Generating small random grids with a given number of occupied sites

István Borsos

Centre for Energy Research, EK-MFA, Budapest, Hungary

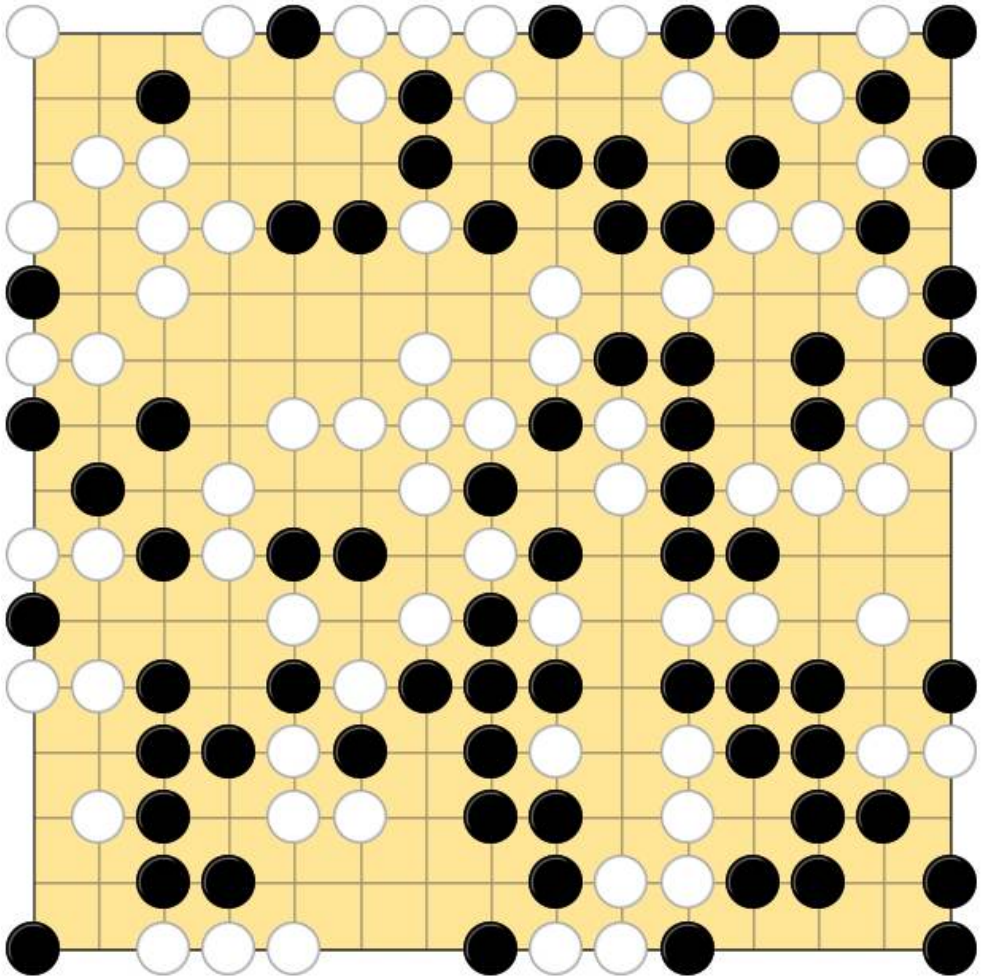
GPU Day 2023
May 15-16, 2023

K-subset problem

Generate random K-element subsets of a base set of N elements with equal probability

The base set and the subsets are represented as bit vectors

A playful application: random positions in board games



15x15 board, 225 sites
71 black pieces
70 white pieces

This position happens to be legal in Go, Gomoku or even in Hex (with rendering the hexagonal board on a square grid and assuming connections also along one of the diagonals).

Traditional sequential algorithms

Variation 1 List based

- Step 1. Build a list of indices of empty sites and set the subset empty
- Step 2. Pick an element randomly from the list, add it to the subset and delete it from the list, advance the counter
- Step 3. Repeat Step 2 until the subset has K elements

Variation 2 Bit probe based

- Step 1. Set the subset empty
- Step 2. Randomly probe a bit position in the subset
- Step 3. If it is set to 0 then set it to 1 and advance the counter of the elements, otherwise reject this probe
- Step 4. Repeat Step 2-3 until the subset has K elements

Weaknesses

- They are inherently slow, at least K steps are needed
- Each step consumes a random number.
- The list in Variation 1 needs a lot of memory
- Variation 2 suffers to find a new empty place by random probing in densely populated sets and consumes a random number for each probe including the rejected ones.
- Variation 2 executes a random number of steps which is disadvantageous for SIMD computers

Single core bit-parallel algorithm

My proposal is a fast algorithm for CPU that can be easily adapted to CUDA

- We have subset Sub of S<K elements (initially empty)
- We have a subset Candidates of elements that can be selected to augment Sub. (Initially the full base set)
- Step 1. [Select and Count] Select a random set of elements from Candidates and count the number of elements in it
- Step 2. [Include] If the selected set can be added to Sub without its new size exceeding K, then we add it to Sub and delete the selected set from the Candidates set
- Step 3. [Exclude] If the selected set is too large to be added to S, then we reduce the Candidates set to the selected set, that is we exclude the unselected Candidates elements from further selection.
- We repeat Step 1-3 until S=K is achieved.

Single core bit-parallel algorithm efficiency

- Because the size of the randomly choosen selected set is on the average half of the size of the candidate set, the candidate set is halved in Step 2 or Step 3, either because of the inclusion or the exclusion of half of its elements.
- So on the average the procedure stops in $log(N)$ steps
- Each step consumes one word of random bits for the selection, so a total of $log(N)$ words of random bits are consumed in the whole procedure

Hardware capabilities needed

- Mostly simple AND, OR, NOT instructions are used
- The only non-basic operation is **counting the ones** in a word
- However, today's CPUs and CUDA cards and the corresponding compilers all have fast implementation of **"population count"** for this purpose

CPU/CUDA implementation

```
uint32_t  subset, candidates, selected_candidates;  // bitsets
int nsub, nsel ;

subset=0;
candidates=~0;
nsub=0;

while (nsub != k) {
    selected_candidates=curand(&localState) & candidates;
    nsel= __popc(selected_candidates);
    if (nsub+nsel<=k){
            subset |= selected_candidates;
            nsub += nsel;
            candidates &= ~selected_candidates;}
    else
            candidates=selected_candidates;
    }
```

Executing the code results in "subset" containing "k" ones, where k is arbitrary in the range 0 to 32.

Straightforward CUDA implementation

The only CUDA specific details are

- calling of the random number generator
- calling of the population count

The rest is the same for a CPU code.

It also works for 64 bit sets, with elementary changes

Divergence

In the execution time, however, there is another factor to consider for CUDA: divergence

- Loop divergence: the algorithm executes a random number of steps of the loop depending on the random sequence, so some threads may take longer to finish, the slowest thread in the warp determines the warp's execution time
- Conditional (IF) divergence: some of the threads execute the IF branch, the others the ELSE branch, but this cannot happen simultaneously

CUDA speed for single word sets

Tests run on a decent mid-range GPU: RTX 3060 Ti

Generation speed: **28.5 billion subsets/second**

Multithread execution speed compared to single thread execution:

- With only one thread per warp working (thus no divergence), execution time is measured and the speed is scaled up to 32 threads:
- Hypothetical generation speed (as if there were no divergence): 58.5 billion subsets/second
- Thus the multithread slowdown factor, caused mostly by divergence, is 2.05

Scaling for other CUDA GPUs

As the program in the critical parts uses almost exclusively registers, the **speed is expected to scale according to the raw computing speed** of the various cards

From single word to larger grids

Storage: how to store the grids?

- Local (per thread) memory: slow
- Shared memory: slower than registers and limited in size (register storage is 4 times larger)
- Registers: no regular use of arrays, as there is no hardware indexing into the registers the array's size and the access pattern must be known at compile time and accepted by the compiler, thus e.g grid size should be known at compile time (otherwise the compiler places the arrays in slow local memory)

All considered, register storage is the only viable solution for fast implementation

Threadwise

Threadwise implementation guidelines:

- Each thread executes the above algorithm
- Each thread works on a different problem
- Replace all sets with arrays of sets
- Change all operations where sets are used to constant step "for" loops operating on the rows of the arrays
- Sum the count of elements over all rows, as the full count of the ones in the subset array should be used for the decisions

Threadwise CUDA code

```
for(int i=0; i<sizeY;i++){
  subset[i]=0;
  candidates[i]=~0;}

  nsub=0;
  while (nsub != k)  {
    nsel=0;
    for(int i=0; i<sizeY;i++){
      selected_candidates[i]=curand(&localState) & candidates[i];
      nsel +=__popc(selected_candidates[i]);}
    if (nsub+nsel<=k){
      for(int i=0; i<sizeY;i++){
         subset[i] |= selected_candidates[i];
         candidates[i] &= ~selected_candidates[i];}
         nsub += nsel;}
    else
      for(int i=0; i<sizeY;i++){
        candidates[i]=selected_candidates[i];}
}
```

Speed of Threadwise for 32x32 grids

Tested again on RTX 3060 Ti

Generation speed: **1.05 billion grids/second**

All grids have K bits set to 1 in the whole grid.
(K arbitrary from 0 to 1024)

Storage problem

For the 32x32 case:

- We have three arrays, each needs 32 registers (of 32 bits), thus **96 registers** in all
- With the rest of the variables and CUDA overhead the above code requires **120 registers** for a grid of 32 by 32 (according to PTX).
- Threads are limited to use 255 registers in CUDA, thus it is feasible for 32x32 grids, though not comfortable, as the subsequent processing may need registers more than the remaining.
- **But for 64x64 grids it does not work at all**, not enough registers.

Is there a way to reduce this problem?

Warpwise implementation

- The threads in a warp work on the same problem
- Different rows of the grids are stored in different threads
- Essentially the temporally executed FOR loops in the Threadwise implementation are executed spatially in the threads of the warp.

As a bonus, we get rid of the divergence problems

- As there is no loop, the loop divergence does not exist
- Even in the if-else conditional only one of the branches is executed in any warp, because the **decision is based on the add-reduction of the whole warp** and the variables in the decision have the same value in all threads. So the threads continue to execute in lockstep.
- So no divergence problem within a warp
- In the Threadwise implementation there is divergence, but apart from the efficiency loss, it does not cause synchronization problems, because the threads do not interact.

Warpwise implementation, the code

```
//----------------------------------------
// Assume the function "warpsum" is defined earlier in the code
//----------------------------------------


subset=0;
nsub=0;
candidates=~0;

while (nsub != k)
{
    selected_candidates=curand(&localState) & candidates;
    ntotalsel=warpsum(__popc(selected_candidates));
    if (nsub+ntotalsel<=k)
    {
        subset |= selected_candidates;
        nsub += ntotalsel;
        candidates &= ~selected_candidates;
    }
    else
        candidates=selected_candidates;
}
```

Summing warp registers

Add-Reduction is a standard procedure in CUDA

Usually through shared and global memory, but that is what we want to avoid.

Warp shuffles help us solve this.

The function "**__shfl_xor_sync**", which is very efficiently implemented in CUDA cards, is described in detail in various CUDA programming materials on warp shuffles introduced with Kepler and modified with Volta.

With it we can build an efficient add-reduction within the warp that sums the same register variable in all the threads of the warp in such a way (butterfly operation) that all the threads get the sum, exactly what we need in our program.

Warp sum

```
#define sizeofwarp 32

__inline__ __device__
int warpsum(int sum) {
#pragma unroll
for (int dist = sizeofwarp/2; dist > 0; dist /= 2)
    sum += __shfl_xor_sync(0xFFFFFFFF,sum, dist);
return sum;}
```

Caveat!
Be sure to use a compile time constant for the size of warp.
The CUDA constant **warpSize** is not available for the compiler at compile time and the compiler cannot unroll the FOR loop, even if the pragma is there, resulting in inefficient loop code.
This slowed down our program by a factor of almost 2!

Speed of Warpwise

Tests again on the same RTX 3060 Ti

32x32 grids were generated.

Generation speed: **650 million grids /second**

Comparison

For 32x32 grids:

Generation speed

- Threadwise implementation is faster (1.05 billion grids/s)
- Warpwise somewhat slower (0.65 billion grids/s)

We see that the extra time to do the warp sum is mostly compensated by the fact that Warpwise is free of the divergence losses.

Register usage (including all variables and CUDA overhead):

- Threadwise 120 registers
- Warpwise 28 registers
- Warpwise easily accomodates larger grids as the basic vesion stores only one row at a time.
- For larger grids: a 64x64 grid can be handled by processing two folded rows instead of one unfolded row and simple modification of the code. Thus each thread will process only four 32 bit word in Warpwise.

Conclusion

Warpwise implementation is a viable option to mitigate the register storage bottleneck while maintaining high speed generation without using shared memory.

Thank you for your attention.