# Utilizing AMD APP SDK to Search for Files

*For AMD OpenCL Coding Competition 2011*

Ateeq Sharfuddin

ateeq.sharfuddin@gmail.com

## Abstract

This document details the work done for a prototype that searches through NTFS volumes quickly utilizing AMD APP SDK. Since this is a prototype (and given the time constraints), only a limited number of features are exposed. The main goal of this prototype was to demonstrate the feasibility of utilizing GPUs for the purposes of searching: one can conclude with the information provided in this document that this prototype succeeds in its mission. Not only does this prototype demonstrate feasibility, it also demonstrates an increase in performance even though only a few optimizations were implemented and the experiments were run on a low-end machine with PCIe 1.1 limiting data transfers. This document first describes how to use the provided prototype, followed by its design and some experimental results, and further work and applications.

## Contents

## Introduction

In this document, a prototype for searching for files in NTFS volumes is described. This prototype is different from the search feature in Windows in that: a) it does not index files, and b) it does not use the Win32 API FindFirstFile/FindNextFile functions. This prototype scans through the Master File Table (MFT) to locate the files matching a user's query.

The requirements to run the prototype and the platforms it has been tested on are first defined. Following that, the user interface and functionality are described. The design is then detailed, and some experimental results are provided. Finally, further work and applications for this prototype are specified.

## Requirements

This prototype has the following requirements:

1. Windows XP SP3 or Windows 7 SP1.
2. AMD APP SDK 2.5's OpenCL runtime (version 2.5.684.213 for runtime and SDK, and display driver version 8.881.0.0000).
3. Run as an Administrator on the machine.

With regard to the first requirement, this prototype has only been tested on Windows XP SP3 (only CPU mode), and Windows 7 SP1. No guarantees are made that this prototype will work on other flavors of Windows (though, one can speculate that this prototype should work fine on Vista). With regard to the OpenCL runtime, the prototype statically links with OpenCL.lib, which is provided with AMD APP SDK 2.5. Thus, if the OpenCL runtime is missing in the system, the DLL loader will issue an error (indicating the missing DLL) and exit. With regard to the third requirement, direct access to volumes is restricted to users who are Administrators, and direct access is necessary to parse the MFT. As such, when you double-click the program, a UAC consent dialog box appears on Windows 7.

If the three requirements are met, you should be greeted with a dialog box similar to Figure 1. I have only tested on GPU and CPU device types (CL_DEVICE_TYPE_GPU and CL_DEVICE_TYPE_CPU, respectively). I did not have the hardware to test on CL_DEVICE_TYPE_ACCELERATOR; as such, Accelerator device types are not guaranteed to work, and are not accessible via the GUI.

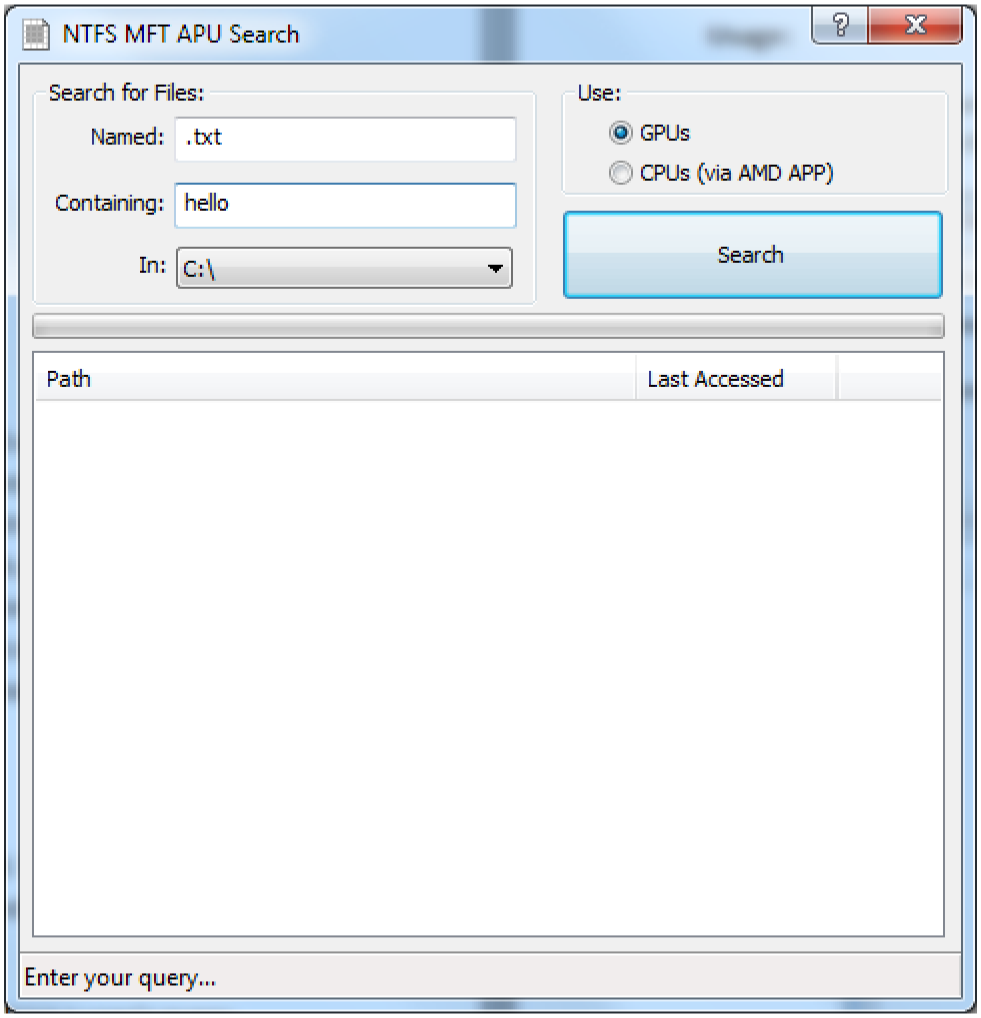


Figure 1: Main Dialog Box for the Prototype

## Platform Tested On

| Device Class | Description |
|---|---|
| **Motherboard** | G31M-S (PCIe 1.1 spec) |
| **Memory** | 4GB 800MHz DDR2 |
| **Hard drive** | WDC WD20EARS |
| **CPU** | Pentium E5300 @ 2.6GHz |
| **GPU** | AMD Radeon HD 6870<br>- Catalyst Software Suite 11.8<br>- AMD APP SDK v2.5 |

***Caveat: I unfortunately did not have a PCIe 2.0 spec hardware or an APU spec hardware. Thus, I could not compare performance against these particular hardware platforms. The expectation is that performance will be superior in these two platforms since most of the latency I have encountered was in reading and writing to device memory, and PCIe 2.0 has twice the bandwidth, and APUs support zero copy path. In fact, I could argue that only APUs would provide the best performance due to zero copy path.***

## Build Instructions

To build the solution, you will need Visual Studio 2010. The expectation is that an environmental variable AMDAPPSDKROOT is defined, as this project utilizes OpenCL.lib under "$(AMDAPPSDKROOT)\lib\x86" and <cl/opencl.h> which it expects to be under "$(AMDAPPSDKROOT)\include."

## Binaries

The release build of the binary (ntfs.amd.exe) has been provided under the .\bin folder.

## Usage

Though the initial design describes a command-line interface, I felt it appropriate to provide a graphical user interface for convenience instead. The *Named* and *Containing* fields take substrings. The *In* combo-box is populated with NTFS drives on startup. And the *GPUs* radio button is disabled if no supported graphics card is detected.

A query $Q$ can be defined as a function $Q\ (\ N, C, I, P\ ) \rightarrow R$ such that:

$N$ is a string over the Unicode alphabet, and $|N| = 256$,
$C$ is a string over the binary alphabet, and $|C| = 256$,
$I$ elements the set of fixed NTFS drives in the system, and
$P$ an element of the set { GPUs, CPUs }.
$R$ is a subset of $F(I)$, where $F(I)$ is the set of all tuples (Path, Last Accessed) representing all the files and their last accessed times in drive $I$.

Suppose you were looking for all files in the C drive having extension "txt" and containing "hello" in the contents utilizing GPUs. So, the query to be issued would be:

$Q$(".txt", "hello", "C:", GPUs)

The prototype would return to you the subset $R$ of $F$("C:") representing all the matches. This query $Q$ is presented in Figure 1 in the Requirements section.

The user provides substrings in the *Named* and *Containing* fields representing $N$ and $C$ in the query $Q$, respectively. The user selects the fixed NTFS drive (removable drives are not shown), representing $I$, and selects whether they wish to use GPUs or CPUs to run their query representing

*P*. The output *R* is presented in the list box at the bottom of the interface. If the machine does not have any GPUs, the GPUs radio button shall be disabled. Clicking the *Search* button initiates the query *Q* to be run utilizing *P*. And the green bar in the middle fills up from left to right indicating progress.

The filename *N* (or the *Named* field in the GUI) is matched as a case-insensitive substring. The content *C* (or the *Containing* field in the GUI) is matched as case-sensitive substring. Note that content *C* matching is performed in binary: that is, if a Unicode text file contains "hello" and we are searching for the string "hello" in the GUI, it will not be found since each character in the file contents is two bytes in size. Also, note that for this prototype, *N* and *C* are substring matches and ***NOT*** wildcard or regular expression matches.

## Notes for Comparing Performance

To compare performance, remember to perform the same query twice: the first query builds paths. The second time the query is performed, the paths built will be reused.

## Design

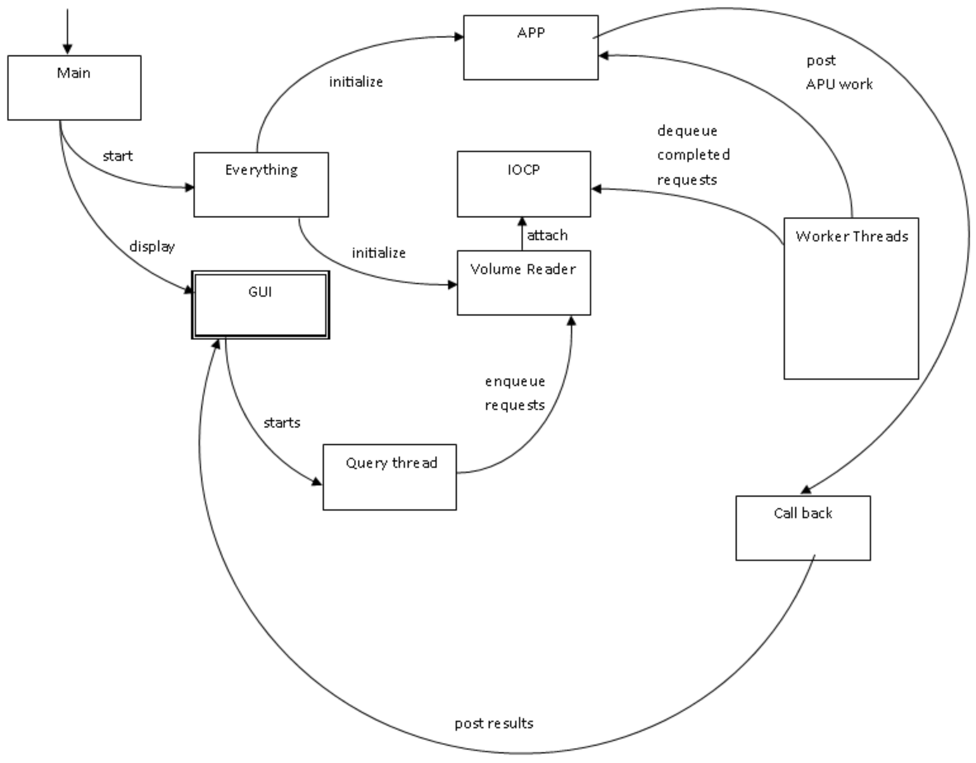


**Figure 2: Flow**

The overall design and flow for the prototype is abstractly defined in Figure 2. In the following subsections, a few key pieces of the diagram are described.

## Main

Main (WinMain in code) is the entry point, and this initializes all the components and displays the **GUI**. The components initialized are the memory pool, the IO completion port (**IOCP**) and the associated worker threads, the volume readers, and the OpenCL devices. Main also compiles the OpenCL code for the devices. The number of worker threads associated to the IOCP depends on the number of processors/cores on the system multiplied by two. Thus, if you have a machine with a single Core Duo processor, four threads will dequeue work from the **IOCP**. The **volume readers** are attached to all the fixed NTFS drives. The removable NTFS drives aren't attached purely for demonstration purposes.

## GUI

When "Search" is clicked in the GUI, a **Query thread** is spawned, and this thread posts read requests to the volume reader. The "Search" button is tri-state. If the user clicks "Search," its text changes to "Stop." If the user then clicks "Stop," its state changes to "Stopping." Finally, its state changes back to "Search" once the system accepts the stop request and finishes up queued up work.

## Query Thread

The query thread calls masterfiletable::findfiles, which sets up ***query-specific*** **cl_mem** objects.

If *N* (*Named* field) is non-empty, three **cl_mem** objects specific to *N* are created: *filenamePattern*, *filenameDelta1*, and *filenameLowerCase*. These are all CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR. In my case, given my hardware spec, CL_MEM_USE_PERSISTENT_MEM_AMD did not improve performance. As such, I did not use this flag.

If *C* (*Contains* field) is non-empty, two **cl_mem** objects specific to *C* are created: *filecontentPattern*, and *filecontentDelta1*. Similar to *N*, these are also created with CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR.

The query thread then issues non-blocking read requests to the drive that complete through the **IOCP**. These read requests are specific to clusters containing the MFT, and each read request is roughly 1MB in size (check ***readsize*** in the init function in main.cpp).

Finally, the query thread waits for all the issued reads to be completed.

## Volume Reader

The volume reader reads from a volume. Asynchronous work completion will be notified via the **IOCP**, and synchronous work will block.

## IOCP

The worker threads wait on this IO Completion Port (IOCP) for incoming work. The read requests queued by the Query thread, when completed, will be dequeued by these worker threads. These threads will then perform further processing and provide work to the GPUs/APUs to perform. Once the GPUs/APUs complete work, the remaining processing is performed through the event callback function.

A host worker thread first initializes the kernels in *masterfiletable*::*setup*, if they haven't already been initialized for the particular thread. *masterfiletable::setup* basically allocates a **cl_kernel** object for each kernel needed, and sets up each kernel's parameters and stores these for future calls (since the worker threads, when done with their tasks, dequeue work from the IOCP again). *masterfiletable::setup* also allocates **cl_mem** objects. These objects are: **records** (allocated with CL_MEM_READ_ONLY | CL_MEM_ALLOC_HOST_PTR), **errors** (allocated with CL_MEM_READ_WRITE), **attributes** (CL_MEM_READ_WRITE), and **reduction** (CL_MEM_READ_WRITE|CL_MEM_ALLOC_HOST_PTR). The worker thread first calls *clEnqueueWriteBuffer* on **records** cl_mem object with the MFT records it received from the IOCP. Then the worker thread calls *clEnqueueWriteBuffer* on **reduction** cl_mem object. Following this, the worker thread calls *inUseFn* kernel, *getAttributesFn* kernel , *matchFileNameFn* kernel (if query contains an *N* field), *reductionFn* kernel, and finally, a *clEnqueueReadBuffer* request on the **reduction** cl_mem object. All of these are non-blocking, and utilize events. A callback (*ClEvCallbackFn*) is associated to the final read from **reduction**, which will perform final processing, which includes file content matching if *C* is non-empty.

## AMD APP kernels

The bulk of the computation is performed by the abstract APP module in Figure 2. The worker thread passes the read request to the APP. Each read request can contain 1024 MFT records (each MFT record is 1K and ***readsize*** mentioned in **Query Thread** is 1MB). The APP performs the following work:

### inUse

This is the first kernel that is executed, and this kernel determines whether the particular MFT Record is in use (that is, not deleted). The error value for the record is updated (0 if in use and 1 if not in use) and propagated downstream.  The cost of this kernel per record is O(1).

### getAttributes

This is the second kernel that is executed, and this kernel locates the first three FILE_NAME attributes in the MFT record. Again, if one of the two kernels prior to *getAttributes* indicates the record to be an error, *getAttributes* does not perform work on the record. Similarly, if no FILE_NAME attribute is found in the record, *getAttributes* indicates the record to have an error value and propagates this downstream. A particular MFT record could theoretically have 40 attribute records (1024 bytes per MFT - ~44 bytes of header, and 40 "empty" resident attributes). Again, this could be stated to be O(1) per kernel.

### matchFileName

This is the third kernel that is executed **only** if *N* (or the *Named* field in the GUI) is non-empty. This kernel performs substring matches with *N* and the FILE_NAME attributes. If no matches can be found, the record is marked as an error. The substring matching algorithm is the Boyer-Moore-Horspool (BMH) algorithm, except for a table looking being used for case-insensitive comparisons.

The best-case runtime for BMH is $\Omega\ (n/m)$, where $n$ is the text length, and $m$ is the pattern length. This is usually the case that this prototype will encounter as the expectation is that we are looking for "a needle in a haystack" so to speak. Also, the maximum $n$ can be is 256 characters, per specifications.

### reduce

This is the next kernel to be executed, and this kernel returns an array back to the host in the following form:

$[n, a, b, c, d, e, \ldots ]$, where the size of this array is 1024 + 1.

The first value $n$ indicates the number of valid records in the array (i.e., the size of the array), and $a$, $b$, $c$, $d$, $e$, … are the offsets to the file records in the read operation that have not encountered an error so far. Note that if *matchFileName* is not executed, all file records are returned (with the exception of records which have invalid update sequence values or are not in use). We can say that the cost of this is again O(1), since there are at most 1024 records per read request.

The value of $n$ is updated by multiple cores, so it is incremented with *atomic_inc*.

### findx

If $C$ (or the *Containing* field in the GUI) is non-empty, the callback function will identify the clusters in the drive that represent the default data attribute of the particular file returned by *reduce*, and initiate the *findx* kernel to identify if the substring $C$ can be found in the data.

The algorithm used by *findx* is a variation of the BMH algorithm. The *findx* kernel is executed on the file contents —the work size is file content size divided by $|C|$. Each thread checks the byte at position get_global_id(0) * ($|C|$ - 1). Let us call this position *pos*. If *buffer*[*pos*] matches correctly with the pattern's $|C|$ - 1 element, *pos* is decremented, and a comparison is done with the pattern's $|C|$ - 2 element and *buffer*[*pos*]. This continues until either the entire pattern is matched or in the case that *buffer*[*pos*] does not match the respective position in the pattern. When this second case occurs, a $delta_1$[ *buffer*[ *pos* ] ] lookup is performed and stored in $v$. If this lookup value $v$ is not $|C|$ (i.e., the byte at *buffer*[ *pos* ] is not in the pattern), we shift *pos* to this position and try matching again. Otherwise, *findx* considers that a match was not found for this particular work.

This algorithm is a modified version of BMH, designed simply to return a match as quickly as possible. Each kernel will perform $2|C|$ comparisons in the degenerate case. And each kernel will perform only 1 comparison in the best case.

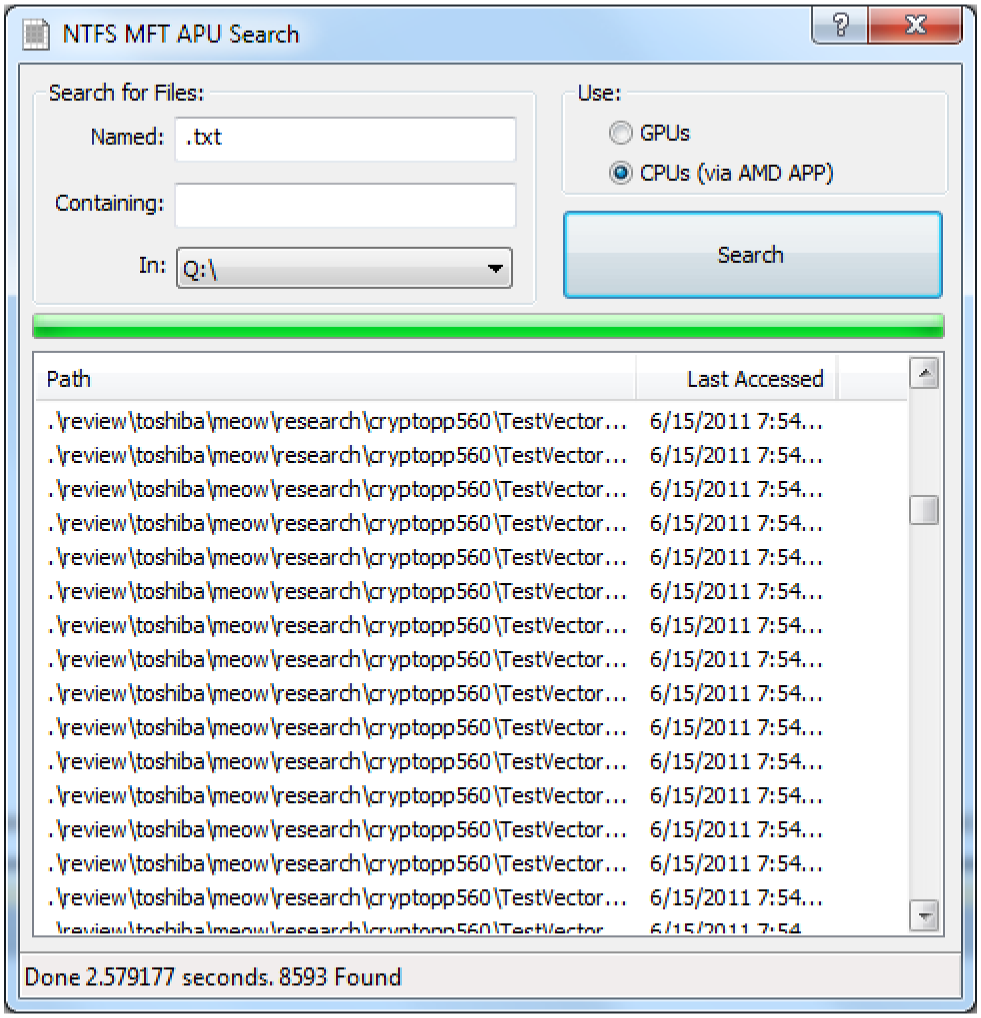
The callback function on the host is provided with the eventual outcome.

## Results

The callback function posts the results to the GUI. One thing to note is that the callback function stores the path references of the files found. This way, future queries, where some of the path hierarchy is the same, take less time.

## Experimental Results

Four sample queries are demonstrated. I then provide a table containing a run of 5 queries and 20 iterations. The overall time taken is visible in the status bar at the bottom of the GUI. The way I compute these numbers is by performing the same query more than once (since, as I mentioned in Results, the first time you query, the path information is built, and subsequent queries reuse this information).



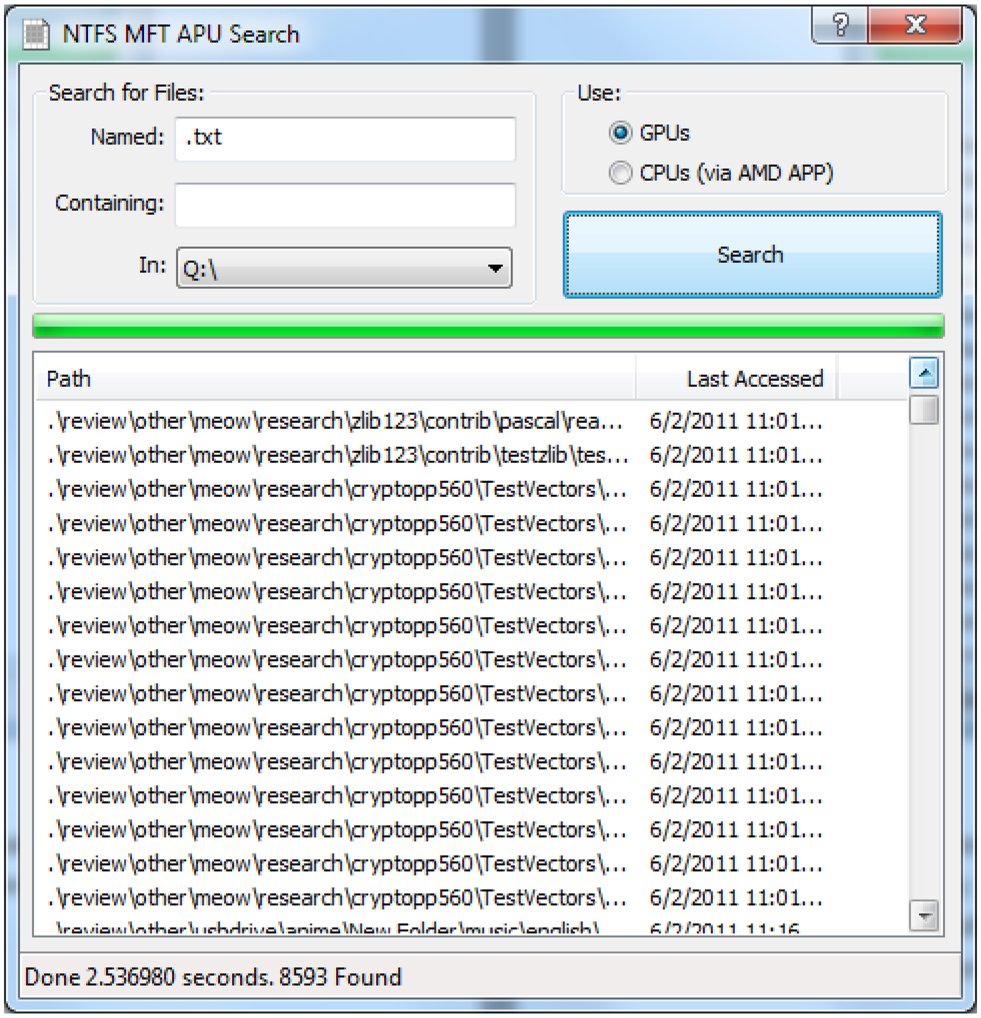
Search for Files:
Named: .txt
Containing:
In: Q:\
Use: GPUs [✓ CPUs (via AMD APP)]

| Path | Last Accessed |
|---|---|
| .\review\toshiba\meow\research\cryptopp560\TestVector... | 6/15/2011 7:54... |
| .\review\toshiba\meow\research\cryptopp560\TestVector... | 6/15/2011 7:54... |
| .\review\toshiba\meow\research\cryptopp560\TestVector... | 6/15/2011 7:54... |
| .\review\toshiba\meow\research\cryptopp560\TestVector... | 6/15/2011 7:54... |
| .\review\toshiba\meow\research\cryptopp560\TestVector... | 6/15/2011 7:54... |
| .\review\toshiba\meow\research\cryptopp560\TestVector... | 6/15/2011 7:54... |
| .\review\toshiba\meow\research\cryptopp560\TestVector... | 6/15/2011 7:54... |
| .\review\toshiba\meow\research\cryptopp560\TestVector... | 6/15/2011 7:54... |
| .\review\toshiba\meow\research\cryptopp560\TestVector... | 6/15/2011 7:54... |
| .\review\toshiba\meow\research\cryptopp560\TestVector... | 6/15/2011 7:54... |
| .\review\toshiba\meow\research\cryptopp560\TestVector... | 6/15/2011 7:54... |
| .\review\toshiba\meow\research\cryptopp560\TestVector... | 6/15/2011 7:54... |
| .\review\toshiba\meow\research\cryptopp560\TestVector... | 6/15/2011 7:54... |
| .\review\toshiba\meow\research\cryptopp560\TestVector... | 6/15/2011 7:54... |
| .\review\toshiba\meow\research\cryptopp560\TestVector... | 6/15/2011 7:54... |
| .\review\toshiba\meow\research\cryptopp560\TestVector | 6/15/2011 7:54 |

Done 2.579177 seconds. 8593 Found

**Figure 3: Q ('.txt', '', 'Q:\', CPUs)**

Figure 3 is a screenshot of a query for txt files in the C drive using CPUs. The overall time taken is visible in the status bar at the bottom of the GUI. I have scrolled down slightly to remove some personally identifiable documents.


NTFS MFT APU Search

Search for Files:

Named: .txt

Containing:

In: Q:\

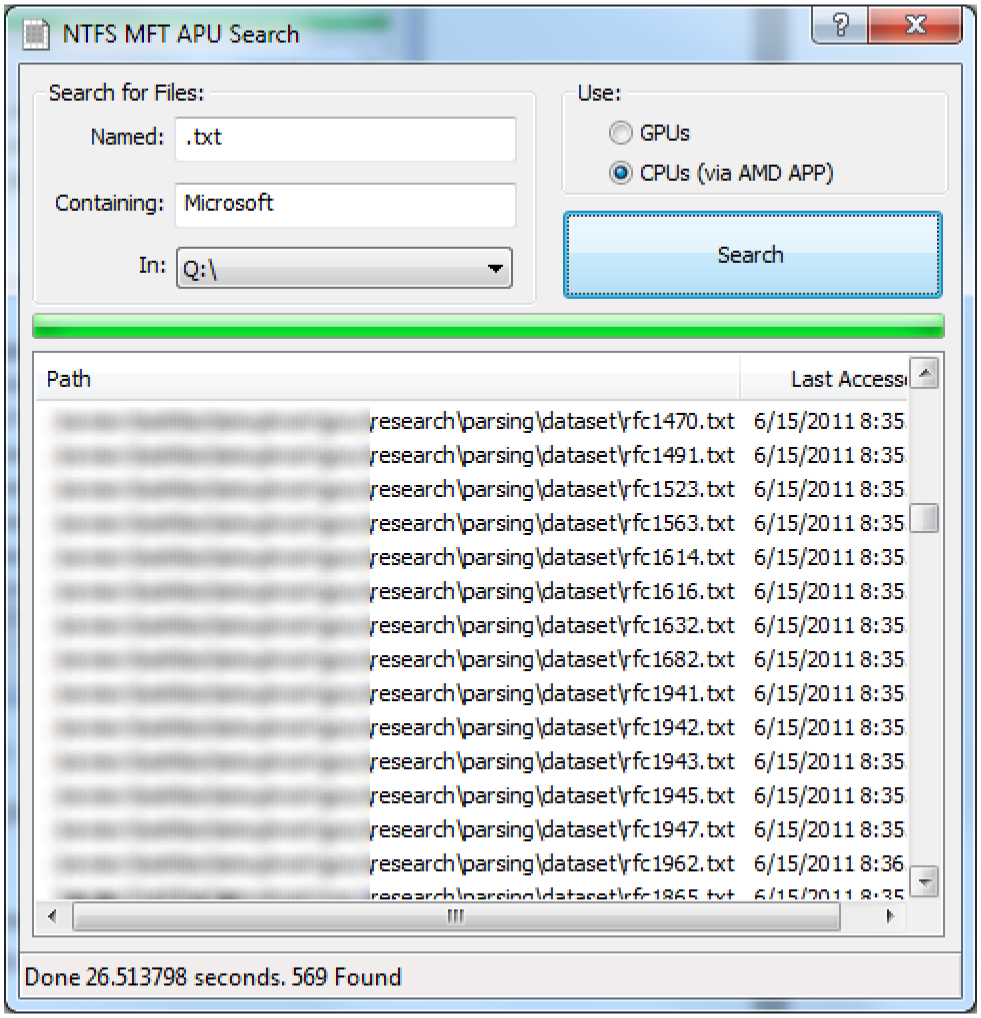
Use: [✓ GPUs] CPUs (via AMD APP)

| Path | Last Accessed |
|---|---|
| .\review\other\meow\research\zlib123\contrib\pascal\rea... | 6/2/2011 11:01... |
| .\review\other\meow\research\zlib123\contrib\testzlib\tes... | 6/2/2011 11:01... |
| .\review\other\meow\research\cryptopp560\TestVectors\... | 6/2/2011 11:01... |
| .\review\other\meow\research\cryptopp560\TestVectors\... | 6/2/2011 11:01... |
| .\review\other\meow\research\cryptopp560\TestVectors\... | 6/2/2011 11:01... |
| .\review\other\meow\research\cryptopp560\TestVectors\... | 6/2/2011 11:01... |
| .\review\other\meow\research\cryptopp560\TestVectors\... | 6/2/2011 11:01... |
| .\review\other\meow\research\cryptopp560\TestVectors\... | 6/2/2011 11:01... |
| .\review\other\meow\research\cryptopp560\TestVectors\... | 6/2/2011 11:01... |
| .\review\other\meow\research\cryptopp560\TestVectors\... | 6/2/2011 11:01... |
| .\review\other\meow\research\cryptopp560\TestVectors\... | 6/2/2011 11:01... |
| .\review\other\meow\research\cryptopp560\TestVectors\... | 6/2/2011 11:01... |
| .\review\other\meow\research\cryptopp560\TestVectors\... | 6/2/2011 11:01... |
| .\review\other\meow\research\cryptopp560\TestVectors\... | 6/2/2011 11:01... |
| .\review\other\meow\research\cryptopp560\TestVectors\... | 6/2/2011 11:01... |
| .\review\other\usbdrive\anime\New Folder\music\english\ | 6/2/2011 11:16 |

Done 2.536980 seconds. 8593 Found

**Figure 4: Q( '.txt', '', 'Q:\', GPUs)**

Figure 4 is a screenshot of the same query but using GPUs this time. Similar to the previous case, I have scrolled down slightly. The overall time taken is indicated in the status bar of the GUI. Notice that even with all the data transfers via PCIe 1.1, the query utilizing GPUs is still faster than the query utilizing CPUs. This should obviously be faster on PCIe 2.0 and APU hardwares. Next two screenshots of queries containing both *N* and *C* parameters are provided.



NTFS MFT APU Search

Search for Files:

Named: .txt

Containing: Microsoft

In: Q:\

Use:

GPUs [✓ CPUs (via AMD APP)]

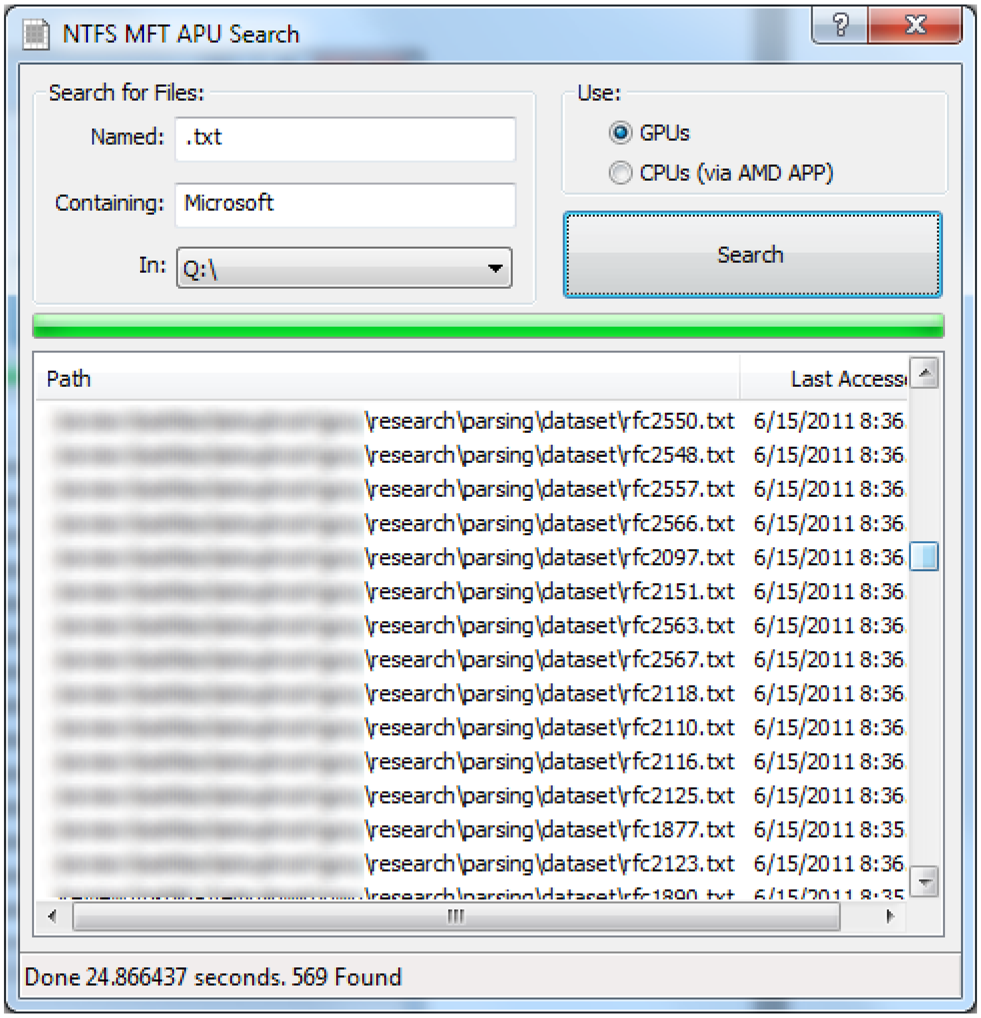
| Path | Last Access |
|---|---|
| [redacted]research\parsing\dataset\rfc1470.txt | 6/15/2011 8:35 |
| [redacted]research\parsing\dataset\rfc1491.txt | 6/15/2011 8:35 |
| [redacted]research\parsing\dataset\rfc1523.txt | 6/15/2011 8:35 |
| [redacted]research\parsing\dataset\rfc1563.txt | 6/15/2011 8:35 |
| [redacted]research\parsing\dataset\rfc1614.txt | 6/15/2011 8:35 |
| [redacted]research\parsing\dataset\rfc1616.txt | 6/15/2011 8:35 |
| [redacted]research\parsing\dataset\rfc1632.txt | 6/15/2011 8:35 |
| [redacted]research\parsing\dataset\rfc1682.txt | 6/15/2011 8:35 |
| [redacted]research\parsing\dataset\rfc1941.txt | 6/15/2011 8:35 |
| [redacted]research\parsing\dataset\rfc1942.txt | 6/15/2011 8:35 |
| [redacted]research\parsing\dataset\rfc1943.txt | 6/15/2011 8:35 |
| [redacted]research\parsing\dataset\rfc1945.txt | 6/15/2011 8:35 |
| [redacted]research\parsing\dataset\rfc1947.txt | 6/15/2011 8:35 |
| [redacted]research\parsing\dataset\rfc1962.txt | 6/15/2011 8:36 |
| [redacted]research\parsing\dataset\rfc1865.txt | 6/15/2011 8:35 |

Done 26.513798 seconds. 569 Found

Figure 5: Q('.txt', 'Microsoft', 'Q:\', CPUs)

Figure 5 is a screenshot demonstrating a query containing both *Named* and *Containing* fields utilizing CPUs.

And finally, Figure 6 is a screenshot demonstrating the same query but utilizing GPUs. Notice that the GPU takes less time than the CPUs even though this prototype performs a significant amount of data transfers (without any attention to alignment) and only a few computations running on PCIe 1.1.



NTFS MFT APU Search

Search for Files:

Named: .txt

Containing: Microsoft

In: Q:\

Use: [✓ GPUs] CPUs (via AMD APP)

Search

| Path | Last Access |
|---|---|
| [redacted]\research\parsing\dataset\rfc2550.txt | 6/15/2011 8:36 |
| [redacted]\research\parsing\dataset\rfc2548.txt | 6/15/2011 8:36 |
| [redacted]\research\parsing\dataset\rfc2557.txt | 6/15/2011 8:36 |
| [redacted]\research\parsing\dataset\rfc2566.txt | 6/15/2011 8:36 |
| [redacted]\research\parsing\dataset\rfc2097.txt | 6/15/2011 8:36 |
| [redacted]\research\parsing\dataset\rfc2151.txt | 6/15/2011 8:36 |
| [redacted]\research\parsing\dataset\rfc2563.txt | 6/15/2011 8:36 |
| [redacted]\research\parsing\dataset\rfc2567.txt | 6/15/2011 8:36 |
| [redacted]\research\parsing\dataset\rfc2118.txt | 6/15/2011 8:36 |
| [redacted]\research\parsing\dataset\rfc2110.txt | 6/15/2011 8:36 |
| [redacted]\research\parsing\dataset\rfc2116.txt | 6/15/2011 8:36 |
| [redacted]\research\parsing\dataset\rfc2125.txt | 6/15/2011 8:36 |
| [redacted]\research\parsing\dataset\rfc1877.txt | 6/15/2011 8:35 |
| [redacted]\research\parsing\dataset\rfc2123.txt | 6/15/2011 8:36 |
| [redacted]\research\parsing\dataset\rfc1890.txt | 6/15/2011 8:35 |

Done 24.866437 seconds. 569 Found

**Figure 6: Q( '.txt', 'Microsoft', 'Q:\', GPUs)**

**Table 1: Mean (in seconds) and Variance for Sample Queries**

| | *mean (CPUs)* | *variance (CPUs)* | *mean (GPUs)* | *variance (GPUs)* |
|---|---|---|---|---|
| Q('.txt', '', … ) | 2.612298857 | 0.000180259 | 2.562433857 | 0.000141148 |
| Q('.txt', 'Microsoft' … ) | 26.34424233 | 0.122749694 | 24.26426429 | 0.391470989 |
| Q('.pdf', …) | 1.998620667 | 0.001098233 | 1.931573333 | 6.99146E-05 |
| Q('.log', …) | 1.727371 | 0.025303839 | 1.616470667 | 0.000275624 |
| Q('.log', 'Microsoft', …) | 2.307173 | 0.000282868 | 2.1432002 | 5.38163E-05 |

In Table 1, you will notice that queries on GPUs are always faster than running queries on CPUs. Having run the APP Profiler, most of the time is occupied by writing and read to and from the device memory, so a performance increase should be noticed running this on newer hardware. Also, note that this is a prototype for a more full-featured product—more computationally intensive operations will be performed in the product.

## Further work

My plan is to expand on this obviously, and this is primarily for demonstration purposes. Only two fields are supported for a query, and this will be expanded (multiple simultaneous queries, and multiple simultaneous pattern searching, searching based on dates, wild card searching, searching specific to forensics, alternate data streams, searching other NTFS attributes, supporting other file systems, etc.).

I also did not get enough time to fully optimize this prototype—I haven't attempted pipelining or stream-lining memory access. I unfortunately have to submit this a few days early. What can I say? We're expecting a hurricane on the East coast, and I'm not sure if I will have power later this week.

## Applications

At the moment, this prototype demonstrates consumer desktop searching. The ultimate goal is to add features that would support other domains such as forensics, data-mining, and information retrieval outside of desktop searching.

## Conclusion

I have demonstrated that even with the amount of data transfer occurring, the queries that run on the GPU are faster than the queries that run on the CPU. Note that further work needs to be done: I was limited by time. I did not spend much time optimizing code or pipelining. The data presented was run on the machine specification described previously, and it is a PCIe 1.1 spec. As such, data transfers are bounded by the PCIe 1.1 bandwidth and slower than what is available in the market. Even so, the data provided demonstrates that queries utilizing CPUs are slower.