character is zeroed. This new array is transferred to the video-card memory, and the kernel `match2` is enqueued. With this approach however, attribute traversing and in-use checking has to be performed on the CPU prior to building this array, since the full 1KB file reference is not being transferred. The pseudocode is provided in Procedure 6. A second pool of threads is also utilized: Threads in this pool only wait for reads from video-card memory to complete, and update the user interface. This decoupling allows the worker threads to focus on read completions from the operating system and not block waiting for the GPU to write results back into main memory.

Procedure 6 findGPU2 ($query$, $records$)

```
filenames ← {0}
results ← {0}
for all i in records do
    filename ⇐ nameAttribute(i)
    if existsAttribute(filename) then
        filenames.add(filename)
    else
        filenames.add({0})
    end if
end for
enqueueWrite(filenames)
enqueueKernel(match2)
event ⇐ enqueueRead(results)
post( threadpool2, callback, event, filenames
results )
```

The kernel `match2` has been modified to only perform compare and merged with `reduce`.

Procedure 7 match2 ($query$, $filenames$, $valids$)

```
id ⇐ get_global_id(0)
if matched(filenames[id], query) then
    k ⇐ atomic_inc(valids)
    valids[k + 1] ← id
end if
```

This `callback` function is called from the second threadpool and the results are posted to the user interface.

8 Experiment

To test our new approach, we updated the user-mode application originally submitted to the AMD OpenCL Innovation Challenge, and devised a simple experiment. The application provides a graphical user interface, which allows the user to select NTFS volumes,

Procedure 8 callback ($event$, $records$, $results$)

```
waitFor(event)
count ← results[0]
for i = 1 to count do
    filename ⇐ nameAttribute(record[i])
    results.append(filename)
end for
return results
```

query for file names, and utilize GPUs (via OpenCL™) or CPUs. When the user clicks "Search," the list box shows the file paths for all files located. The status bar shows the total time taken to complete the query. The user interface for our implementation is shown below, with a query performed for "microsoft." For this experiment, retrieving the full file path has been disabled, as this requires additional file reference lookups.

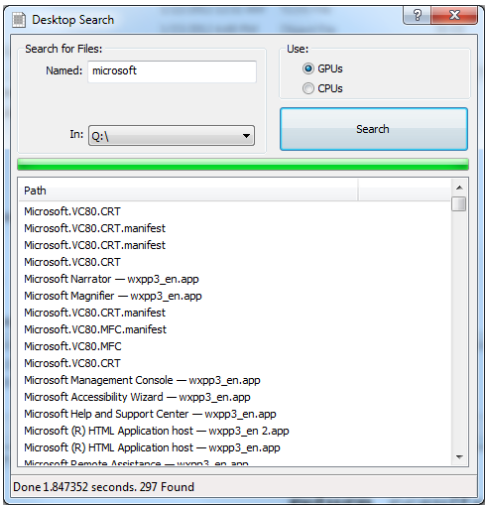

Figure 2: *The Desktop Search User Interface: Querying for "microsoft" on the Q drive.*

We generated 8,000 files on an NTFS volume, in 100 file increments, all containing the same five character extension. After each 100 files are created, we query for the files using the CPU approach and the GPGPU approach. The NTFS volume had a total of 234,240 file references at the end of the experiment (and 226,240 in the beginning). We compare the performance of GPU implementation described in the Section 7. The CPU implementation is almost exactly the same as the GPU implementation in Section 7, with the only difference being that second threadpool is not used, given that the CPU implementation does not need to wait for results to be read back into main memory from video-card memory.

We ran our experiment on a commodity machine with a Core Duo CPU (at 2.6GHz each) and 4 GB of DDR2

RAM with PCI Express 1.1 Bus. The video-card we utilized was an AMD Radeon HD 6870, with 1120 stream processors and 1 GB of DDR5 RAM. The hard drive we utilized was a Western Digital 7200 RPM drive, with 64MB cache.

9 Results

Table 1: Times in seconds, in 1,000 file increments

Number of Files	GPU Time	CPU Time
1000	1.82	1.87
2000	2.15	2.47
3000	2.34	2.81
4000	2.49	3.24
5000	2.63	3.53
6000	2.92	4.27
7000	3.08	4.64
8000	3.36	5.02

The results show that GPGPU approach is always faster than the CPU implementation. The cost of reading the entire *$Mft* from the hard drive is roughly 1.60 seconds. If we deduct this time from the results of both the approaches, the GPU approach is roughly 1.8 times faster on average, with minimum being 1.1 times and maximum being 3.4 times. Table 1 shows the performance in 1,000 file increments. Figure 3 shows the performance of the two implementations.

Note that this is a user-mode implementation– as such, the operating system scheduler will interrupt this application's threads as it sees fit.

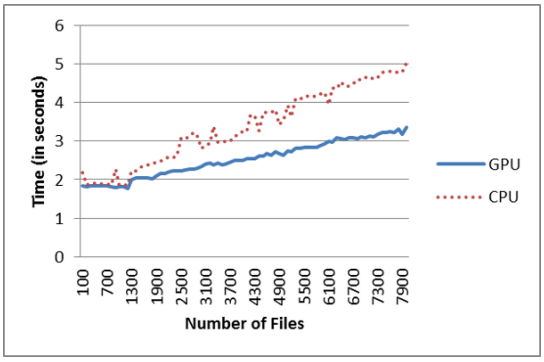

Figure 3: *Graph of our results. The CPU performance is in red, and the GPU performance is in blue, starting at 100 files and ending at 8,000 files. The GPGPU approach is always faster.*

We do not compare performance against the "dir" implementation, given that the "dir" approach takes minutes to complete and not seconds. We also do not compare our performance against other Desktop Search appliances; this is something we would like to compare against in the future. Most Desktop Search appliances rely on indexing: Windows Search can return no results when the files have not been indexed, and if non-indexed lookups is enabled, the speed is as slow as the "dir" implementation.

We could of course make the performance astronomically faster, by copying a large block of file references into video-card memory (say 900MB) prior to the user issuing a "Search" request. However, the intent of this paper was only to demonstrate a feasible consumer application using the GPGPU approach.

10 Future Work

The experiment designed for this paper is somewhat primitive. The approach taken to generate the files will, more likely than not, produce file references in a consecutive block in the *$Mft*. In the future, one goal is to uniformly distribute file references being queried for. Performance could also be tested on other hardware, for example, on a machine with a PCI Express 2.0 bus, and multiple video-cards, and ranges of video cards. We would also like to implement querying other attributes (such as timestamp queries, file owner, etc.) and multiple simultaneous queries (such as, *retrieve all .txt files created before 2010 by user 'x' having "microsoft" in the content*). And we would also like investigate indexing and index lookups with GPGPU.

11 Conclusion

Though video cards supporting GPGPU have become pervasive, there does not appear to be any applications utilizing GPGPU for the common household. This paper demonstrated a consumer application for GPGPUs: Desktop Search. A rudimentary implementation achieved an $1.8x$ performance gain. We aim to further develop on this idea– to design a more thorough experiment, and also to support more attributes for querying, and implement other available algorithms for searching.

12 Availability

The AMD OpenCL Innovation Challenge entry will be available as a sample in the AMD APP SDK: this is the fully parallel implementation. The improved GPGPU approach described in this paper can be derived from this fully parallel implementation.

References

[1] OpenCL™ Conformant Products, `http://www.khronos.org/conformance/adopters/conformant-products/`, Retrieved January 22, 2012.

[2] M. Ireton, OpenCL™ and Microsoft C++ AMP, `http://blogs.amd.`

com/developer/2011/07/07/opencl-and-microsoft-c-amp/ (2011), Retrieved January 22, 2012.

[3] CUDA Toolkit, http://developer.nvidia.com/cuda-toolkit, Retrieved January 22, 2012.

[4] DirectCompute Support on NVIDIA's CUDA Architecture GPUs, http://developer.nvidia.com/directcompute, Retrieved January 22, 2012.

[5] G. Wilson and W. Banzhaf, *Linear genetic programming GPGPU on Microsoft's Xbox 360*, IEEE Congress on Evolutionary Computation (2008) p. 378-385.

[6] Z. Yang, Y. Zhu, Y. Phu. *Parallel Image Processing Based on CUDA*, International Conference on Computer Science and Software Engineering (2008) p. 198-201.

[7] L. Zhang, R. Nevatia, *Efficient scan-window based object detection using GPGPU*, CVPRW (2008), p. 1-7.

[8] T. Murakami, R. Kasahara, T. Saito. *An implementation and its evaluation of password cracking tool parallelized on GPGPU*, International Symposium on Communications and Information Technologies (2010) p. 534-538.

[9] G. Hu, J. Ma, and B. Huang, *Password Recovery for RAR Files Using CUDA*, IEEE Conference on Dependable, Autonomic and Secure Computing 8 (2009) p. 486-490.

[10] E. Wu, Y. Liu, *Emerging technology about GPGPU*, APCCAS (2008), p. 618-622.

[11] S. Chernov, P. Serdyukov, P.-A. Chirita, G. Demartini, W. Nejdl, *Building a Desktop Search Test-Bed*, Advances in Information Retrieval 4425 (2007), p. 686-690.

[12] C. Lu, M. Shukla, S.H. Subramanya, Y. Wu. *Performance Evaluation of Desktop Search Engines*, IEEE International Conference on Information Reuse and Integration (2007), p 110-115.

[13] J. Chen, H. Guo, W. Wu, and C. Xie. *Search your memory! - an associative memory based desktop search system*, SIGMOD 35 (2009), p. 1099-1102.

[14] M. E. Russinovich and D. A. Solomon, *Microsoft® Windows® Internals, Fourth Edition*, Microsoft Press (2005) p. 717-785.

[15] AMD® Accelerated Parallel Processing OpenCL™Programming Guide (v1.3f), http://developer.amd.com/sdks/AMDAPPSDK/assets/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide.pdf, 2011, p 18-23. Retrieved January 22, 2012.

[16] FILE_RECORD_SEGMENT_HEADER structure, http://msdn.microsoft.com/en-us/library/bb470124(v=vs.85).aspx, Retrieved January 22, 2012.

[17] MULTI_SECTOR_HEADER structure, http://msdn.microsoft.com/en-us/library/bb470212(v=vs.85).aspx, Retrieved January 22, 2012.

[18] FILE_NAME structure, http://msdn.microsoft.com/en-us/library/bb470123(v=vs.85).aspx, Retrieved January 22, 2012.

[19] MFT_SEGMENT_REFERENCE structure, http://msdn.microsoft.com/en-us/library/bb470211(v=vs.85).aspx, Retrieved January 22, 2012.

[20] ATTRIBUTE_RECORD_HEADER structure, http://msdn.microsoft.com/en-us/library/bb470039(v=vs.85).aspx, Retrieved January 22, 2012.

[21] ATTRIBUTE_LIST_ENTRY structure, http://msdn.microsoft.com/en-us/library/bb470038(v=vs.85).aspx, Retrieved January 22, 2012.

[22] R. N. Horspool, *Practical fast searching in strings*. Software Practice and Experience 10 (1980) p. 501-506.