1 Benchmarking Program

This section describes the usage of the benchmarking program and its source code. XXX what does the benchmark program do? The structure of the source file is listed below.

1a ⟨*bench.cpp* 1a⟩≡

```
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <ctime>
#include <cstdint>
#include <sycl/sycl.hpp>
#include <omp.h>
⟨Additional headers 12a⟩
⟨Macro definitions 2a⟩
⟨Type definitions 1b⟩
⟨Function prototypes 2d⟩
⟨Global variables 2b⟩
⟨Function definitions 3d⟩
int main(int argc, const char **argv)
{
        ⟨Process arguments 2c⟩
        ⟨Possibly initialize dump 5c⟩
        ⟨Time the benchmark 4b⟩
        ⟨Print the output 5a⟩
        ⟨Possibly finalize dump 5d⟩
        return 0;
}
```

Defines:
`argc`, used in chunk 2c.
`argv`, used in chunks 2–4.
`main`, never used.

The program supports the following hash algorithms and methods to generate the hashes.

1b ⟨*Type definitions* 1b⟩≡ (1a)

```
enum algorithm {SHA224, SHA256, BLAKE3};
enum method {SERIAL, OPENMP, SYCL_CPU, SYCL_GPU};
```

Uses `algorithm` 2b and `method` 2b.

The following `printf` template is used for printing output to the standard output.

2a ⟨*Macro definitions* 2a⟩≡ (1a) 3b ▷

```
#define OUTPUT_TEMPLATE "hashes_per_block =\t%u\t" \
        "num_blocks =\t%u\t" \
        "algorithm =\t%s\t" \
        "runner =\t%s\t" \
        "elapsed (s) =\t%f\n"
```

Uses `algorithm` 2b, `elapsed` 4b, `hashes_per_block` 2b, and `num_blocks` 2b.

1.1 Argument Processing

This subsection details the code for processing `argv`. The arguments listed above are stored in the following global variables, respectively. Algorithms and methods are stored using enumerations defined earlier.

2b ⟨*Global variables* 2b⟩≡ (1a) 3a ▷

```
static unsigned hashes_per_block;
static unsigned num_blocks;
static enum algorithm algorithm;
static enum method method;
```

Defines:
- `algorithm`, used in chunks 1–3, 5–8, and 11–13.
- `hashes_per_block`, used in chunks 2a, 3c, 5a, 7, 8, and 11a.
- `method`, used in chunks 1b, 3c, 5a, and 7a.
- `num_blocks`, used in chunks 2a, 3c, 5a, 8, and 11a.

If the incorrect number of arguments were given, then we print an error message and exit. Currently, we are assuming that the user will have access to the documentation and will be able to find the usage information.

2c ⟨*Process arguments* 2c⟩≡ (1a) 3c ▷

```
if (argc != 5) {
        fprintf(stderr, "%s: incorrect number of arguments.\n", argv[0]);
        return 1;
}
```

Uses `argc` 1a and `argv` 1a.

Processing each argument takes several function calls. Also, if there is an error processing the arguments, we want to exit with an error. The below functions will handle these tasks. They both take `argv` and the argument within `argv` to parse. The `parse_enumerator` function also takes an array of valid enumerators and the number of valid enumerators; the parsed value is the index of the enumerator within the array. Both functions return the parsed value.

2d ⟨*Function prototypes* 2d⟩≡ (1a) 6b ▷

```
static unsigned parse_unsigned(const char **argv, int arg_num);
static int parse_enumerator(const char **argv, int arg_num,
                const char **enumerators, const unsigned num_enumerators);
```

Uses `argv` 1a, `parse_enumerator` 4a, and `parse_unsigned` 3d.

The algorithm and method enumerations both need an array of enumerators for `parse_enumerator`. The order of strings within the array `must` match the order of enumerators within the enumeration definition.

3a ⟨*Global variables* 2b⟩+≡ (1a) ◁2b 5b▷

```
static const char *algorithms[] = {"sha224", "sha256", "blake3"};
static const char *methods[] = {"serial", "openmp", "sycl-cpu", "sycl-gpu"};
```

It will be useful to have a macro to take the length of an array.

3b ⟨*Macro definitions* 2a⟩+≡ (1a) ◁2a

```
#define LENGTH(arr) (sizeof(arr) / sizeof((arr)[0]))
```

C++ wants additional casts for enumerators.

3c ⟨*Process arguments* 2c⟩+≡ (1a) ◁2c

```
hashes_per_block = parse_unsigned(argv, 1);
num_blocks = parse_unsigned(argv, 2);
algorithm = (enum algorithm)
        parse_enumerator(argv, 3, algorithms, LENGTH(algorithms));
method = (enum method)parse_enumerator(argv, 4, methods, LENGTH(algorithms));
```

Uses `algorithm` 2b, `argv` 1a, `hashes_per_block` 2b, `method` 2b, `num_blocks` 2b, `parse_enumerator` 4a, and `parse_unsigned` 3d.

Parsing integers is simple using C's `sscanf`. If it does not match any inputs items, which is the only case for failure here, it returns `EOF`. It cannot fail before the first match, because there is only one item to match.

3d ⟨*Function definitions* 3d⟩≡ (1a) 4a▷

```
static unsigned parse_unsigned(const char **argv, int arg_num)
{
        int rc;
        unsigned result;
        rc = sscanf(argv[arg_num], "%u", &result);
        if (rc == EOF) {
                fprintf(stderr, "%s: could not parse \%s" as an unsigned "
                        "integer\n", argv[0], argv[arg_num]);
                exit(1);
        }
        return result;
}
```

Defines:
`parse_unsigned`, used in chunks 2d and 3c.
Uses `argv` 1a.

Enumerators are parsed using a linear scan and C's `strcmp`. There are ways to make this more efficient, but it probably does not matter. Printing the list of enuemrators as part of the error message is non-trivial, and possibly unnecessary.

4a ⟨*Function definitions* 3d⟩+≡ (1a) ◁3d 5e▷

```
static int parse_enumerator(const char **argv, int arg_num,
                const char **enumerators, const unsigned num_enumerators)
{
        unsigned i;
        for (i = 0; i < num_enumerators; i++)
                if (strcmp(argv[arg_num], enumerators[i]) == 0)
                        return i;
        fprintf(stderr, "%s: could not match \%s" to ",
                        argv[0], argv[arg_num]);
        for (i = 0; i < num_enumerators - 1; i++)
                fprintf(stderr, "\%s", ", enumerators[i]);
        fprintf(stderr, "or \%s".\n", enumerators[i]);
        exit(1);
        return 0;
}
```

Defines:
`parse_enumerator`, used in chunks 2d and 3c.
Uses `argv` 1a.

1.2 Timing the Benchmark

The POSIX clock interface is used to get the time before and after the hashes are generated. Since only a relative time is required, `CLOCK_MONOTONIC` is sufficient. The elapsed time is stored in the variable `elapsed`.

4b ⟨*Time the benchmark* 4b⟩≡ (1a)

```
double elapsed;
struct timespec start, end;
clock_gettime(CLOCK_MONOTONIC, &start);
⟨Run the benchmark 7a⟩
clock_gettime(CLOCK_MONOTONIC, &end);
elapsed = (double)end.tv_sec - (double)start.tv_sec;
elapsed += ((double)end.tv_nsec - (double)start.tv_nsec) / 1e12L;
```

Defines:
`elapsed`, used in chunks 2a and 5a.

1.3 Printing the Output

All of the variables that are needed for output have been defined. The string names for the algorithm and method are printed instead of their enumerator's number.

5a ⟨*Print the output* 5a⟩≡ (1a)

```
printf(OUTPUT_TEMPLATE, hashes_per_block, num_blocks,
                algorithms[algorithm], methods[method], elapsed);
```

Uses `algorithm` 2b, `elapsed` 4b, `hashes_per_block` 2b, `method` 2b, and `num_blocks` 2b.

1.4 Dumping Hashes

If the `DUMP` preprocessor macro is defined, the program should dump its hashes to a file. This is for validating that the generated hashes are correct.

See the check-dumps.sh script to compare the hashes side-by-side.

5b ⟨*Global variables* 2b⟩+≡ (1a) ◁3a 7d▷

```
#ifdef DUMP
static FILE *dump_stream;
#endif
```

5c ⟨*Possibly initialize dump* 5c⟩≡ (1a)

```
#ifdef DUMP
dump_stream = fopen("bench-hashes.dat", "w");
#endif
```

5d ⟨*Possibly finalize dump* 5d⟩≡ (1a)

```
#ifdef DUMP
fclose(dump_stream);
#endif
```

All of the generators will call `dump`. When dumping is enabled, it will be an actual function. Otherwise, it will be a macro that expands to nothing.

5e ⟨*Function definitions* 3d⟩+≡ (1a) ◁4a 6c▷

```
#ifdef DUMP
static void dump(unsigned char *buffer, size_t num_hashes, size_t hash_size)
{
        ⟨Dump hashes in buffer 6a⟩
}
#else
#define dump(x, y, z) /* dump */
#endif
```

Defines:
`dump`, used in chunks 8 and 11a.

Each hash is printed in hexadecimal, on its own line. TODO: it might be useful to print the index along with the hash.

6a ⟨*Dump hashes in* `buffer` 6a⟩≡ (5e)

```
unsigned i, j;
for (i = 0; i < num_hashes; i++) {
        for (j = 0; j < hash_size; j++) {
                fprintf(dump_stream, "%02x", buffer[i * hash_size + j]);
        }
        fprintf(dump_stream, "\n");
}
```

1.5 Supporting Different Hash Algorithms

The goal is to support several different hash algorithms and compare their performance. The following function will be used to dispatch the correct hash function according the algorithm, and ensure the result is written to `buf[slot]`. The algorithm must be passed explicitly, otherwise the SYCL compiler will complain.

6b ⟨*Function prototypes* 2d⟩+≡ (1a) ◁2d

```
static void run_hash(enum algorithm algorithm, uint64_t input,
                unsigned char *buf, unsigned slot);
```

Uses `algorithm` 2b and `run_hash` 6c.

The downside of this approach is that the algorithm is checked every time a hash is generated. This could be avoided by calling a function that is set to the appropriate hash function, but SYCL does not support calling function pointers in its kernels.

6c ⟨*Function definitions* 3d⟩+≡ (1a) ◁5e 9▷

```
static void run_hash(enum algorithm algorithm, uint64_t input,
                unsigned char *buf, unsigned slot)
{
        switch (algorithm) {
        case SHA224:
                ⟨Hash input to buf[slot] with SHA-224 12b⟩
                break;
        case SHA256:
                ⟨Hash input to buf[slot] with SHA-256 12e⟩
                break;
        case BLAKE3:
                ⟨Hash input to buf[slot] with BLAKE3 13b⟩
                break;
        }
}
```

Defines:
`run_hash`, used in chunks 6b, 8, and 11a.
Uses `algorithm` 2b.

1.6 Supporting Different Running Methods

The other feature of this program is that it supports several drivers to run the has generation code. Some runners may require common, local variables.

7a ⟨*Run the benchmark* 7a⟩≡ (4b)

```
⟨Local declarations for the runner 7b⟩
switch (method) {
case SERIAL:
        ⟨Run benchmark in serial 8a⟩
        break;
case SYCL_CPU:
        ⟨Run benchmark with SYCL on the CPU 11b⟩
        break;
case SYCL_GPU:
        ⟨Run benchmark with SYCL on the GPU 11c⟩
        break;
case OPENMP:
        ⟨Run benchmark with OpenMP 8b⟩
        break;
}
⟨Delete any local declarations 7c⟩
```

Uses `method` 2b.

Something that all of the runners will need is a buffer to write the hashes to. This requires knowing the size of each hash in bytes. They are stored in the `digest_size` array.

7b ⟨*Local declarations for the runner* 7b⟩≡ (7a)

```
unsigned char *output_buffer =
                new unsigned char[hashes_per_block * digest_size[algorithm]];
```

Uses `algorithm` 2b and `hashes_per_block` 2b.

7c ⟨*Delete any local declarations* 7c⟩≡ (7a)

```
delete[] output_buffer;
```

7d ⟨*Global variables* 2b⟩+≡ (1a) ◁5b

```
static const unsigned digest_size[] = {28u, 32u, 32u};
```

1.7 Running in Serial

Running in serial is simple.

8a ⟨*Run benchmark in serial* 8a⟩≡ (7a)

```
for (uint64_t i = 0; i < num_blocks; i++) {
        for (uint64_t j = 0; j < hashes_per_block; j++) {
                run_hash(algorithm, i * hashes_per_block + j, output_buffer, j);
        }
        dump(output_buffer, hashes_per_block, digest_size[algorithm]);
}
```

Uses algorithm 2b, dump 5e, hashes_per_block 2b, num_blocks 2b, and run_hash 6c.

1.8 Running with OpenMP

The same as serial, but with an OpenMP pragma.

There are probably other ways to parallelize this.

8b ⟨*Run benchmark with OpenMP* 8b⟩≡ (7a)

```
#pragma omp parallel
for (uint64_t i = 0; i < num_blocks; i++) {
#pragma omp for
        for (uint64_t j = 0; j < hashes_per_block; j++) {
                run_hash(algorithm, i * hashes_per_block + j, output_buffer, j);
        }
        dump(output_buffer, hashes_per_block, digest_size[algorithm]);
}
```

Uses algorithm 2b, dump 5e, hashes_per_block 2b, num_blocks 2b, and run_hash 6c.

1.9 Running with SYCL

The following function looks for devices that match the selector, and returns queues for them. If `use_all` is false, then only on queue is returned.

9 ⟨*Function definitions* 3d⟩+≡ (1a) ◁6c 10▷

```
template<class Selector>
static std::vector<sycl::queue> make_queues(Selector sel, bool use_all)
{
        sycl::platform p(sel);
        std::vector<sycl::device> ds = p.get_devices();
        if (!use_all) {
                for (unsigned i = 1; i < ds.size(); i++) {
                        ds.pop_back();
                }
        }
        std::vector<sycl::queue> result;
        for (sycl::device d : ds) {
                sycl::queue q(d);
                result.push_back(q);
        }
        return result;
}
```

Defines:
`make_queues`, used in chunk 11a.

The following function creates buffers for each device in the given vector. If allocation fails, the program exits with an error.

TODO: better error message

10 ⟨*Function definitions* 3d⟩+≡ (1a) ◁9 11a▷

```
static std::vector<unsigned char *> alloc_buffers(std::vector<sycl::queue> qs,
                int buffer_size)
{
        std::vector<unsigned char *> result;
        for (sycl::queue q : qs) {
                unsigned char *b = sycl::malloc_device<unsigned char>(
                                buffer_size, q);
                if (b == nullptr) {
                        fprintf(stderr, "sycl::malloc_device failed when "
                                        "called %u bytes were requested.\n",
                                        buffer_size);
                        exit(1);
                }
                result.push_back(b);
        }
        return result;
}
```

Defines:
`alloc_buffers`, used in chunk 11a.

Running with SYCL requires a few local variables and polymorphic types. A separate function is used to deal with this.

TODO: document use_all.

11a ⟨*Function definitions* 3d⟩+≡ (1a) ◁10

```
template <class Selector>
static void run_sycl(Selector sel, unsigned char *host_buffer, bool use_all)
{
        std::vector<sycl::queue> qs = make_queues(sel, use_all);
        int buffer_size = (hashes_per_block * digest_size[algorithm])
                        / qs.size();
        int hashes_per_device = hashes_per_block / qs.size();
        std::vector<unsigned char *> buffers = alloc_buffers(qs, buffer_size);
#pragma omp parallel if(qs.size() > 1) num_threads(qs.size())
        {
                int t = omp_get_thread_num();
                enum algorithm alg = algorithm;
                unsigned char *host_ptr = host_buffer + buffer_size * t;
                for (uint64_t i = 0, base = 0; i < num_blocks;
                                        i++, base += hashes_per_device) {
                        sycl::event hashes_ev = qs[t].parallel_for(sycl::range<1>(hashe
                                run_hash(alg, base + idx, buffers[t], idx);
                        });
                        sycl::event copy_ev = qs[t].memcpy(host_ptr, buffers[t], buffe
                        copy_ev.wait();
                        dump(host_ptr, hashes_per_device, digest_size[algorithm]);
                }
        }
        for (unsigned i = 0; i < buffers.size(); i++) {
                sycl::free(buffers[i], qs[i]);
        }
}
```

Defines:
run_sycl, used in chunk 11.
Uses algorithm 2b, alloc_buffers 10, dump 5e, hashes_per_block 2b, make_queues 9, num_blocks 2b, and run_hash 6c.

TODO runners that set use_all to false.

11b ⟨*Run benchmark with SYCL on the CPU* 11b⟩≡ (7a)

```
run_sycl(sycl::cpu_selector_v, output_buffer, true);
```

Uses run_sycl 11a.

11c ⟨*Run benchmark with SYCL on the GPU* 11c⟩≡ (7a)

```
run_sycl(sycl::gpu_selector_v, output_buffer, true);
```

Uses run_sycl 11a.

1.10 SHA-224 Hash Algorithm

The SHA-224 algorithm used here comes from an implementation that I found online.[1] There are some optimizations that could be made with a custom implementation, because we know exactly how long the message is, etc.

12a ⟨*Additional headers* 12a⟩≡ (1a) 12c▷

```
#include "sha224.hpp"
```

Using this implementation requires only the following function calls. The hash algorithms are implemented in `switch` case bodies, so a new scope is needed to declare local variables well.

12b ⟨*Hash* `input` *to* `buf[slot]` *with SHA-224* 12b⟩≡ (6c)

```
{
        class SHA224 ctx;
        ctx.init();
        ctx.update((const unsigned char *)&input, sizeof(input));
        ctx.final(buf + slot * digest_size[algorithm]);
}
```

Uses `algorithm` 2b.

As it turns out `SYCL_EXTERNAL` is not universal, and everything `must` be in the same translation to be portable across SYCL compilers, it seems. Although the below strategy works, it's bad practice and should be replaced.

12c ⟨*Additional headers* 12a⟩+≡ (1a) ◁12a 12d▷

```
#include "sha224.cpp"
```

1.11 SHA-256 Hash Algorithm

I modified the SHA-224 implementation to be SHA-256. I have not completely verified that it is correct.

12d ⟨*Additional headers* 12a⟩+≡ (1a) ◁12c 12f▷

```
#include "sha256.hpp"
```

12e ⟨*Hash* `input` *to* `buf[slot]` *with SHA-256* 12e⟩≡ (6c)

```
{
        class SHA256 ctx;
        ctx.init();
        ctx.update((const unsigned char *)&input, sizeof(input));
        ctx.final(buf + slot * digest_size[algorithm]);
}
```

Uses `algorithm` 2b.

See the SHA-224 section for an explanation.

12f ⟨*Additional headers* 12a⟩+≡ (1a) ◁12d 13a▷

```
#include "sha256.cpp"
```

[1] `http://www.zedwood.com/article/cpp-sha224-function`

1.12 BLAKE3 Hash Algorithm

The BLAKE3 implementation comes from the BLAKE3 reference implemenation. It had to be modified slightly to work with C++ and SYCL.

13a ⟨*Additional headers* 12a⟩+≡ (1a) ◁12f 13c▷
```
#include "blake3.h"
```

The interface for BLAKE3 uses different names and arguments than SHA-2, but functionally the same otherwise.

13b ⟨*Hash* `input` *to* `buf[slot]` *with BLAKE3* 13b⟩≡ (6c)
```
{
        blake3_hasher hasher;
        blake3_hasher_init(&hasher);
        blake3_hasher_update(&hasher, &input, sizeof(input));
        blake3_hasher_finalize(&hasher,
                        buf + slot * digest_size[algorithm],
                        digest_size[algorithm]);
}
```
Uses `algorithm` 2b.

See the SHA-224 section for an explanation.

13c ⟨*Additional headers* 12a⟩+≡ (1a) ◁13a
```
#include "blake3.cpp"
#include "blake3_dispatch.cpp"
#include "blake3_portable.cpp"
```

1.13 Support Scripts

The run-bench.sh script automates the collection of data using the benchmark, and outputs the results to the file bench-results. It generates SHA-256 and BLAKE3 hashes on the CPU and GPU with block sizes and hash counts read from the standard input. The generate-inputs.sh script, described later, generates suitable inputs.

It fails for the huge buffer sizes. This should be investigated, but it does not appear to affect the completion of the script or the results.

13d ⟨*run-bench.sh* 13d⟩≡
```
#!/bin/sh

while read -r x y
do
        ./bench $x $y sha256 openmp
        ./bench $x $y sha256 sycl-cpu
        ./bench $x $y sha256 sycl-gpu
        ./bench $x $y blake3 openmp
        ./bench $x $y blake3 sycl-cpu
        ./bench $x $y blake3 sycl-gpu
done | tee bench-results
```

1.14 Making Graphs

The following script can be used to convert the output of the benchmark script to a grap(1) graph, suitable to `copy` in report.ms. See the report source of that file to see how to use the output of this script.

This script needs some work, especially with selecting ticks.

Assumes 256-bit hashes.

14 ⟨*to-grap.sh* 14⟩≡

```
#!/bin/sh
echo ’label left "Real Time" "(s)"’
echo ’label bot "Block Size"’
awk ’-F\t’ ’
        function tosize(n, i, suffix, suffixes) {
                ⟨Convert bytes to human-readable units 15a⟩
        }
        BEGIN {
                xmin = 1e12
        }
        /openmp/ && /sha/ {char[count] = "ci"}
        /cpu/ && /sha/ {char[count] = "sq"}
        /gpu/ && /sha/ {char[count] = "pl"}
        /openmp/ && /blake/ {char[count] = "bu"}
        /cpu/ && /blake/ {char[count] = "*D"}
        /gpu/ && /blake/ {char[count] = "mu"}
        {
                x[count] = $2
                y[count] = $10
                ymax = $10 > ymax ? $10 : ymax
                xmax = $2 > xmax ? $2 : xmax
                xmin = $2 < xmin ? $2 : xmin
                xtick[$2]++
                count++
        }
        END {
                printf "coord y 0, %d log x\n", ymax + 1
                ⟨Select and print ticks 15b⟩
                printf "\"\\(sq SHA-256, CPU\" \"\\(pl SHA-256, GPU\" " \
                        "\"\\(*D BLAKE3, CPU\" \"\\(mu BLAKE3, GPU\" " \
                        "\"\\(ci SHA256, OMP\" \"\\(bu BLAKE3, OMP\" " \
                        "\"\" ljust at (%f,%f)\n", xmax / 8, ymax * 0.8
                for (i = 0; i < count; i++) {
                        if (!(i in char)) continue
                        printf "\"\\(%s\" at (%f,%f)\n", char[i], x[i], y[i]
                }
        }
’
```

TODO is there an easier way to do this with gawk?

15a ⟨*Convert bytes to human-readable units* 15a⟩≡ (14)

```
suffixes[0] = "B"
suffixes[1] = "KiB"
suffixes[2] = "MiB"
suffixes[3] = "GiB"
i = 0
suffix = 0
while (n > 1024) {
        n = n / 1024
        suffix++
}
return sprintf("%d%s", n, suffixes[suffix])
```

TODO improve this. This seems bad.

15b ⟨*Select and print ticks* 15b⟩≡ (14)

```
printf "ticks bot out at"
len = asorti(xtick)
comma = 0
for (i = 0; i < len; i += 2) {
        if (xtick[i] == 0) {
                i--
                continue
        }
        printf "%s%d \"%s\"", comma ? ", " : " ", xtick[i], tosize(xtick[i] * 32)
        comma = 1
}
printf "\n"
```

1.15 Generating Inputs

The following script generates inputs for run-bench.sh. It takes two arguments: the amount of bytes to generate and the maximum block size. It assumes 256-bit hashes.

Arguments are taken as powers of two. For example, to generate 16GiB of hashes with a maximum block size of 4GiB, use the command line below. Note that 2^{34} = 16GiB and 2^{31} = 2GiB.

```
./generate-inputs.sh 34 31
```

16 ⟨*generate-inputs.sh* 16⟩≡

```
#!/bin/sh

if [ $# -lt 2 ]
then
        printf 'usage: ./generate-inputs.sh TOTAL_SIZE MAX_BLOCK_SIZE\n' >&2
        printf '\tArguments are powers of 2, e.g., 32 gives 2^32 = 16GiB\n' >&2
        printf '\t2^0=1 2^1=2 2^2=4 2^3=8 2^4=16 2^5=32 2^6=64 2^7=128 ' >&2
        printf '2^8=256 2^9=512\n' >&2
        exit 1
fi

total_bytes=$1
max_block_bytes=$2
digest_size_bytes=5 # 256 bits = 32 bytes, 32=2^5

total=$((total_bytes - digest_size_bytes))
max_block=$((max_block_bytes - digest_size_bytes))

for b in $(seq 10 $max_block)
do
        printf '%d\t%d\n' $((1 << b)) $((1 << (total - b)))
done
```

1.16 Verifying Hashes

This script generates dumps of hashes generated on the CPU and GPU, then displays them side-by-side with less(1).

17 ⟨*check-dumps.sh* 17⟩≡

```
#!/bin/sh
./bench 1024 1024 blake3 serial
mv bench-hashes.dat bench-hashes-serial.txt
./bench 1024 1024 blake3 sycl-cpu
mv bench-hashes.dat bench-hashes-cpu.txt
./bench 1024 1024 blake3 sycl-gpu
mv bench-hashes.dat bench-hashes-gpu.txt
paste bench-hashes-serial.txt bench-hashes-cpu.txt bench-hashes-gpu.txt | less
wc -l bench-hashes-serial.txt bench-hashes-cpu.txt bench-hashes-gpu.txt
```

2 Index

2.1 Chunks

2.2 Identifiers