Programming GPUs for non-graphics workloads – from General Purpose GPU (GPGPU) to GPU compute

Tim Kaldewey
Research Staff Member
IBM TJ Watson Research Center
tkaldew@us.ibm.com
Disclaimer

The author's views expressed in this presentation do not necessarily reflect the views of IBM.

Acknowledgements

I would like to thank all my co-authors from IBM and my prior positions at Oracle and UCSC whose work I am showing in this presentation.

I would also like to thank Patrick Cozzi for inviting me to teach in his classes multiple years in a row and for letting me re-use his introductory material.
Agenda

Programming GPUs for non-graphics workloads

- GPGPU
  - A brief introduction
  - Index Search implemented using openGL and Cg

- Modern GPU computing with CUDA
  - A very brief intro to CUDA
  - Index Search in CUDA
  - Performance optimizations
  - A new GPU-optimal (index) search algorithm
GPU Programming pre-CUDA

- The graphics pipeline

- Vertex Processor - geometric transformations of vertices in 3D space
- Rasterizer - transforms geometric primitives (triangles) into pixels
- Fragment Processor – colors the pixels
- **Programmable** were only vertex and fragment processors
GPGPU Programming

- GPGPU(.org) started in 2002 by Mark Harris
- Using Graphics APIs to solve non-graphics tasks
  - E.g. OpenGL & Cg
- Required use of graphics APIs
  - OpenGL for data transfers
  - Cg to “program”
  - Operations:
    - geometric transformations using the vertex processor (scatter)
    - coloring using the fragment processor (gather)
  - Vertices are stored as float4 (x,y,z,w)
  - Textures = 2D arrays of float4 vectors (r,g,b,a)
  - Compute = drawing
GPGPU Programming

Steps for GPGPU compute:

1. Organize data in a screen size array
2. Set up a viewport with 1:1 pixel:texel ratio
3. Create and bind texture of the same size
4. Download input data into texture
5. Bind (load) fragment program (computational kernel)
6. Render a screen size quad to perform computation, i.e. run fragment program on each Pixel
7. Read back results
Let’s Pick a Simple, but Omnipresent Task … Search

• Why Search?

• Honestly, how many times a day do you visit:

  Google®  Yahoo!®
Let’s Pick a Simple, but Omnipresent Task … Search

• Why Search?

• Honestly, how many times a day do you visit:

  Google™  Yahoo!

• How do you search (millions of) documents efficiently?

• Use an inverted index

<table>
<thead>
<tr>
<th>Keyword</th>
<th>DocID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adam</td>
<td>1, 2, 3</td>
</tr>
<tr>
<td>Bethlehem</td>
<td>4, 5</td>
</tr>
<tr>
<td>Character</td>
<td>1, 2, 3, 301, 5790</td>
</tr>
<tr>
<td>Drachenflieger</td>
<td>301, 317, 5790</td>
</tr>
<tr>
<td>Eva</td>
<td>1, 2</td>
</tr>
<tr>
<td>Flughafenbahnhof</td>
<td>5790</td>
</tr>
<tr>
<td>Grabdenkmal</td>
<td>2, 5790</td>
</tr>
<tr>
<td>Haubentaucher</td>
<td>300, 5790</td>
</tr>
</tbody>
</table>

(sorted)
Searching an Index

- The task: search an inverted (document) index

<table>
<thead>
<tr>
<th>Keyword</th>
<th>DocID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adam</td>
<td>1,2,3</td>
</tr>
<tr>
<td>Bethlehem</td>
<td>4,5</td>
</tr>
<tr>
<td>Character</td>
<td>1,2,3,301,5790</td>
</tr>
<tr>
<td>Drachenflieger</td>
<td>301,317,5790</td>
</tr>
<tr>
<td>Eva</td>
<td>1,2</td>
</tr>
<tr>
<td>Flughafenbahnhof</td>
<td>5790</td>
</tr>
<tr>
<td>Grabdenkmal</td>
<td>2,5790</td>
</tr>
<tr>
<td>Haubentaucher</td>
<td>300,5790</td>
</tr>
</tbody>
</table>

Can be stored separately. Lookup by position.

16 characters max.
### Searching an Index

- The task: search an inverted (document) index

<table>
<thead>
<tr>
<th>Keyword</th>
<th>DocID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adam</td>
<td>1,2,3</td>
</tr>
<tr>
<td>Bethlehem</td>
<td>4,5</td>
</tr>
<tr>
<td>Character</td>
<td>1,2,3,301,5790</td>
</tr>
<tr>
<td>Drachenflieger</td>
<td>301,317,5790</td>
</tr>
<tr>
<td>Eva</td>
<td>1,2</td>
</tr>
<tr>
<td>Flughafenbahnhof</td>
<td>5790</td>
</tr>
<tr>
<td>Grabdenkmal</td>
<td>2,5790</td>
</tr>
<tr>
<td>Haubentaucher</td>
<td>300,5790</td>
</tr>
</tbody>
</table>

16 characters max.

- On the CPU we use a few library calls and we are done

```c
char searchkey[16] = "Flughafenbahnhof";
result = bsearch((void*)&searchkey,index,
                 numentries,sizeof(char)*16,
                 (int(*)(const void*,const void*)) strcmp);
```
GPGPU Search – Data Format

• Storing data
  – Obviously you want 1:1 pixel-to-texture element (texel) ratio
    *unless you would like to play Scrabble ;-)*
  – Ascii mapped to 0.0 to 255.0
  – 1 pixel stores 4 chars (better?)
  – Mark beginning of words with 0.1
  – Need to store pointer/position in document index ptr
  – Align word boundaries with pixel boundaries r
  – Null-terminated strings 0.0
GPGPU Search – Data Storage

• Store data in texture

```c
float* data = malloc(sizeof(float)*1200*1200*4);
...
data[pos++] = 0.1;
data[pos++] = *(float*)&docindex;
for (i=0;i<=strlen(currentString);i++) {
    data[pos++] = (float)currentString[i];
}
...
GLuintSubImage2D(GL_TEXTURE_RECTANGLE_ARB,
                 0,0,0, // detail level, x-, y- offset
                 1200, 1200, // size
                 GL_RGBA, // texture format
                 GL_FLOAT, // data format
                 data); // data pointer
```
GPGPU Search in Action

• Comparing search key with stored strings

• Simple test for equality
  – Compare floats directly
  – Color by color
GPGPU Search Code

float4 search(float2 coords: WPOS,
              uniform samplerRECT texCgFrag) : COLOR {
    float2 data_coords = coords;
    float2 searchkey_coords = float2(0.5, 0.5);
    float4 data = texRECT(texCgFrag, data_coords);
    float4 searchkey = texRECT(texCgFrag, searchkey_coords);
    float done = 0.0;
    if (data.r == 0.1) {
        if (done == 0.0) {
            if (data.b != searchkey.b) done = -1.0;
            if (data.b == searchkey.b)
                if (data.b == 0.0) done = 1.0;
        }
        if (done == 0.0) {
            if (data.a != searchkey.a) done = -1.0;
        ...
    }
GPGPU Search – Code Execution

• To execute the code:
  drawQuad(1200,1200);

• Result uses a magic number (not used for ASCII mapping) 0.9

• After completion Result is anywhere in the texture

• Copying whole texture back to main memory inefficient

• Reduction
GPGPU Search – Reduction

• To execute the code:
  
  ```
  drawQuad(1200,1200);
  ```

• Result uses a magic number (not used for ASCII mapping) 0.9

• After completion Result is anywhere in the texture

• Copying whole texture back to main memory inefficient

• Reduction:

\[ 
\begin{array}{ccc}
0.0 & 0.0 & 0.0 \\
\ldots & & \\
\end{array} 
\]

\[ 
\begin{array}{ccc}
0.0 & 0.0 & 0.9 \\
0.0 & 0.9 & \end{array} 
\]

\[ 
\begin{array}{ccc}
0.9 \\
\end{array} 
\]
GPGPU Search – Reduction

- Reduction means gathering “neighborhood” data

```c
float4 reduce (float2 coords: WPOS,
    uniform samplerRECT texCgFrag2) : COLOR {
    float2 topleft = ((coords-0.5)*2.0)+0.5;
    float4 val1 = texRECT(texCgFrag2, topleft);
    float4 val2 = texRECT(texCgFrag2, topleft+float2(1,0));
    float4 val3 = texRECT(texCgFrag2, topleft+float2(1,1));
    float4 val4 = texRECT(texCgFrag2, topleft+float2(0,1));
    float4 result = (0.0,0.0,0.0,0.0,0.0);
    if (val4.r == 0.9) result = val4;
    if (val3.r == 0.9) result = val3;
    if (val2.r == 0.9) result = val2;
    if (val1.r == 0.9) result = val1;
    return result;
}
```
GPGPU Search – Reduction

• Repeat until we end up with a single pixel
• Search result will be in top left pixel

\[ \text{numPasses} = (\text{int})(\log((\text{double})\text{width})/\log(2.0)) \]
\[ \text{for (i=0; i<\text{numPasses}; i++) { } } \]

\[ \text{outputWidth} = \text{outputWidth} / 2; \]
\[ \text{drawQuad(outputWidth, outputWidth); } \]

...
GPGPU Search - Performance

- 10k Berkeley DB index operations (insert delete), all require searching the index first, Test001.tcl

- Berkley DB uses B-trees, which needed to be flattened for the GPU

Time required for 10k insert/delete operations using a dual-core 2.2ghz AMD Opteron vs. an nVidia 7900GS with 7 vertex and 20 fragment processors.
GPGPU Search - Performance

- 10k Berkeley DB index operations (insert delete), all require searching the index first, Test001.tcl

- Berkley DB uses B-trees, which needed to be flattened for the GPU

Time required for 10k insert/delete operations using a dual-core 2.2ghz AMD Opteron vs. an nVidia 7900GS with 7 vertex and 20 fragment processors.
GPGPU Search – Where does time go?

Time required for 10k insert/delete operations using a dual-core 2.2ghz AMD Opteron vs. an nVidia 7900GS with 7 vertex and 20 fragment processors.

- **Data Transfer ~40%**
  - More efficient data mapping, e.g. 4 char = 1 float problematic?

- CUDA made GPGPU obsolete ...
Agenda

Programming GPUs for non-graphics workloads

- GPGPU
  - A brief introduction
  - Index Search implemented using OpenGL and Cg

- Modern GPU computing with CUDA
  - A very brief intro to CUDA
  - Index Search in CUDA
  - Performance optimizations
  - A new GPU-optimal (index) search algorithm
CUDA Key Concepts – Function Classifiers

- __global__
  - callable from host
  - must return void
- __device__
  - callable only from device
  - function inlined by default (newer CUDA versions)
- Global and device functions
  - No recursion (except Fermi)
  - No static variables
  - No malloc()
  - Careful with function calls through pointers (Fermi)
  - Cannot access host memory “directly”
CUDA Key Concepts – Memory address spaces

• Host (CPU) and Device (GPU) have separate (memory) address spaces
  • Data needs to be “transferred” to/from the GPU
  • Simplest way is to explicitly copy data to/from device memory
  • Data copy always initiated by host

```c
cudaMemcpy(void* dst,
            const void* src,
            size_t count,
            cudaMemcpyHostToDevice | cudaMemcpyDeviceToHost);
```

• Specify direction of data copy
  •ToDevice for input data
  •ToHost for results
• When calling __global__ function pass dst pointer
CUDA Key concepts – Vector types

- `char[1-4], uchar[1-4], short[1-4], ushort[1-4], int[1-4], uint[1-4], long[1-4], ulong[1-4], longlong[1-2], ulonglong[1-2]`

- `float[1-4], double[1-2]`

- `dim3`

- Available in host and device code

- **Construct with** `make_<type name>`
  
  - `int2 i2 = make_int2(1, 2);`
  - `float4 f4 = make_float4(1.0f, 2.0f, 3.0f, 4.0f);`

- **Access with** `.x, .y, .z, and .w`
  
  - `int x = i2.x;`
  - `int y = i2.y`

- **No .r, .g, .b, .a, etc. like OpenGL, Cg**
• **Calling GPU** (__global__) function requires to specify
  • grid dimensions – How many blocks of threads to launch
    • 1 block executes on 1 streaming multiprocessor to completion
  • block dimensions – How many threads are in a block
    • threads execute in groups of 32 (warps) in SIM[T/D] fashion
    • #threads > warp can be synchronized with __syncthreads()
CUDA Key Concepts – “Global” Variables

```c
__device__ int a_dev;
...
__shared__ int a_smem;
```

- **__device__** variables
  - stored in device memory
  - accessible from all blocks
- **__shared__** variables
  - stored in shared on-chip memory (space constraints?)
  - accessible only within a block
Index search on the CPU

On the CPU we use a few library calls and we are done

```c
char searchkey[16] = "Flughafenbahnhof";
result = bsearch((void*)searchkey, indexCPU, numentries, sizeof(char)*16,
                 (int(*)(const void*,const void*)) strcmp);
```
A Simple implementation of (index) search

<table>
<thead>
<tr>
<th>Keyword</th>
<th>char indexCPU[4711];</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adam</td>
<td>indexCPU[0]</td>
</tr>
<tr>
<td>Bethlehem</td>
<td>indexCPU[16]</td>
</tr>
<tr>
<td>Character</td>
<td>indexCPU[32]</td>
</tr>
<tr>
<td>Drachenflieger</td>
<td>...</td>
</tr>
<tr>
<td>Eva</td>
<td></td>
</tr>
<tr>
<td>Flughafenbahnhof</td>
<td></td>
</tr>
<tr>
<td>Grabdenkmal</td>
<td></td>
</tr>
<tr>
<td>Haubentaucher</td>
<td></td>
</tr>
</tbody>
</table>

16 characters max.

• On the CPU we use a few library calls and we are done

```c
char searchkey[16]= "Flughafenbahnhof";
result = bsearch((void*)searchkey,indexCPU,
                 numentries,sizeof(char)*16,
                 (int(*)(const void*,const void*)) strcmp);
```

• Can we just port a CPU implementation?
Index search on the CPU

• Get the data to the GPU

```c
char* indexGPU;
char* searchkeysGPU;
char* resultsGPU;

// copy the data
cudaMalloc((void**)&indexGPU, sizeof(char)*wordlength*entries);
cudaMemcpy(indexGPU, indexCPU, sizeof(char)*wordlength*entries, CudaMemcpyHostToDevice);

// copy the searchkey(s)
cudaMalloc((void**)&searchkeysGPU, ...)
cudaMemcpy(searchkeysGPU, searchkeysCPU, sizeof(char)*wordlength*numsearches, CudaMemcpyHostToDevice);

// make room for the results
cudaMalloc((void**)&resultsGPU, ...)
```
A Simple GPU implementation

• Get the data to the GPU

```c
char* indexGPU;
char* searchkeysGPU;
char* resultsGPU;
// copy the data
cudaMalloc((void**)&indexGPU, sizeof(char)*wordlength*entries);
cudaMemcpy(indexGPU, indexCPU, sizeof(char)*wordlength*entries,
    CudaMemcpyHostToDevice);
// copy the searchkey(s)
cudaMalloc((void**)&searchkeysGPU, ...)
cudaMemcpy(searchkeysGPU, searchkeysCPU,
    sizeof(char)*wordlength*numsearches,
    CudaMemcpyHostToDevice);
// make room for the results
cudaMalloc((void**)&resultsGPU, ...)
```

• Know your hardware (GTX 285, 30 SMs, 8 cores each, 240 cores)
  • Set up an execution configuration & call global function

```c
dim3 Dg = dim3(30,0,0);
dim3 Db = dim3(8,0,0);
searchGPU<<<Dg,Db>>>(indexGPU, entries...)
```
A Simple GPU implementation

• The GPU kernel

```c
__global__ void searchGPU(char* index, int entries, int wordlength,
                         char* search_keys, int* results) {
    char* res;
    // use block and thread numbers for indexing
    res = bsearch(&search_keys[((blockIdx.x*BLOCK_SIZE)+threadIdx.x) *wordlength],
                  index,
                  entries,
                  wordlength);
    // use block and thread numbers for indexing
    results[(blockIdx.x*BLOCK_SIZE)+threadIdx.x] = (res-data)/
                        MAX_WORD_LENGTH;
}
```
A Simple GPU implementation

- The GPU kernel

```c
__global__ void searchGPU(char* index, int entries, int wordlength, char* search_keys, int* results) {
    char* res;
    // use block and thread numbers for indexing
    res = bsearch(&search_keys[((blockIdx.x*BATCH_SIZE)+threadIdx.x)*wordlength],
                   index,
                   entries,
                   wordlength);
    // use block and thread numbers for indexing
    results[(blockIdx.x*BATCH_SIZE)+threadIdx.x] = (res-data)/MAX_WORD_LENGTH;
}
```

- There is no libc on the GPU =(  
- Just stick __device__ in front of the libc code?  
- “bsearch” is recursive, but there is no recursion on the GPU  
  ➔ Write a iterative one ...
A Simple GPU binary search

```c
__device__ char* bsearchGPU(char *key, char *base, int n, int size)
{
    char *mid_point;
    int cmp;

    while (n > 0) {
        mid_point = (char *)base + size * (n >> 1);
        if ((cmp = strcmpGPU(key, mid_point)) == 0)
            return (char *)mid_point;
        if (cmp > 0) {
            base = (char *)mid_point + size;
            n = (n - 1) >> 1;
        } // cmp < 0
        else n >>= 1;
    }
    return (char *)NULL;
}
```

• Still need strcmp
A Simple GPU binary search

```c
__device__ char* bsearchGPU(char *key, char *base, int n, int size){
    char *mid_point;
    int cmp;

    while (n > 0) {
        mid_point = (char *)base + size * (n >> 1);
        if ((cmp = strcmpGPU(key, mid_point)) == 0)
            return (char *)mid_point;
        if (cmp > 0) {
            base = (char *)mid_point + size;
            n = (n - 1) >> 1;
        } // cmp < 0
        else n >>= 1;
    }
    return (char *)NULL;
}
```

- Still need strcmp
- Again, stick `__device__` in front of the libc code

```c
__device__ int strcmpGPU(char* s1, char* s2){
    while (*s1 == *s2++)
        if (*s1++ == 0) return 0;
    return (*s1 - *(s2 - 1));
}
```
Binary Search on the GPU

- Searching a large data set (512MB) with 33 million (225) 16-character strings
Binary Search on the GPU – Why is it slow?

• Searching a large data set (512MB) with 33 million (225) 16-character strings

• It's slower than a CPU implementation for all data set sizes!
  – Let's try some optimizations ...
Search requires to compare

- Search naturally requires MANY comparisons

- The `strcmp()` library function:

```c
int strcmp(const char* s1, const char* s2) {
    while (*s1 == *s2++)
        if (*s1++ == 0) return 0;
    return (*s1 - *(s2 - 1));
}
```
Search requires to compare

- Search naturally requires MANY comparisons
- The `strcmp()` library function:

```c
int strcmp(const char* s1, const char* s2){
    while (*s1 == *s2++)
        if (*s1++ == 0)return 0;
    return (*s1 - *(s2 - 1));
}
```

- **Byte-wise** memory access is known to be slow
Optimizing compare operations

• How about vector string comparison, a la SSE?

• No Byte vectors on the GPU … but Integer vectors

![Diagram showing string comparison and memory representation using hex values for MSB and LSB.]
Optimizing compare operations

- How about vector string comparison, a la SSE?
- No Byte vectors on the GPU ... but Integer vectors

```
String
  a b c d
  61 62 63 64
  64 63 62 61

Memory (hex)

Integer (decimal) 1684234849

<

MSB LSB
d c b a
64 63 62 61
61 62 63 64

Store as char[4] MSB first

Load as int LSB first (little endian)

> 1633837924
```
Optimizing compare operations

- How about vector string comparison, a la SSE?
- No Byte vectors on the GPU … but Integer vectors

- Loading character strings as int changes endianness
- CPU has bswap, on the GPU we have to write it:

```c
#define BSWP( x ) ; \\
temp = ( x ) << 24 ; \\
temp = temp | ( ( ( x ) << 8 ) & 0x00FF0000 ) ; \\
temp = temp | ( ( unsigned ) ( x ) >> 8 ) & 0x0000FF00 ) ; \\
x = temp | ( ( unsigned ) ( x ) >> 24 ) ;
```
Optimizing compare operations

• Comparing integer vectors (bswap for <> skipped for clarity)

```c
__device__ int intcmp(uint4* a, uint4* b){
    int r =1;
    if ((*a).x < (*b).x)
        r=-1;
    else if ((*a).x == (*b).x) {
        if ((*a).y < (*b).y)
            r=-1;
        else if ((*a).y == (*b).y) {
            if ((*a).z < (*b).z)
                r=-1;
            else if ((*a).z == (*b).z) {
                if ((*a).w < (*b).w)
                    r=-1;
                else if ((*a).w == (*b).w)
                    r=0;
            }
        }
    }
    return r;
}
```

• Still dereferencing 16 memory pointers ...
Binary Search on the GPU – Why is it slow?

• Searching a large data set (512MB) with 33 million (225) 16-character strings

• With intcmp it's only marginally faster than a CPU implementation
• We still do pointer chasing, i.e. roundtrips to memory ...
Reducing global memory access

- Intcmp is memory latency sensitive

<table>
<thead>
<tr>
<th>Processor</th>
<th>L1 [cyc]</th>
<th>L2 [cyc]</th>
<th>L3 [cyc]</th>
<th>mem [cyc]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Core i7 2.6GHz</td>
<td>4</td>
<td>10</td>
<td>40</td>
<td>350</td>
</tr>
<tr>
<td>nVidia GT200b 1.5 GHz</td>
<td>4</td>
<td>n/a</td>
<td>n/a</td>
<td>500</td>
</tr>
</tbody>
</table>

- We can use shared memory like L1

x 16 for each comparison !!!
Reducing global memory access

- Intcmp is memory latency sensitive

```c
__shared__ uint4 cache[NUM_THREADS*2];
__device__ uint4* bsearchGPU( uint4 *key, uint4 *base,
                                 size_t nmemb, size_t size)
{
    uint4 *mid_point;
    int cmp;
    cache[threadIdx.x*2]= *key;

    while (nmemb > 0) {
        mid_point = (uint4 *)base + size * (nmemb >> 1);
        cache[threadIdx.x*2+1]= *mid_point;
        if ((cmp = intcmp(&cache[threadIdx.x*2],
                          &cache[threadIdx.x*2+1]))== 0)
            return (uint4 *)mid_point;
    }
}
```

- We can use shared memory like L1

<table>
<thead>
<tr>
<th>Processor</th>
<th>L1 [cyc]</th>
<th>L2 [cyc]</th>
<th>L3 [cyc]</th>
<th>mem [cyc]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Core i7 2.6GHz</td>
<td>4</td>
<td>10</td>
<td>40</td>
<td>350</td>
</tr>
<tr>
<td>nVidia GT200b 1.5 GHz</td>
<td>4</td>
<td>n/a</td>
<td>n/a</td>
<td>500</td>
</tr>
</tbody>
</table>

x 16 for each comparison !!!
Binary Search on the GPU – optimized

- Searching a large data set (512MB) with 33 million (225) 16-character strings

Is binary search optimal for a SIM[D/T] architecture?
GPU architecture reminder – SIMD/SIMT

• Inside Streaming Multitprocessor
  – Single Instruction Multiple Threads/Data (SIMT/SIMD)
  – All PEs in 1SM execute same instruction or no-op (SIMD threads)
  – Warps of 32 threads (or more to hide memory latency)
What happens during Multi-threaded Binary Search?

• Index: a sorted char array 32 entries
• 4 queries: t, 8, f, r
• 4 processor cores: P1–P4
• 1 processor core – 1 search: P0:t, P1:8, P2:f, P3:r
• Theoretical worst-case execution time: \( \log_2(32)=5 \)
What happens during Multi-threaded Binary Search?

- **Index**: a sorted char array 32 entries
- **4 queries**: t, 8, f, r
- **4 processor cores**: P1 – P4
- **1 processor core – 1 search**: P0: t, P1: 8, P2: f, P3: r
- Theoretical worst-case execution time: $\log_2(32)=5$

**Iter. 1)**

```
4 5 6 7 8 9 a b c d e f g h i j k l m n o p q r s t u v w x y z
```

- P0: t, P1: 8, P2: f, P3: r

**Iter. 2)**

```
4 5 6 7 8 9 a b c d e f g h i j k l m n o p q r s t u v w x y z
```

- P1: 8, P2: f
- P0: t, P3: r
What happens during Multi-threaded Binary Search?

Iter. 2)  
```
4 5 6 7 8 9 a b c d e f g h i j k l m n o p q r s t u v w x y z
```

P1: 8, P2: f

Iter. 3)  
```
4 5 6 7 8 9 a b c d e f g h i j
```

P1: 8, P2: f, P0: t, P3: r

Iter. 4)  
```
7 8 9 a b
```

P1: 8, P3: r, P0: t

Iter. 5)  
```
7 8 9
```

P1: 8, P0: t
Multi-threaded Binary Search - Analysis

- 100% utilization requires #cores concurrent queries
- Queries finishing early
  - utilization < 100%
- Memory access collisions
  - serialized memory access
- #memory accesses $\log_2(n)$
- More threads
  - more results
  - response time likely to be worst case: $\log_2(n)$

Can we improve the worst case?
Binary Search

- How Do you (efficiently) search an index?

  - Open phone book ~middle
  - 1st name = whom you are looking for?
  - < , > ?
  - Iterate

  - Each iteration: #entries/2 (n/2)
  - Total time: $\log_2(n)$
Parallel (Binary) Search

• What if you have some friends (3) to help you?

• Give each of them $\frac{1}{4}$.

• Divide et impera!

  – Each is using binary search takes $\log_2(n/4)$
  • All can work in parallel $\Rightarrow$ faster: $\log_2(n/4) < \log_2(n)$

* You probably want to tear it a little more intelligent than that, e.g. at the binding ;-)
Parallel (Binary) Search

• What if you have some friends (3) to help you?

• Divide et impera!
  – Each is using binary search takes $\log_2(n/4)$
  • All can work in parallel ➞ faster: $\log_2(n/4) < \log_2(n)$
  • 3 of you are wasting time!

• Give each of them $\frac{1}{4}$ *

* You probably want to tear it a little more intelligent than that, e.g. at the binding ;-)

[Image of a person flipping through a large book, with a picture of a torn yellow pages]

[Image of yellow pages torn in half]
P-ary Search

- Divide et impera !!

- How do we know who has the right piece ?
P-ary Search

• Divide et impera !!

• How do we know who has the right piece?

  • It's a sorted list:
    - Look at first and last entry of a subset
    - If first entry < searched name < last entry
      - Redistribute
      - Otherwise … throw it away
    - Iterate
P-ary Search

- What do we get?

  - Each iteration: \( n/4 \Rightarrow \log_4(n) \)
  - Assuming redistribution time is negligible:
    \( \log_4(n) < \log_2(n/4) < \log_2(n) \)
  - But each does 2 lookups!
  - How time consuming are lookup and redistribution?
P-ary Search

• What do we get?

  • Each iteration: n/4
    \[ \log_4(n) \]

  • Assuming redistribution time is negligible:
    \[ \log_4(n) < \log_2(n/4) < \log_2(n) \]

  • But each does 2 lookups!

  • How time consuming are lookup and redistribution?

\[ \begin{array}{cc}
\text{memory access} & \text{synchronization} \\
\end{array} \]
P-ary Search

• What do we get?

• Searching a database index can be implemented the same way
  – Friends = Processor cores (threads)
  – Without destroying anything ;-)
P-ary Search - Implementation

• Strongly relies on fast synchronization
  • friends = threads / vector elements

Iteration 1)

<p>| | | | | | | | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
<td>f</td>
<td>g</td>
<td>h</td>
<td>i</td>
<td>j</td>
<td>k</td>
<td>l</td>
<td>m</td>
<td>n</td>
<td>o</td>
</tr>
</tbody>
</table>

\[
\begin{array}{cccccc}
P_0 & : & g & P_1 & : & g & P_2 & : & g & P_3 & : & g
\end{array}
\]
P-ary Search - Implementation

- Strongly relies on fast synchronization
- \text{friends} = \text{threads} / \text{vector elements}

\begin{align*}
\text{Iteration 1)} & \\
\text{Iteration 2)} & \\
\text{Iteration 1)} &
\begin{align*}
\text{P}_0 & : g \\
\text{P}_1 & : g \\
\text{P}_2 & : g \\
\text{P}_3 & : g \\
\text{cdedefghijk} &
\end{align*}
\end{align*}

\begin{align*}
\text{Iteration 2)} & \\
\text{Iteration 2)} &
\begin{align*}
\text{P}_0 & \text{ P}_1 \text{ P}_2 \text{ P}_3 : g \\
\text{cdedefghijk} &
\end{align*}
\end{align*}
P-ary Search - Implementation

• Strongly relies on fast synchronization
  • friends = threads / vector elements

 iteration 1)

 Iteration 2)

• Synchronization ~ repartition cost
• pthreads ($$), cmpxchng($)
• SIMD SSE-vector, GPU threads via shared memory (~0)

• Implementation using a B-tree is similar and (obviously) faster
P-ary Search - Implementation

- B-trees group pivot elements into nodes

- Access to pivot elements is coalesced instead of a gather
- Nodes can also be mapped to
  - Cache Lines (CSB+ trees)
  - Vectors (SSE)
  - #Threads per block
P-ary Search on a sorted integer list – Implementation (1)

```c
__shared__ int offset;
__shared__ int cache[BLOCKSIZE+2]

__global__ void parySearchGPU(int* data, int length,
                                int* list_of_search_keys, int* results)

    int start, sk;
    int old_length = length;

    // initialize search range starting with the whole data set
    if (threadIdx.x == 0) {
        offset = 0;
        // cache search key and upper bound in shared memory
        cache[BLOCKSIZE] = 0x7FFFFFFF;
        cache[BLOCKSIZE+1] = list_of_search_keys[blockIdx.x];
        results[blockIdx.x] = -1;
    }

    __syncthreads();
    //
    sk = cache[BLOCKSIZE+1];
```
__shared__ int offset;
__shared__ int cache[BLOCKSIZE+2]

__global__ void parySearchGPU(int* data, int length,
                             int* list_of_search_keys, int* results)

int start, sk;
int old_length = length;
// initialize search range starting with the whole data set
if (threadIdx.x == 0) {
    offset = 0;
    // cache search key and upper bound in shared memory
    cache[BLOCKSIZE] = 0x7FFFFFFF;
    cache[BLOCKSIZE+1] = list_of_search_keys[blockIdx.x];
    results[blockIdx.x] = -1;
}
    syncthreads();
//
    sk = cache[BLOCKSIZE+1];

Why?
// repeat until the #keys in the search range < #threads
while (length > BLOCKSIZE){
    // calculate search range for this thread
    length = length/BLOCKSIZE;
    if (length * BLOCKSIZE < old_length) length += 1;
    old_length = length;
    // why don’t we just use floating point?
    start = offset + threadIdx.x * length;
    // cache the boundary keys
    cache[threadIdx.x] = data[start];
    __syncthreads();
    // if the searched key is within this thread's subset,
    // make it the one for the next iteration
    if (sk >= cache[threadIdx.x] && sk < cache[threadIdx.x+1]){
        offset = start;
    }
    __syncthreads();
    // all threads start next iteration with the new subset
}
// repeat until the #keys in the search range < #threads
while (length > BLOCKSIZE){
  // calculate search range for this thread
  length = length/BLOCKSIZE;
  if (length * BLOCKSIZE < old_length) length += 1;
  old_length = length;
  // why don’t we just use floating point?
  start = offset + threadIdx.x * length;
  // cache the boundary keys
  cache[threadIdx.x] = data[start];
  __syncthreads();
  // if the searched key is within this thread's subset,
  // make it the one for the next iteration
  if (sk >= cache[threadIdx.x] && sk < cache[threadIdx.x+1]){ 
    offset = start;
  }
  __syncthreads();
  // all threads start next iteration with the new subset
}
// last iteration
start = offset + threadIdx.x;
if (sk == data[start])
    results[blockIdx.x] = start;
P-ary Search – Analysis

- 100% processor utilization for each query
- Multiple threads can find a result
  - How does this impact correctness?

```
c d e f g h i j k
```

```
P0  P1  P2  P3: g
```
P-ary Search – Analysis

• 100% processor utilization for each query
• Multiple threads can find a result
  • How does this impact correctness?
• Convergence depends on #threads
  ▪ GTX285: 1 SM, 8 cores(threads) → p=8
• Better Response time
  • $\log_p(n)$ vs $\log_2(n)$

![Graph showing the comparison between p-ary search and binary search]
P-ary Search – Analysis

- 100% processor utilization for each query
- Multiple threads can find a result
  - Does not change correctness
- Convergence depends on #threads

GTX285: 1 SM, 8 cores(threads) → p=8

- Better Response time
  - $\log_p(n)$ vs $\log_2(n)$

- More memory access
  - $(p*2 \text{ per iteration}) \times \log_p(n)$
  - Caching
    - $(p-1) \times \log_p(n)$ vs. $\log_2(n)$
P-ary Search – Analysis

• 100% processor utilization for each query
• Multiple threads can find a result
  • Does not change correctness
• Convergence depends on #threads

GTX285: 1 SM, 8 cores(threads) → p=8

• Better Response time
  • \( \log_p(n) \) vs \( \log_2(n) \)

• More memory access
  • \( p \times 2 \) per iteration * \( \log_p(n) \)
  • Caching
    (\( p-1 \)) * \( \log_p(n) \) vs. \( \log_2(n) \)

• Lower Throughput
  • \( 1/\log_p(n) \) vs \( p/\log_2(n) \)
P-ary Search (GPU) – Throughput

• Superior throughput compared to conventional algorithms

Searching a 512MB data set with 134mill. 4-byte integer entries, Results for a nVidia GT200b, 1.5GHz, GDDR3 1.2GHz.
P-ary Search (GPU) – Response Time

- Response time is workload independent for B-tree implementation

Searching a 512MB data set with 134mill. 4-byte integer entries,
Results for a nVidia GT200b, 1.5GHz, GDDR3 1.2GHz.
P-ary Search (GPU) – Scalability

- GPU Implementation using SIMT (SIMD threads)
- Scalability with increasing #threads (P)

64K search queries against a 512MB data set with 134mill. 4-byte integer entries, Results for a nVidia GT200b, 1.5GHz, GDDR3 1.2GHz.
P-ary Search (GPU) – Scalability

- GPU Implementation using SIMT (SIMD threads)
- Scalability with increasing #threads (P)

![Graph showing scalability of P-ary search with increasing threads]

64K search queries against a 512MB data set with 134mill. 4-byte integer entries, Results for a nVidia GT200b, 1.5GHz, GDDR3 1.2GHz.
Questions?