# CS121 Parallel Computing
# Lab 2 – Cuckoo Hashing

A hash table is a data structure for maintaining a set of objects that allows fast insertion, deletion and lookup operations. There are a number of different methods for constructing hash tables, including chaining, linear and quadratic probing, double hashing, perfect hashing, etc. One widely used method in practice is *cuckoo hashing*, which guarantees $O(1)$ time lookup operations, in contrast to most other methods. This lab asks you to implement cuckoo hashing using CUDA. We begin with an overview of how cuckoo hashing works, then describe the requirements of the lab.

**Overview of Cuckoo Hashing**

Cuckoo hashing is named after a type of bird which lays its eggs in other birds' nests, and whose chicks push out other eggs from the nest after they hatch. Cuckoo hashing uses an array of size $n$, allowing up to $n$ items to be stored in the hash table. For simplicity, we only consider storing keys in the hash table and ignore any associated data.

We first describe the insert operation, which is the most complex. Each key $k$ can be stored into one of $t > 1$ locations in the table, where $t$ is generally a small number such as 2. These locations are given by $h_1(k), h_2(k), \ldots, h_t(k)$, where $h_1, h_2, \ldots, h_t$ are fixed hash functions. To insert $k$, we store it in location $h_1(k)$. If $h_1(k)$ was previously empty, then we are done. However, if there was a key $k_1$ previously stored in $h_1(k)$, then we need to *evict* $k_1$ and store it elsewhere in the table. In particular, suppose $h_1(k) = h_{i_1}(k_1)$; that is, location $h_1(k)$ is the $i_1$'th location where $k_1$ can be stored. Then we store $k_1$ in the next possible location $h_{i_1+1}(k_1)$; if $i_1 + 1 > t$, we wrap around and store $k_1$ at location $h_1(k_1)$. Then, similar to before, we check if the new location for $k_1$ was previously empty. If so, we are done. Otherwise, if the location was previously occupied by a key $k_2$, and if this location is the $i_2$'th possible location for $k_2$, i.e. $h_{i_2}(k_2)$, we evict $k_2$ and store it at its next possible location $h_{i_2+1}(k_2)$ (wrapping around if necessary). Storing $k_2$ may cause another eviction, requiring we repeat the previous procedure.

An insertion procedure either terminates with an insertion into an empty location, or leads to a long chain of evictions. Worse still, the chain can sometimes cycle back on itself and cause an infinite loop. In order to avoid overly long eviction chains, we place a bound $M = O(\log n)$ on the length of a chain. If an insertion causes more than $M$ evictions, we declare a *failure* using the current hash functions $h_1, \ldots, h_t$. We then restart the insertion process by picking a new set of hash functions of $h'_1, \ldots, h'_t$, and trying to insert *all* the keys using the new functions. That is, we rebuild the hash table using a new set of hash functions. We repeat this process until all the keys are successfully inserted using some set of hash functions. An illustration of the insertion process can be see at the Wikipedia page https://www.wikiwand.com/en/Cuckoo_hashing.

In contrast to insertions, looking up a key $k$ using cuckoo hashing is very simple. We simply check the locations $h_1(k), \ldots, h_t(k)$ to see if $k$ is stored in any of them. Note that this always takes a constant $O(t)$ number of steps. Similarly, to delete $k$ we check whether if it is stored at locations $h_1(k), \ldots, h_t(k)$, and set a location to empty if $k$ is found there.

**Implementing Cuckoo Hashing in CUDA**

For this lab, you will implement cuckoo hashing on the GPU using CUDA. First, design a highly parallel cuckoo hashing algorithm optimized for the GPU architecture. State clearly which hash functions you use. Your overall goal is to make the algorithm as fast as possible.

Next, use your algorithm to perform the following experiments. Perform each experiment first with two hash functions (i.e. $t = 2$), then again with three functions. When inserting $n$ items, use a bound of $4 \log n$ on the length of an eviction chain before restarting. Measure the amount of time each experiment takes, and report the speed of your algorithm in terms of millions of operations (insertions or lookups) per second. To obtain reliable results, perform each experiment 5 times.

1. Create a hash table of size $2^{25}$ in GPU global memory, where each table entry stores a 32-bit integer. Insert a set of $2^s$ random integer keys into the hash table, for $s = 10,11, \ldots 24$.
2. Insert a set $S$ of $2^{24}$ random keys into a hash table of size $2^{25}$, then perform lookups for the following sets of keys $S_0, \ldots, S_{10}$. Each set $S_i$ should contain $2^{24}$ keys, where $(100 - 10i)$ percent of the keys are randomly chosen from $S$, and the remainder are random 32-bit keys. For example, $S_0$ should contain only random keys from $S$, while $S_5$ should 50% random keys from $S$, and 50% completely random keys.
3. Fix a set of $n = 2^{24}$ random keys, and measure the time to insert the keys into hash tables of sizes $1.1n, 1.2n, \ldots, 2n$. Also, measure the insertion times for hash tables of sizes $1.01n, 1.02n$ and $1.05n$. Terminate the experiment if it takes too long and report the time used.
4. Using $n = 2^{24}$ random keys and a hash table of size $1.4n$, experiment with different bounds on the maximum length of an eviction chain before restarting. Which bound gives the best running time for constructing the hash table? Note however you are not required to find the optimal bound.

**Submission**

Prepare a report clearly describing your algorithm, your benchmarks, and the configuration of the machine you benchmarked on. Also describe any optimizations you performed or problems you encountered. Please submit your report and code (in an easily compilable form) to fanrui@shanghaitech.edu.cn, using the subject line "CS121 Lab 2", by the lab due date **11:59pm, June 9, 2019**.