BENCHMARKING TECHNIQUES FOR PERFORMANCE ANALYSIS OF OPERATING SYSTEMS AND PROGRAMS

Submitted by:

SHUBHAM GUPTA	16BCE0232
UTKARSH SHARMA	16BCE0226
JEEVANJYOTI PRADHAN	16BCE0457
APOORVA J	16BCE2070
ISHTDEEP SINGH	16BCE2125

Under the Guidance of,

PROF. SHAIK NASEERA

(SCHOOL OF COMPUTER SCIENCE AND ENGINEERING)

October, 2017

ABSTRACT

Benchmarking of computer systems is an important, albeit sometimes tedious task that gives insight into the performance of a system, exposes flaws, and allows for comparison between systems or versions. Current benchmarking suites for the UNIX operating system are strongly weighted towards CPU and hardware performance. As operating systems grow more complex, so do the applications running on them. The type of open benchmarks that are widely available today do not take this into account, making them somewhat unrealistic and uninformative.
We present the comparison between the three commonly used operating systems chosen by us:

1) **Mac OS 10.12.6**
2) **Windows 10 Home 64**
3) **Linux OS**

We had chosen **Intel core i5** processor for our output generation.
We had used several tools in our attempt to compare and analyse the performance of these operating systems namely Geekbench4, Everest Ultimate edition etc. A detailed and comprehensive conclusion has been drawn at the last based on the outputs we had received after undertaking several tests.

INTRODUCTION

Benchmarks are used for performance evaluation of computers and are representatives of applications that run of actual systems. By continuously identifying, understanding, and adapting outstanding practices and processes found inside and outside a system, the performance can be improved.

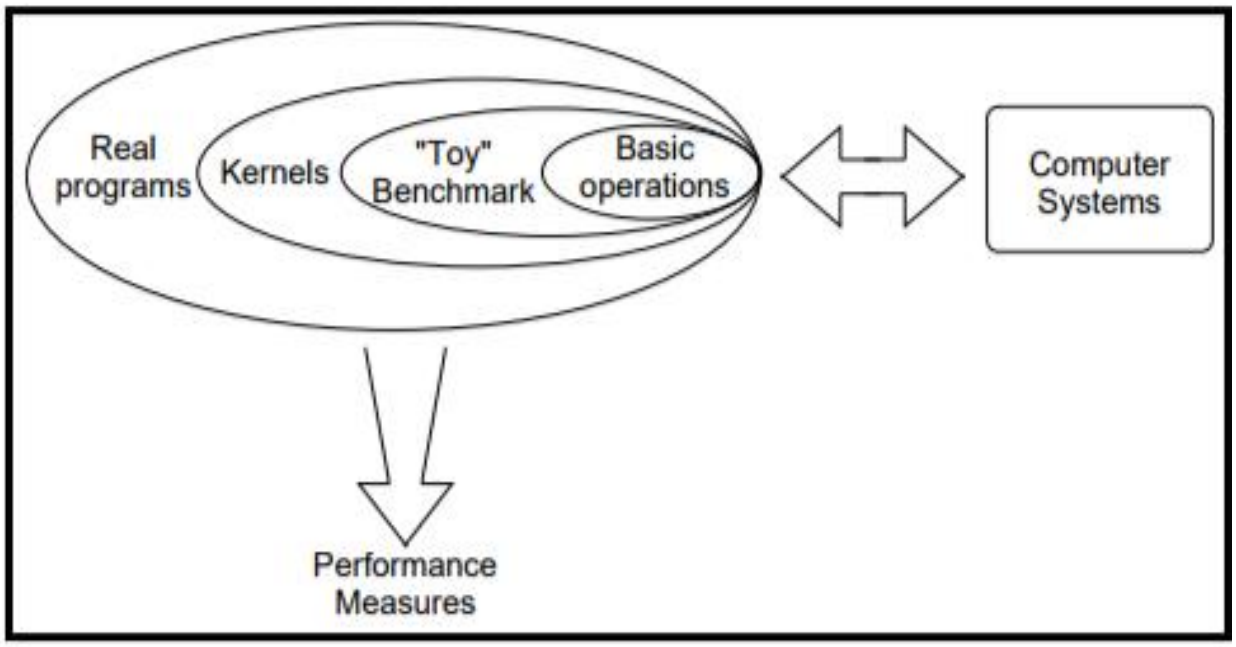
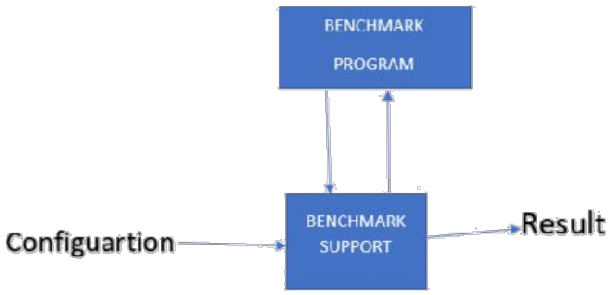


Figure 7.1. Benchmark hierarchy.

The prime focus of Benchmarking is on improvement of any given business process by exploiting "best practices" rather than merely measuring the best performance of an OS.



Benchmark system design

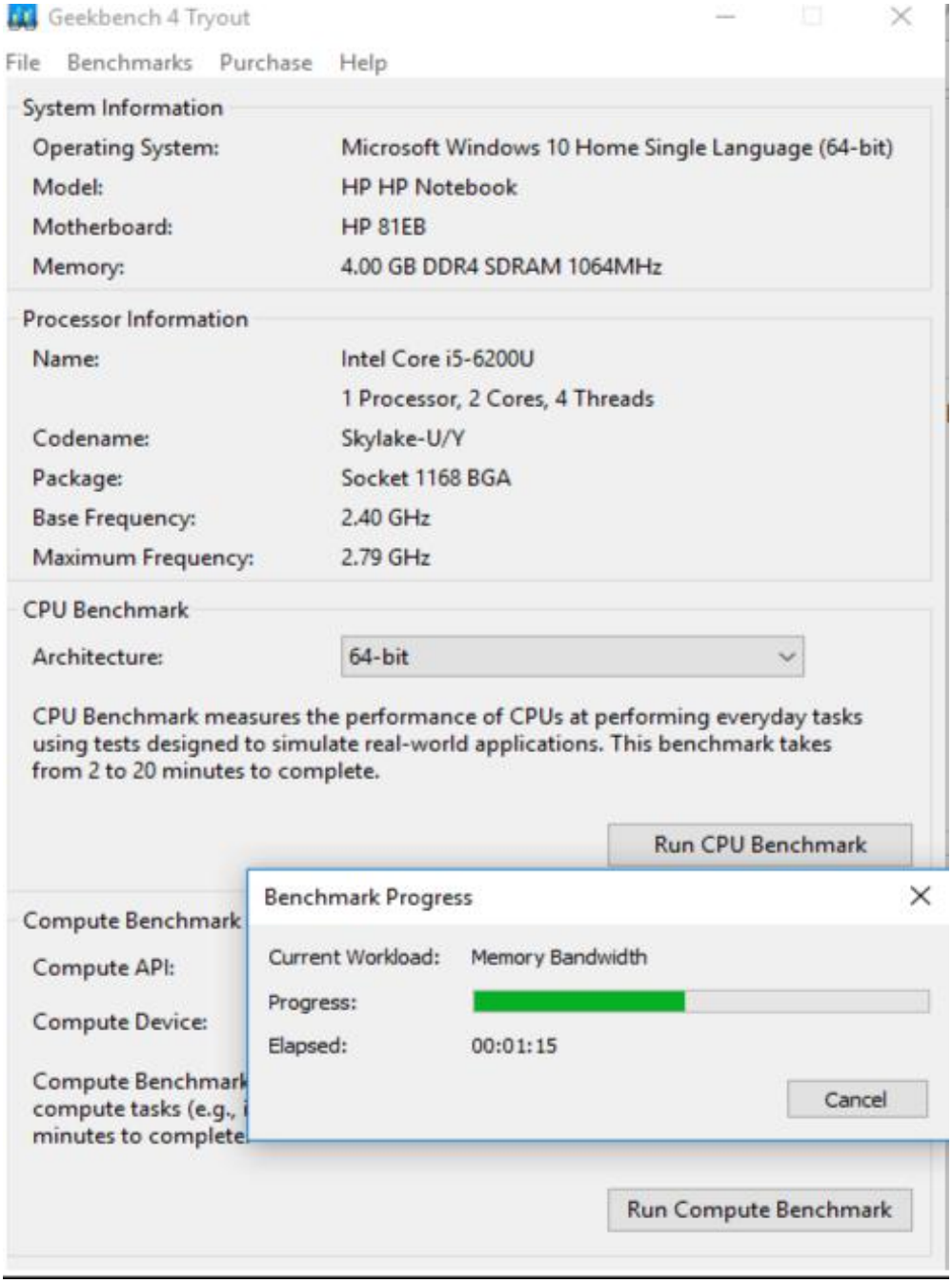
We will be using benchmarking software, GeekBench 4 on all the platforms to study overall performance of systems in every aspect.
And will be conducting a run of set of open source codes, manipulated according to the requirements to examine typical operating system's parameters.

Overview of Windows vs Linux vs MACOS:

Computer arch supported	Workstation ,personal computer ,media center,table pc, embedded	X86,x86-64, powerPc,SPARC,alpha,other	68k,powerPc
Gui default is on	OK	Depends on distribution	OK
Target system type	Workstation, personal computer, media center, tablet PC, embedded	Desktop/Server Depends on distribution	Workstation, personal computer
File system supported	NTFS, FAT exFAT ISO9660, UDF, 3rd party drivers support ext2, ext3, ReiserFS[t 10] HFS+, FATX and HFS(with third party driver)	Ext2, ext3, ext4, ReiserFS, FAT, ISO, 9660, UDF, NFS and others	HFS+, HFS, MFS(Mac OS 8.0 and before), AFP, ISO, 9660, Fat(system 7 and later), UDF
User friendly for lay users	Very user friendly	Depends on distribution	OK
Integrated firewall	Window firewall	Chroot, capability-based security, [s 5] secomp, SELinux	Application Firewall
Encrypted the system	OK	Ok	OK
System threads	huge	negligible	negligible
Shell terminal	CMD	Bash shell powerful shell with a lot of features	BASH
Kernel type	Hybrid	Monolithic with modules	Monolithic with modules

IMPLEMENTATION AND OUTPUTS:

1) WINDOWS OPERATING SYSTEM



HP HP Notebook

Single-Core Score	Multi-Core Score
3338	6129

Geekbench 4.1.3 Tryout for Windows x86 (64-bit)

Result Information

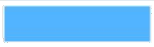
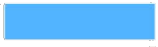
Upload Date	October 23 2017 06:03 PM
Views	1

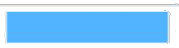
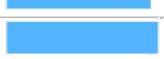
System Information

System Information	
Operating System	Microsoft Windows 10 Home Single Language (64-bit)
Model	HP HP Notebook
Motherboard	HP 81EB
Memory	4096 MB DDR4 SDRAM 1064MHz
Northbridge	Intel Skylake-U 08
Southbridge	Intel Skylake-Y PCH 31
BIOS	Insyde F.26

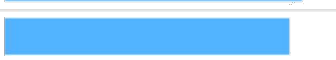
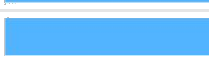
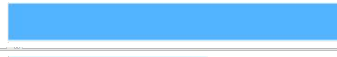
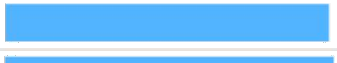
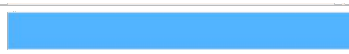
Processor Information	
Name	Intel Core i5-6200U
Topology	1 Processor, 2 Cores, 4 Threads
Identifier	GenuineIntel Family 6 Model 78 Stepping 3
Base Frequency	2.40 GHz
Maximum Frequency	2.79 GHz
Package	Socket 1168 BGA
Package	Skylake-U/Y
L1 Instruction Cache	32.0 KB x 2
L1 Data Cache	32.0 KB x 2
L2 Cache	256 KB x 2
L3 Cache	3.00 MB x 1

Single-Core Performance

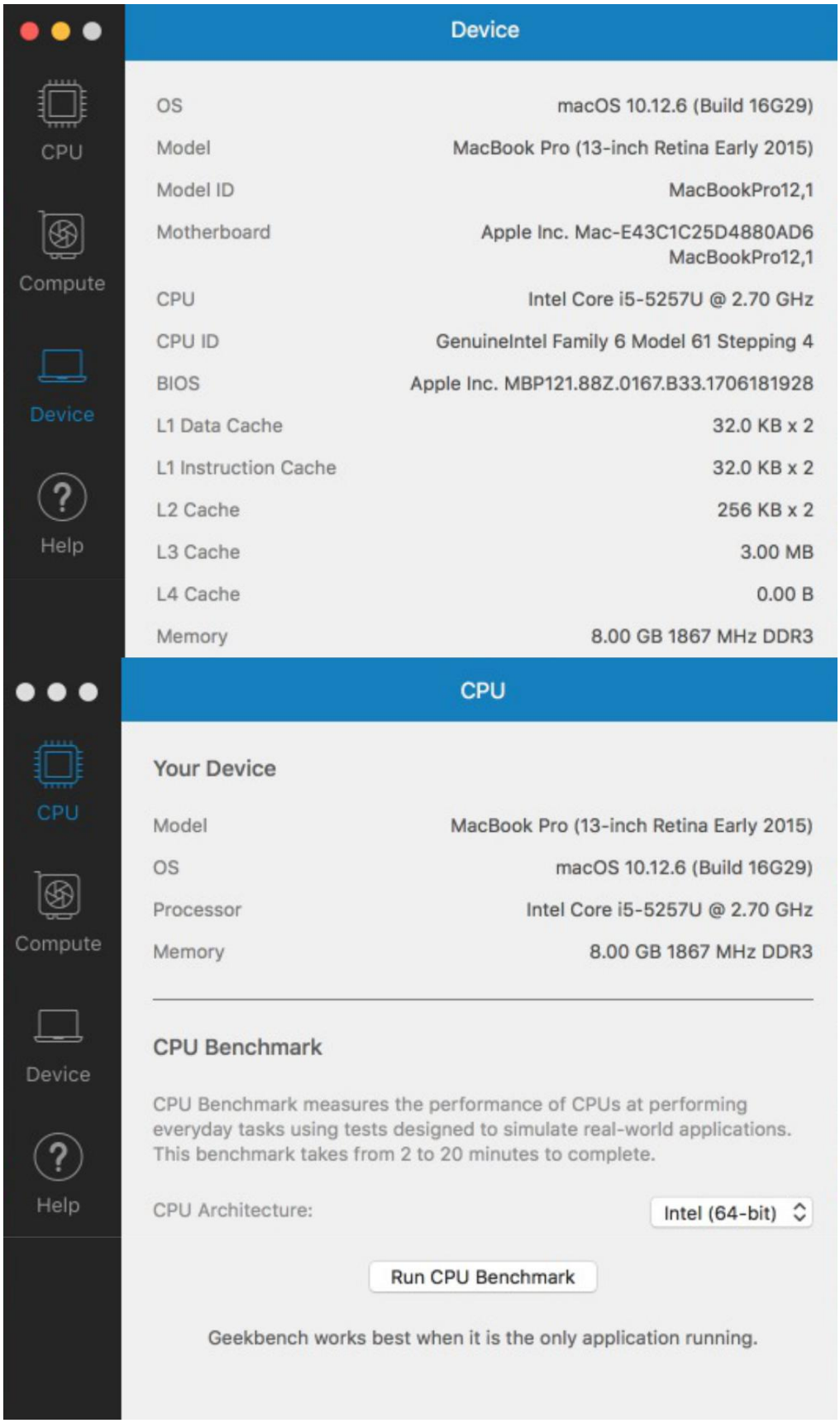
Single-Core Score	3338	
Crypto Score	3259	
Integer Score	3442	
Floating Point Score	3220	
Memory Score	3299	
AES	3259 2.45 GB/sec	
LZMA	3187 4.99 MB/sec	
JPEG	3293 20.6 Mpixels/sec	
Canny	3354 48.6 Mpixels/sec	
Lua	3232 3.32 MB/sec	
Dijkstra	3400 2.39 MTE/sec	
SQLite	3385 93.3 Krows/sec	
HTML5 Parse	3203 14.8 MB/sec	
HTML5 DOM	3784 3.45 MElements/sec	
Histogram Equalization	3239 161.2 Mpixels/sec	
PDF Rendering	3191 84.8 Mpixels/sec	

Camera	3374 9.36 Images/sec	
SGEMM	3248 68.7 Gflops	
SFFT	3428 6.65 Gflops	
N-Body Physics	3092 2.31 Mpairs/sec	
Ray Tracing	3195 466.6 Kpixels/sec	
Rigid Body Physics	3284 6656.0 FPS	
HDR	3434 12.5 Mpixels/sec	
Gaussian Blur	3061 53.6 Mpixels/sec	
Speech Recognition	3185 27.1 Words/sec	
Face Detection	3123 612.4 Kwin/sec	
Memory Copy	2885 7.99 GB/sec	
Memory Latency	6682 77.8 ns	
Memory Bandwidth	2288 12.6 GB/sec	

Multi-Core Performance

Multi-Core Score	6129	
Crypto Score	5345	
Integer Score	7233	
Floating Point Score	6570	
Memory Score	3181	
AES	5345 4.03 GB/sec	
LZMA	7207 91.9 MB/sec	
Dijkstra	7134 4.62 MTE/sec	
SQLite	7217 209.1 Rows/sec	
HTML5 Parse	6389 29.0 MB/sec	
HTML5 DOM	6089 5.52 MElements/sec	
Histogram Equalization	7081 221.3 Mpixels/sec	
PDF Rendering	6287 167.9 Mpixels/sec	
LLVM	12091 831.3 functions/sec	
Camera	7702 21.5 Images/sec	
SGEMM	4352 92.0 Gflops	
SFFT	5502 13.7 Gflops	
N-Body Physics	6948 5.19 Mpairs/sec	
Ray Tracing	7079 1.03 Mpixels/sec	
Rigid Body Physics	8257 24172.0 FPS	
HDR	8173 29.8 Mpixels/sec	
Gaussian Blur	6931 121.4 Mpixels/sec	
Speech Recognition	5735 49.1 Words/sec	
Face Detection	7228 2.11 Mwin/sec	
Memory Copy	2922 8.10 GB/sec	
Memory Latency	6407 80.1 ns	
Memory Bandwidth	2038 10.9 GB/sec	

2) macOS


Device
CPU
Compute
Device
Help
OS
macOS 10.12.6 (Build 16G29)
Model
MacBook Pro (13-inch Retina Early 2015)
Model ID
MacBookPro12,1
Motherboard
Apple Inc. Mac-E43C1C25D4880AD6
MacBookPro12,1
CPU
Intel Core i5-5257U @ 2.70 GHz
CPU ID
GenuineIntel Family 6 Model 61 Stepping 4
BIOS
Apple Inc. MBP121.88Z.0167.B33.1706181928
L1 Data Cache
32.0 KB x 2
L1 Instruction Cache
32.0 KB x 2
L2 Cache
256 KB x 2
L3 Cache
3.00 MB
L4 Cache
0.00 B
Memory
8.00 GB 1867 MHz DDR3
CPU
CPU
Compute
Device
Help
Your Device
Model
MacBook Pro (13-inch Retina Early 2015)
OS
macOS 10.12.6 (Build 16G29)
Processor
Intel Core i5-5257U @ 2.70 GHz
Memory
8.00 GB 1867 MHz DDR3
CPU Benchmark
CPU Benchmark measures the performance of CPUs at performing everyday tasks using tests designed to simulate real-world applications. This benchmark takes from 2 to 20 minutes to complete.
CPU Architecture:
Intel (64-bit)
Run CPU Benchmark
Geekbench works best when it is the only application running.

MacBook Pro (13-inch Retina Early 2015)

Single-Core Score	Multi-Core Score
3755	7276

Geekbench 4.1.3 Tryout for Mac OS X x86 (64-bit)

Result Information

Upload Date	October 23 2017 04:44 PM
Views	2

System Information

System Information	
Operating System	macOS 10.12.6 (Build 16G29)
Model	MacBookPro (13-inch Retina Early 2015)
Motherboard	Apple Inc. Mac-E43C1C25D4880AD6 MacBookPro12,1
Memory	8192 MB 1867 MHz DDR3
Northbridge	
Southbridge	
BIOS	Apple Inc. MBP121.88Z.0157.B33.1706181938

Processor Information	
Name	Intel Core i5-5257U
Topology	1 Processor, 2 Cores, 4 Threads
Identifier	GenuineIntel Family 6 Model 61 Stepping 4
Base Frequency	2.70 GHz
Package	
L1 Instruction Cache	32.0 KB x 2
L1 Data Cache	32.0 KB x 2
L2 Cache	256 KB x 2
L3 Cache	3.00 MB x 1

Single-Core Performance

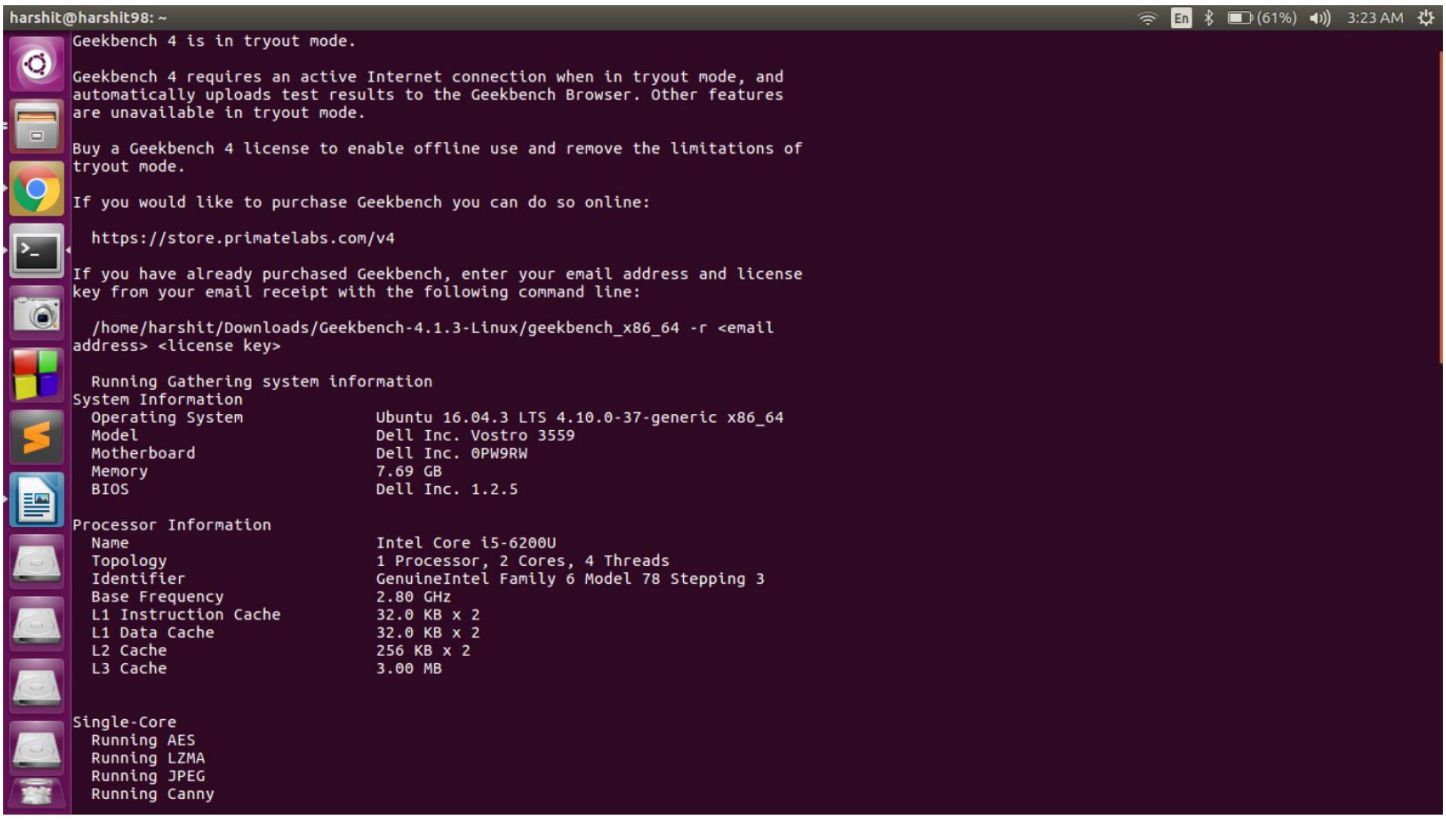
Single-Core Score	3755
Crypto Score	3128
Integer Score	3731
Floating Point Score	3721
Memory Score	4017
AES	3128 2.36 GB/sec
LZMA	3604 6.47 MB/sec
JPEG	3881 31.2 Mpixels/sec
Canny	3501 48.6 Mpixels/sec
Lua	3439 3.69 MB/sec
Dijkstra	3902 2.04 MTE/sec
SQLite	3345 92.8 Krows/sec
HTML5 Parse	3162 14.4 MB/sec
HTML5 DOM	4165 3.77 MElements/sec
Histogram Equalization	3351 655.9 Mpixels/sec
PDF Rendering	3283 67.2 Mpixels/sec
LLVM	5171 494.3 Functions/sec
Camera	3818

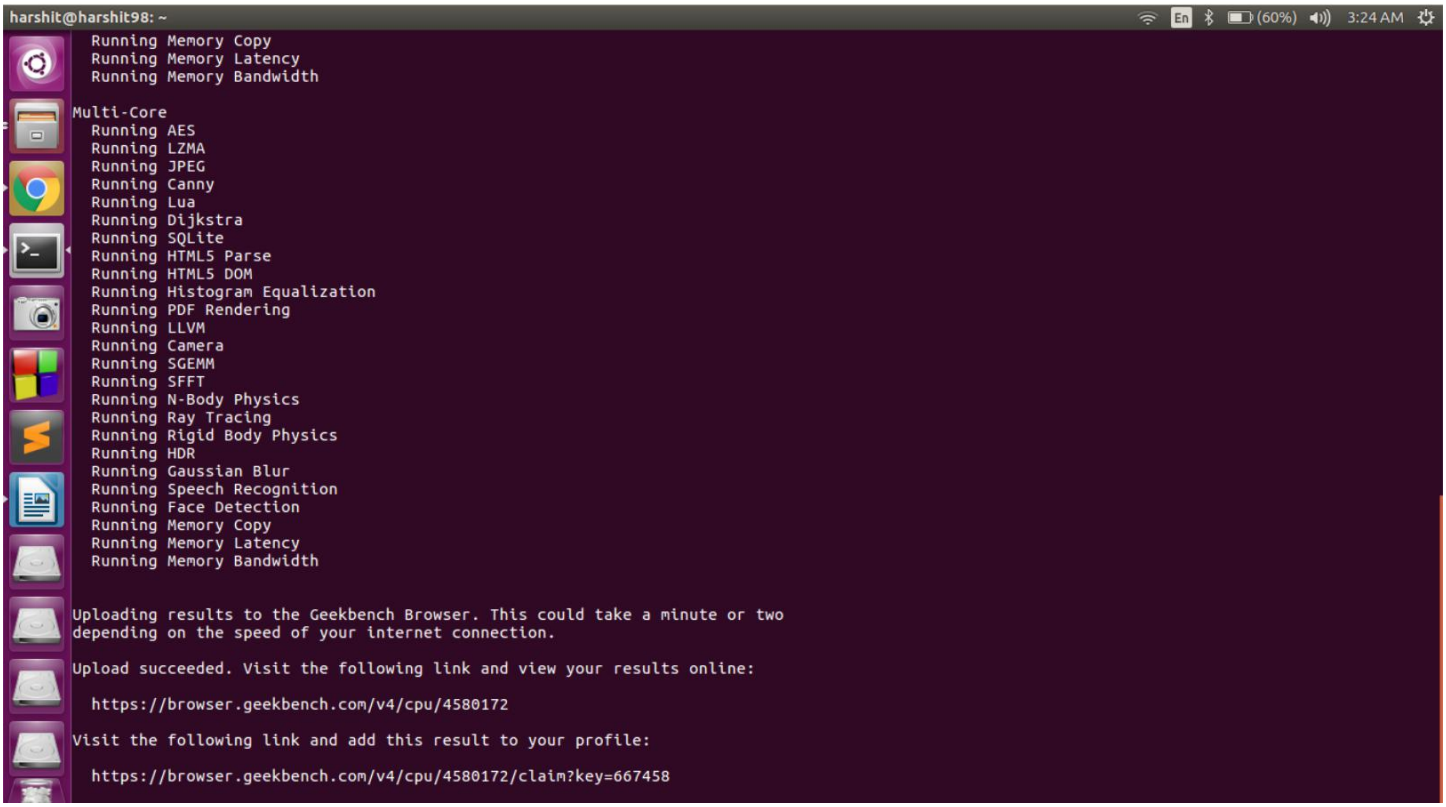
N-Body Physics	3512 2.79 Mpairs/sec	
Ray Tracing	3452 694.2 Kpixels/sec	
Rigid Body Physics	4561 13284.9 FPS	
HDR	4083 14.6 Mpixels/sec	
Gaussian Blur	3567 62.6 Mpixels/sec	
Speech Recognition	3432 29.4 Words/sec	
Face Detection	3633 1.05 Mwindows/sec	
Memory Copy	3364 9.32 GB/sec	
Memory Latency	5647 79.1 ns	
Memory Bandwidth	3475 18.6 GB/sec	

Multi-Core Performance

Multi-Core Score	7276
Crypto Score	6264
Integer Score	8162
Floating Point Score	7968
Memory Score	4499
AES	5284 4.72 GB/sec
LZMA	7930 12.4 MB/sec
SQLite	5594 182.8 Krows/sec
HTML5 Parse	5537 30.1 MB/sec
HTML5 DOM	5793 0.95 MElements/sec
Histogram Equalization	7648 235.9 Mpixels/sec
PDF Rendering	7359 105.4 Mpixels/sec
LLVM	14327 883.1 functions/sec
Camera	9510 28.6 images/sec
SGEMM	5847 122.6 Gflops
SFFT	8310 29.7 Gflops
N-Body Physics	8098 6.65 Mpairs/sec
Ray Tracing	7684 1.12 Mpixels/sec
Rigid Body Physics	11120 32653.0 FPS
HDR	9490 34.4 Mpixels/sec
Gaussian Blur	7795 136.8 Mpixels/sec
Speech Recognition	5337 84.2 Words/sec
Face Detection	8237 2.41 Mwindows/sec
Memory Copy	4382 12.1 GB/sec
Memory Latency	5367 83.7 ns
Memory Bandwidth	3892 20.8 GB/sec

3) LINUX OPERATING SYSTEM





Single-Core Score	Multi-Core Score
3386	6639

Geekbench 4.1.3 Tryout for Linux x86 (64-bit)

Result Information

Upload Date	October 23 2017 04:12 PM
Views	1

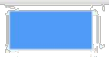
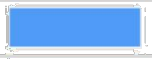
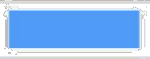
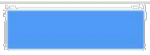
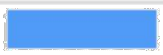
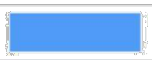
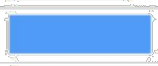
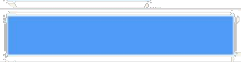
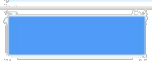
System Information

System Information	
Operating System	Ubuntu 16.04.3 LTS 4.10.0-37-generic x86_64
Model	Dell Inc. Vostro 3559
Motherboard	Dell Inc. 0PW9RW
Memory	7873 MB
BIOS	Dell Inc. 1.2.5

Processor Information	
Name	Intel Core i5-6200U
Topology	1 Processor, 2 Cores, 4 Threads
Identifier	GenuineIntel Family 6 Model 78 Stepping 3
Base Frequency	2.80 GHz
L1 Instruction Cache	32.0 KB x 2
L1 Data Cache	32.0 KB x 2
L2 Cache	256 KB x 2
L3 Cache	3.00 MB x 1

Single-Core Performance

Single-Core Score	3386
Crypto Score	2214
Integer Score	3551
Floating Point Score	3265
Memory Score	3489

AES	2214 1.57 GB/sec	
LZMA	2576 8.03 MB/sec	
JPEG	3524 28.4 Mpixels/sec	
Canny	3499 48.9 Mpixels/sec	
Lua	3570 3.67 MB/sec	
Dijkstra	3856 2.61 MTE/sec	
SQLite	3503 97.1 Rows/sec	
HTML5 Parse	3524 18.0 MB/sec	
HTML5 DOM	2874 2.69 MElements/sec	
Histogram Equalization	3112 97.5 Mpixels/sec	
PDF Rendering	3838 102.5 Mpixels/sec	
LLVM	5880 458.3 Functions/sec	
Camera	3692 19.2 Images/sec	

N-Body Physics	3340
Ray Tracing	1995
Rigid Body Physics	3285
HDR	4255
Gaussian Blur	3531
Speech Recognition	3255
Face Detection	3017
Memory Copy	3482
Memory Latency	4287
Memory Bandwidth	2847

Multi-Core Performance

Multi-Core Score	**6639**
Crypto Score	**6634**
Integer Score	**7769**
Floating Point Score	**6998**
Memory Score	**3558**
AES	6634
LZMA	6522
JPEG	6245
Canny	7814
Lua	7206
Dijkstra	9450
SQLite	7474
HTML5 Parse	7273
HTML5 DOM	3532
Histogram Equalization	7291
PDF Rendering	6902
LLVM	15401
Camera	8914
SGEMM	6549
SFFT	7229
N-Body Physics	6390
Ray Tracing	6386
Rigid Body Physics	8799
HDR	9508
Gaussian Blur	7372
Speech Recognition	5683
Face Detection	5949
Memory Copy	3611
Memory Latency	4253

CODES for RAW KERNEL AND FILE SYSTEM PERFORMANCE

Memory allocation:

```
//  This is free and unencumbered software released into the public domain.
//  For more information, see UNLICENSE.

#include "common/time.h"

#include <stdio.h>

#if defined(_WIN32)
#define WIN32_LEAN_AND_MEAN
#include <process.h>
#include <windows.h>
#else
#include <pthread.h>
#endif

#if defined(_WIN32)

//  WIN32 thread implementation.
typedef HANDLE thread_t;

static unsigned WINAPI thread_fun(void* arg) {
 //  We do nothing here...
 (void)arg;
 return 0u;
}

static thread_t create_thread() {
 return (HANDLE)_beginthreadex((void*)0, 0, thread_fun, (void*)0, 0, (unsigned*)0);
}

static void join_thread(thread_t thread) {
 if (WaitForSingleObject(thread, INFINITE) != WAIT_FAILED) {
   CloseHandle(thread);
 }
}

#else

//  POSIX thread implementation.
typedef pthread_t thread_t;

static void* thread_fun(void* arg) {
 //  We do nothing here...
 (void)arg;
 return (void*)0;
}

static thread_t create_thread() {
 thread_t result;
 pthread_create(&result, (const pthread_attr_t*)0, thread_fun, (void*)0);
 return result;
}

static void join_thread(thread_t thread) {
 pthread_join(thread, (void**)0);
}
```

```
#endif // WIN32

#define NUM_THREADS 100

static const double BENCHMARK_TIME = 5.0;

int main() {
 printf("Benchmark: Create/teardown of %d threads...\n",
 NUM_THREADS); fflush(stdout);

 double best_time = 1e9;
 const double start_t = get_time();
 while (get_time() - start_t < BENCHMARK_TIME) {
  thread_t threads[NUM_THREADS];
  const double t0 = get_time();

  // Create all the child threads.
  for (int i = 0; i < NUM_THREADS; ++i) {
   threads[i] = create_thread();
  }

  //  Wait for all the child threads to finish.
  for (int i = 0; i < NUM_THREADS;
  ++i) { join_thread(threads[i]);
  }

  double dt = get_time() - t0;
  if (dt < best_time) {
   best_time = dt;
  }
 }

 printf("%f us / thread\n", (best_time / (double)NUM_THREADS) *
 1000000.0); fflush(stdout);
 return 0;
}
```

Create Threads:

```
#include "common/time.h"

#include <stdio.h>

#if defined(_WIN32)
#define WIN32_LEAN_AND_MEAN
#include <process.h>
#include <windows.h>
#else
#include <pthread.h>
#endif

#if defined(_WIN32)

//  WIN32 thread implementation.
typedef HANDLE thread_t;

static unsigned WINAPI thread_fun(void* arg) {
 // We do nothing here...
```

```
 (void)arg;
 return 0u;
}

static thread_t create_thread() {
 return (HANDLE)_beginthreadex((void*)0, 0, thread_fun, (void*)0, 0, (unsigned*)0);
}

static void join_thread(thread_t thread) {
 if (WaitForSingleObject(thread, INFINITE) != WAIT_FAILED) {
  CloseHandle(thread);
 }
}

#else

//  POSIX thread implementation.
typedef pthread_t thread_t;

static void* thread_fun(void* arg) {
 //  We do nothing here...
 (void)arg;
 return (void*)0;
}

static thread_t create_thread() {
 thread_t result;
 pthread_create(&result, (const pthread_attr_t*)0, thread_fun, (void*)0);
 return result;
}

static void join_thread(thread_t thread) {
 pthread_join(thread, (void**)0);
}

#endif // WIN32

#define NUM_THREADS 100

static const double BENCHMARK_TIME = 5.0;

int main() {
 printf("Benchmark: Create/teardown of %d threads...\n",
 NUM_THREADS); fflush(stdout);

 double best_time = 1e9;
 const double start_t = get_time();
 while (get_time() - start_t < BENCHMARK_TIME) {
  thread_t threads[NUM_THREADS];
  const double t0 = get_time();

  // Create all the child threads.
  for (int i = 0; i < NUM_THREADS; ++i) {
   threads[i] = create_thread();
  }

  //  Wait for all the child threads to finish.
  for (int i = 0; i < NUM_THREADS; ++i) {
```

```
    join_thread(threads[i]);
   }

   double dt = get_time() - t0;
   if (dt < best_time) {
    best_time = dt;
   }
  }

  printf("%f us / thread\n", (best_time / (double)NUM_THREADS) *
  1000000.0); fflush(stdout);

  return 0;
}
```

Create Files:

```
// This is free and unencumbered software released into the public domain.
// For more information, see UNLICENSE.

#include <time.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

static const double BENCHMARK_TIME = 5.0;

static const int NUM_FILES = 65534;

static double my_log2(double x) {
 static const double LOG2SCALE = 1.442695040888963;
 return log(x) * LOG2SCALE;
}

static int num_hex_chars(int max_int) {
 int num_bits = (int)ceil(my_log2((double)max_int));
 return (num_bits + 3) / 4;
}

static char path_separator() {
#if defined(_WIN32)
 return '\\';
#else
 return '/';
#endif
}

static void to_hex(int x, char* str, const int str_len) {
 static const char TOHEX[16] = {
  '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'a', 'b', 'c', 'd', 'e', 'f'
 };
```

```
  str += str_len - 1;
  for (int i = 0; i < str_len; ++i) {
    *str-- = TOHEX[x & 15];
    x = x >> 4;
  }
}

static void create_file(const char* file_name) {
  static const char FILE_DATA[32] = {
    0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25,
    26, 27, 28, 29, 30, 31
  };

  FILE *f = fopen(file_name, "wb");
  if (!f) {
    fprintf(stderr, "*** Unable to create file \"%s\"\n", file_name);
    exit(1);
  }
  fwrite(FILE_DATA, 1, sizeof(FILE_DATA), f);
  fclose(f);
}

static void delete_file(const char* file_name) {
  remove(file_name);
}

int main(int argc, const char** argv) {
                //time_t sec;
  //   sec = time
  (NULL); if (argc != 2)
  {
    printf("Usage: %s root-folder\n",
    argv[0]); exit(1);
  }

  printf("Benchmark: Create/delete %d files...\n", NUM_FILES);
  fflush(stdout);

  // Create a path string.
  int hex_len = num_hex_chars(NUM_FILES - 1);
  size_t root_path_len = strlen(argv[1]);
  size_t path_len = root_path_len + 1 + hex_len;
  char* file_name = (char*)malloc(path_len + 1);
  if (!file_name) {
    fprintf(stderr, "*** Out of memory!\n");
    exit(1);
  }
  strncpy(file_name, argv[1], root_path_len);
  file_name[root_path_len] = path_separator();
  file_name[path_len] = 0;

  double best_time = 1e9;
```

```
 const time_t start_t = time(NULL);
 while (time(NULL) - start_t < BENCHMARK_TIME) {
  const double t0 = time(NULL);

  for (int file_no = 0; file_no < NUM_FILES; ++file_no) {
   // Construct the file name for this file.
   to_hex(file_no, &file_name[root_path_len + 1], hex_len);

   //  Create the file.
   create_file(file_name);
  }

  for (int file_no = 0; file_no < NUM_FILES; ++file_no) {
   // Construct the file name for this file.
   to_hex(file_no, &file_name[root_path_len + 1], hex_len);

   //  Delete the file.
   delete_file(file_name);
  }
  double dt = time(NULL) - t0;
  if (dt < best_time) {
   best_time = dt;
  }
 }
 printf("%f us / file\n", (best_time / (double)NUM_FILES) * 1000000.0);
 fflush(stdout);
 free((void*)file_name);
 return 0;
}
```

Create processes:

```
//  This is free and unencumbered software released into the public domain.
//  For more information, see UNLICENSE.

#include "common/time.h"

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

static const double BENCHMARK_TIME = 5.0;
static const int NUM_PROCESSES = 100;

int main() {
 printf("Benchmark: Create/teardown of %d processes...\n",
 NUM_PROCESSES); fflush(stdout);

 double best_time = 1e9;
 const double start_t = get_time();
```

```
 while (get_time() - start_t < BENCHMARK_TIME) {
  pid_t processes[NUM_PROCESSES];
  const double t0 = get_time();

  // Create all the processes.
  for (int i = 0; i < NUM_PROCESSES; ++i) {
   pid_t pid = fork();
   if (pid == 0) {
    exit(0);
   } else if (pid > 0) {
   processes[i] = pid;
   } else {
    fprintf(stderr, "*** Unable to create process no. %d\n", i);
    exit(1);
   }
  }

  //  Wait for all child processes to terminate.
  for (int i = 0; i < NUM_PROCESSES; ++i)
  { waitpid(processes[i], (int*)0, 0);
  }
  double dt = get_time() - t0;
  if (dt < best_time) {
  best_time = dt;
  }
 }
 printf("%f us / process\n", (best_time / (double)NUM_PROCESSES) *
 1000000.0); fflush(stdout);
 return 0;
}
```

Launch programs:

```
#include "common/time.h"

#include <stdio.h>
#include <stdlib.h>

#if defined(_WIN32)

//  Windows implementation. #define
WIN32_LEAN_AND_MEAN
#include <string.h>
#include <windows.h>

typedef struct {
 HANDLE hProcess;
 HANDLE hThread;
} process_t;

  static process_t create_process(const char* arg0, int process_no) {
      // The program name is the first argument to the parent process.
```

```
 const char* prg_name = arg0;

 // Construct a valid command line.
 // TODO(m): We could do this outside of the benchmark
 loop. size_t prg_name_len = strlen(prg_name);
 char* cmd_line = (char*)malloc(prg_name_len +
 5); if (!cmd_line) {
  fprintf(stderr, "*** Out of memory (process #%d)\n", process_no);
  exit(1);
 }
 strncpy(&cmd_line[1], prg_name, prg_name_len);
 cmd_line[0] = "";
 cmd_line[prg_name_len + 1] =
 "";  cmd_line[prg_name_len + 2]
 = ' '; cmd_line[prg_name_len + 3]
 = '-'; cmd_line[prg_name_len + 4]
 = 0;

 // Start the child process.
 STARTUPINFO startup_info;
 memset(&startup_info, 0, sizeof(STARTUPINFO));
 startup_info.cb = sizeof(STARTUPINFO);
 PROCESS_INFORMATION process_info;
 if (!CreateProcess(prg_name, cmd_line, NULL, NULL, TRUE, 0, NULL, NULL, &startup_info,
  &process_info)) { fprintf(stderr, "*** Unable to create process no. %d\n", process_no);
  exit(1);
 }

 free((void*)cmd_line);
 return (process_t){ process_info.hProcess, process_info.hThread };
}

static void wait_process(process_t pid) {
 WaitForSingleObject(pid.hProcess, INFINITE);
 CloseHandle(pid.hProcess);
 CloseHandle(pid.hThread);
}

#else

// Unix implementation.
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

typedef pid_t process_t;

static process_t create_process(const char* arg0, int process_no) {
 pid_t pid = fork();
 if (pid == 0) {
  execlp(arg0, arg0, "-", (char*)0);
 } else if (pid < 0) {
```

```
   fprintf(stderr, "*** Unable to create process no. %d\n", process_no);
   exit(1);
  }
  return (process_t)pid;
}

static void wait_process(process_t pid) {
  waitpid(pid, (int*)0, 0);
}

#endif

#define NUM_PROGRAMS 100

static const double BENCHMARK_TIME = 5.0;

int main(int argc, char** argv) {
  //  When called as a child-process, the first argument is defined.
  if (argc > 1) {
    exit(0);
  }

  printf("Benchmark: Launch %d programs...\n",
  NUM_PROGRAMS); fflush(stdout);

  double best_time = 1e9;
  const double start_t = get_time();
  while (get_time() - start_t < BENCHMARK_TIME) {
    process_t processes[NUM_PROGRAMS];
    const double t0 = get_time();

    // Create all the processes.
    for (int i = 0; i < NUM_PROGRAMS; ++i) {
      processes[i] = create_process(argv[0], i);
    }

    //  Wait for all child processes to terminate.
    for (int i = 0; i < NUM_PROGRAMS; ++i)
    { wait_process(processes[i]);
    }

    double dt = get_time() - t0;
    if (dt < best_time) {
      best_time = dt;
    }
  }

  printf("%f us / program\n", (best_time / (double)NUM_PROGRAMS) *
  1000000.0); fflush(stdout);
  return 0;
}
```

COMPARISION:

1. Single Thread CPU benchmark:

The Single Thread CPU benchmark, like all processor benchmarks attempts to estimate how quickly a processor is able to perform a wide variety of calculations.
The test issues as series of complex instructions to the processor and times how long the processor takes to complete the tasks.
The faster the processor is able to complete the tasks, the higher the benchmark score.
The GeekBench Single Thread CPU test only runs one stream of instructions rather than multiple parallel streams per core.
The majority of consumer applications (MS World, Internet Explorer, Google Chrome and most games), although multi-threaded, rarely utilize more than one thread at a time, so this test, like any single threaded benchmark, can be seen as a reasonable real world test for typical consumer workloads.

2. Multi Thread CPU benchmark:

Many types of programs can take advantage of threading as a program design benefit to make the program more responsive to the user, *Improved performance and concurrency, imultaneous access to multiple applications*, etc. Like, MS World, Internet Explorer, Google Chrome and most games.
Image processing can often be done in parallel e.g. split the image into 4 and do the work in 1/4 of the time but it depends upon the algorithm being run.
Rendering of animation (from 3DMax,etc.) is massively parallel as each frame can be rendered independently to others -- meaning that 10's or 100's of computers can be chained together to help out.
GUI programming often helps to have at least two threads when doing something slow, like processing large number of files - this allows the interface to remain responsive. So, it is important to conduct Multi Thread CPU benchmark.

Parameters and Their Definitions:

a) Memory Bandwidth:

Memory bandwidth is the rate at which data can be read from or stored into a semiconductor memory by a processor. Memory bandwidth is usually expressed in units of bytes/second.

b) Memory Latency and Memory copy:

Memory latency "is the length of time between the memory's receipt of a read request and its release of data corresponding with the request."

c) **HTML parser:**
The HTML parser is one of the most complicated and sensitive pieces of a browser. It controls how your HTML source code is turned into web pages and, as such, changes to it are rare. It Helps to test Web-browser functionability.

d) **Dijkstra:**
Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks. Dijkstra's original algorithm does not use a min-priority queue and runs in time.

e) **Canny:**
This is an implementation of the Canny Edge Detection algorithm using Open in C++. OpenCL applications can be heavily optimized (or broken) depending on the target hardware, the range of target hardware has been significantly limited

f) **Ray tracking:**
In computer graphics, ray tracing is a rendering technique for generating an image by tracing the path of light as pixels in an image plane and simulating the effects of its encounters with virtual objects.

g) **Advanced Encryption Standard (AES):**
Advanced Encryption Standard, a symmetric 128-bit block data encryption technique. AES works at multiple network layers simultaneously. Helps to compute encoding and decoding speed.

h) **HDR:**
We recommend an Internet connection speed of at least 25 megabits per second to stream titles available in HDR and Dolby Vision at Ultra HD 4K resolution. If your Internet connection speed is lower, you can still enjoy HDR at lower resolutions like 1080p.

i) **Gaussian blur:**
In image processing, a Gaussian blur (also known as Gaussian smoothing) is the result of blurring an image by a Gaussian function. It is a widely used effect in graphics software, typically to reduce image noise and reduce detail.

j) **Face defection:**
Face detection is an algorithm being used in a variety of applications that identifies human faces in digital images. Helps to test graphics.

3. RAW KERNEL AND FILE SYSTEM PERFORMANCE:

These single threaded and multi-threaded applications to be used efficiently needs certain software resources to be present on a computer. These prerequisites are known as **system requirements.**
So, keeping in mind all the system requirements to run single threaded and multi-threaded applications we will conduct certain tests on these resources provided by operating systems(thread management, address spaces and interprocess communication).

To get a clearer picture, we set out to benchmark some typical core primitives of operating systems. This includes:

a) Process creation.
b) Thread creation.
c) File creation.
d) Memory allocation.
e) Launching programs

To build the benchmarks we need typical C compiler, meson and ninja, to focus on speed and time more accurately and precisely.

Ninja:

Ninja is a small build system with a focus on speed. It differs from other build systems in two major respects: it is designed to have its input files generated by a higher-level build system, and it is designed to run builds as fast as possible.

Meson:

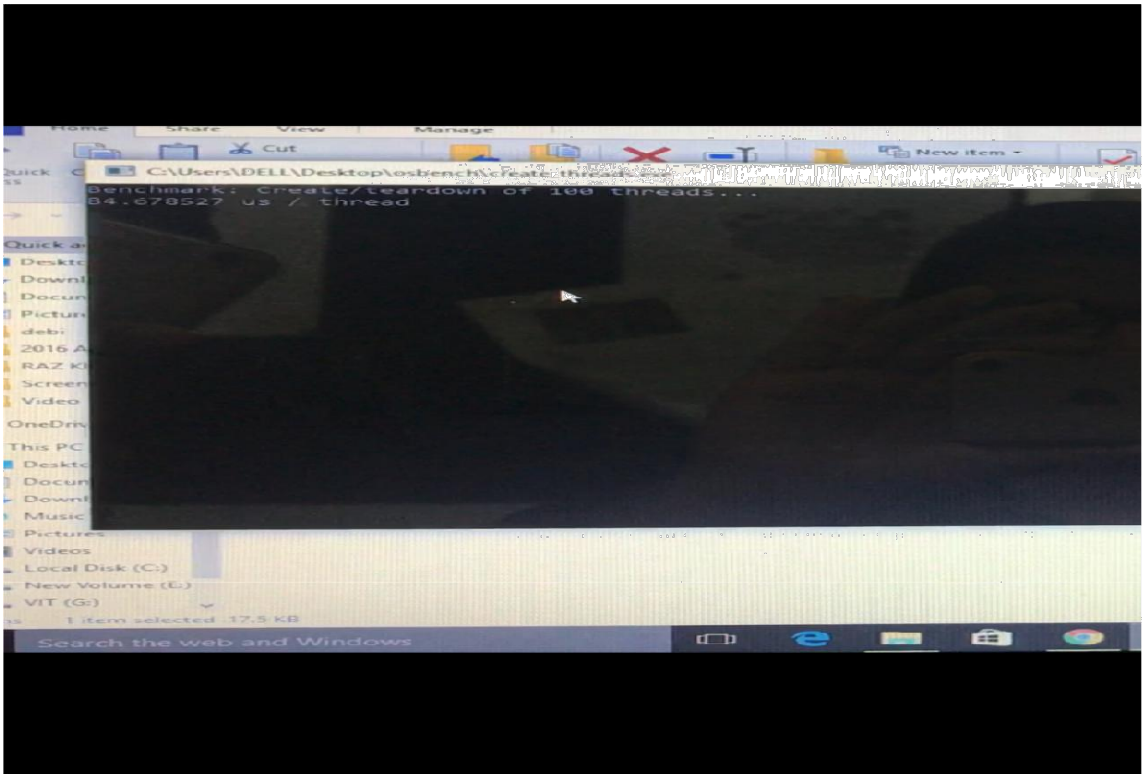
Meson is an open source build system meant to be both extremely fast, and, even more importantly, as user friendly as possible.

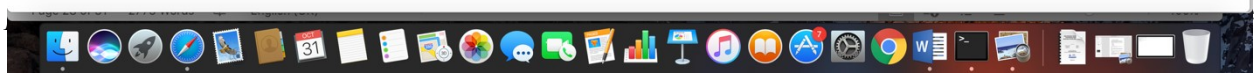
a) Creating threads

In this benchmark 100 threads are created. Each thread terminates immediately without doing any work, and the main thread waits for all child threads to terminate. The time it takes for a single thread to start and terminate is measured.

Apparently macOS is about twice as fast as Windows at creating threads, whereas Linux is about three times faster than Windows.

```
Terminal  Shell  Edit  View  Window  Help
jeevanjyotipradhan — create_threads — 181×49
Last login: Tue Oct 31 09:02:37 on ttys000
jeevanjyotis-MacBook-Pro:~ jeevanjyotipradhan$ /Users/jeevanjyotipradhan/out/builddr/create_threads ; exit;
Benchmark: Create/teardown of 100 threads...
16.250610 us / thread
logout
Saving session...
...copying shared history...
...saving history...truncating history files...
...completed.

[Process completed]
```

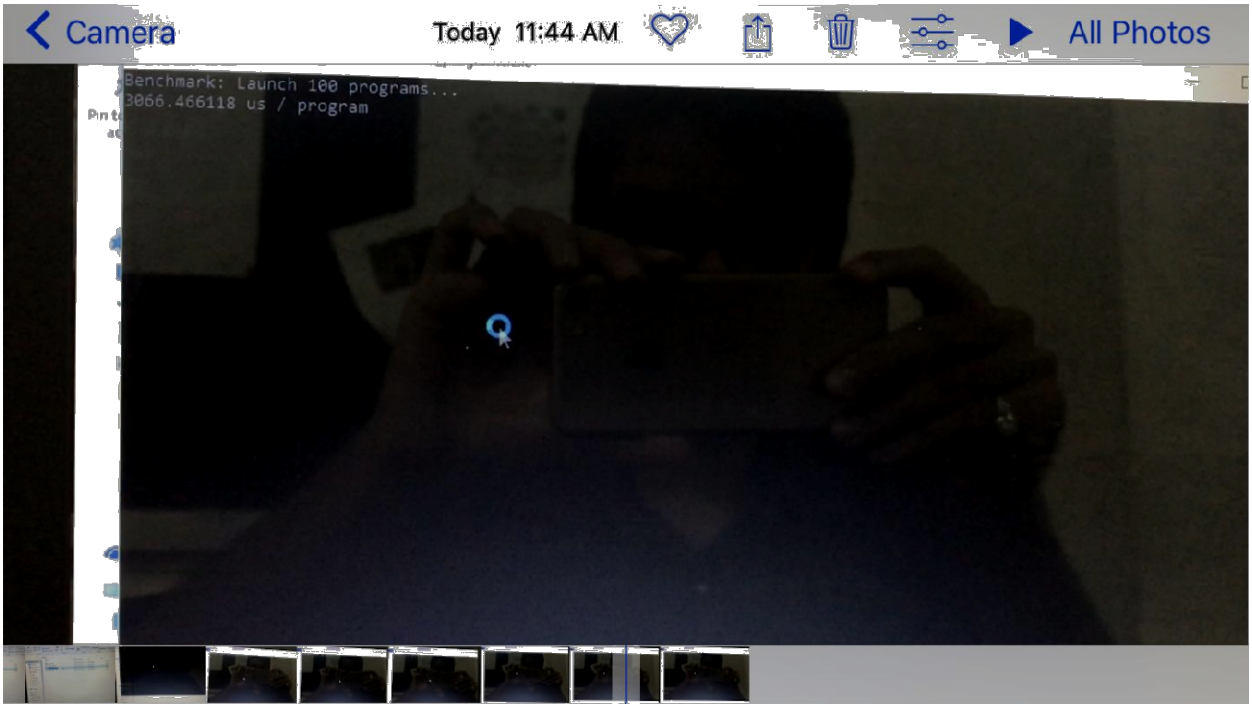




c) Launching programs

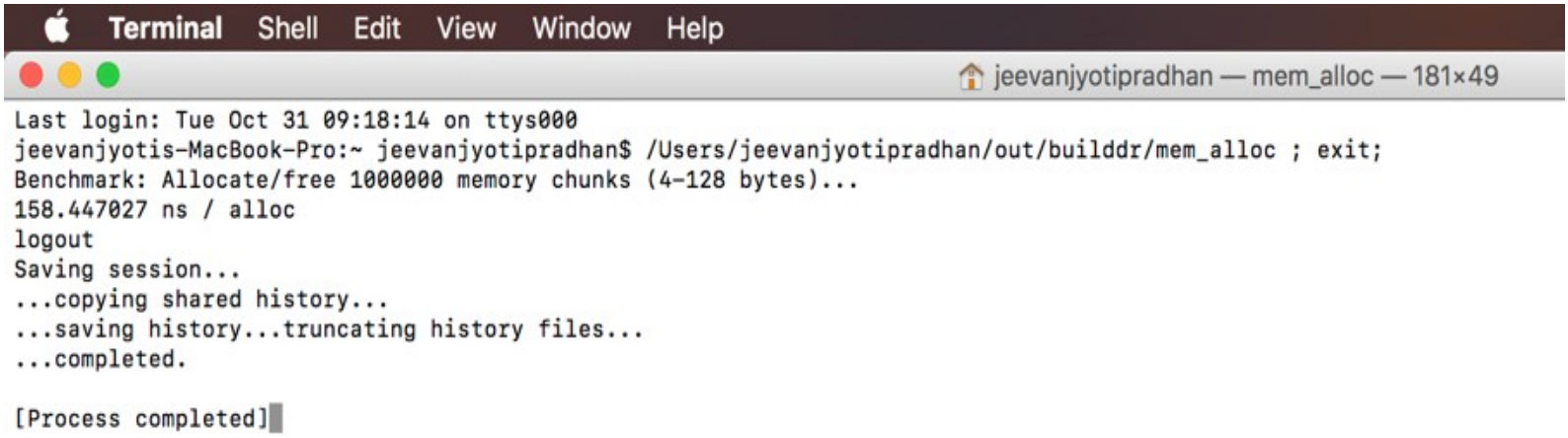
Launching a program is essentially an extension to process creation: in addition to creating a new process, a program is loaded and executed (the program consists of an empty **main()** function and exists immediately). On Linux and macOS this is done using **fork()** + **exec()**, and on Windows it is done using **CreateProcess()**.

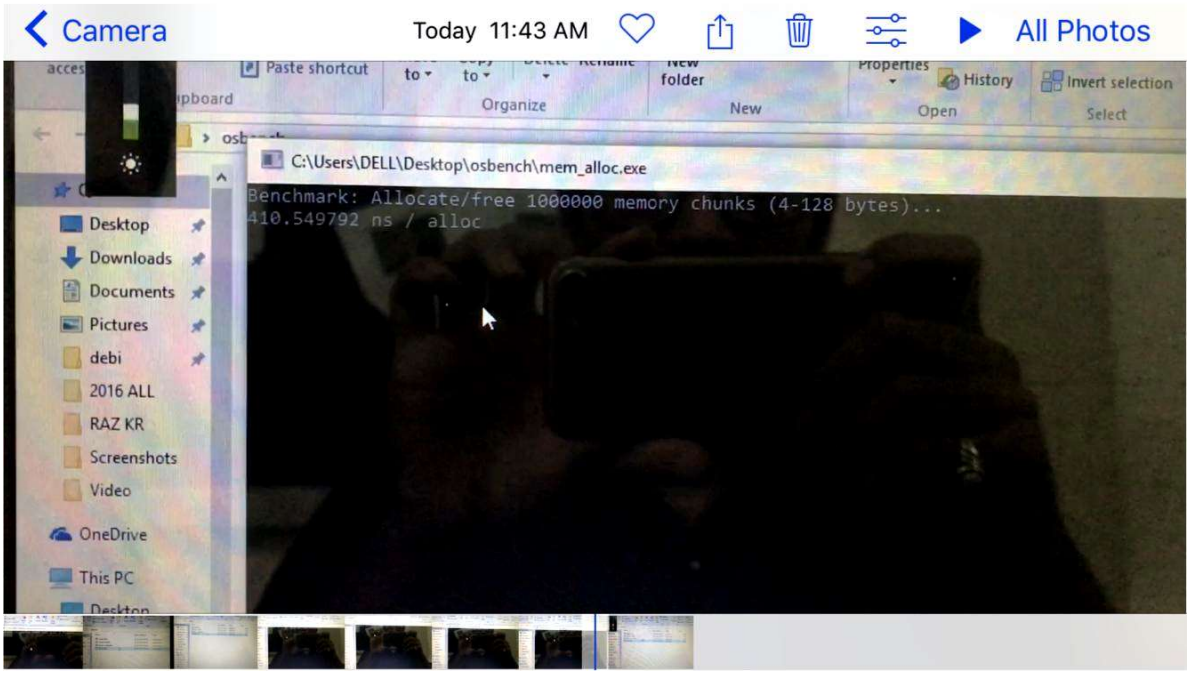
```
Terminal  Shell  Edit  View  Window  Help
jeevanjyotipradhan — launch_programs — 181×49
Last login: Tue Oct 31 09:16:11 on ttys000
jeevanjyotis-MacBook-Pro:~ jeevanjyotipradhan$ /Users/jeevanjyotipradhan/out/builddr/launch_programs ; exit;
Benchmark: Launch 100 programs...
970.530510 us / program
logout
Saving session...
...copying shared history...
...saving history...truncating history files...
...completed.

[Process completed]
```



d) Allocating memory

The memory allocation performance was measured by allocating 1,000,000 small memory blocks (4-128 bytes in size) and then freeing them again.





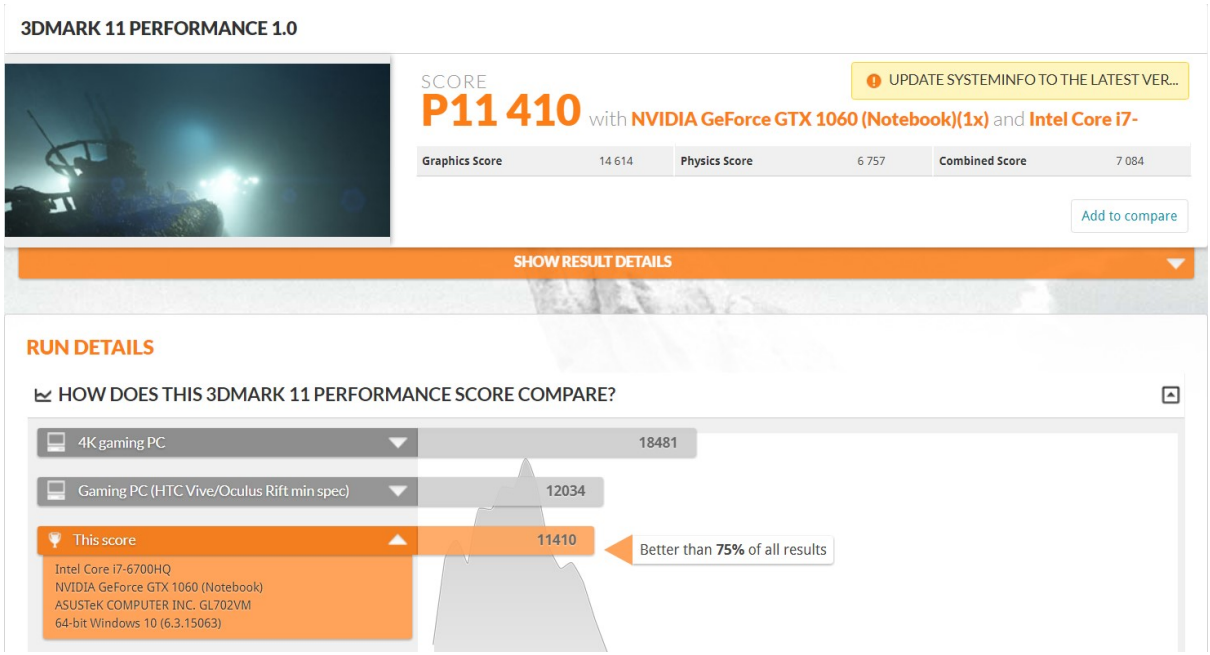
GPU Benchmarking

GPU used – Nvidia Geforce GTX 1060 (notebook version)
GPU specifications –
NVIDIA cuda cores – 1280
Graphic card memory – 6GB GDDR5
Base clock - 1506 Mhz
Boost clock – 1708 Mhz
Memory interface – 192bit

Benchmarking softwares used :

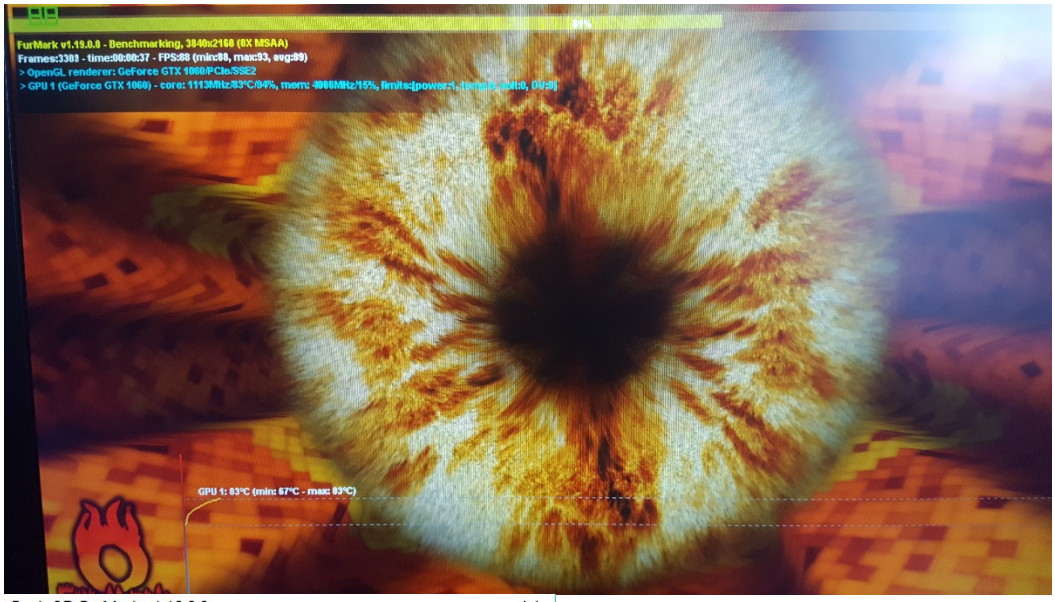
1.Futuremark

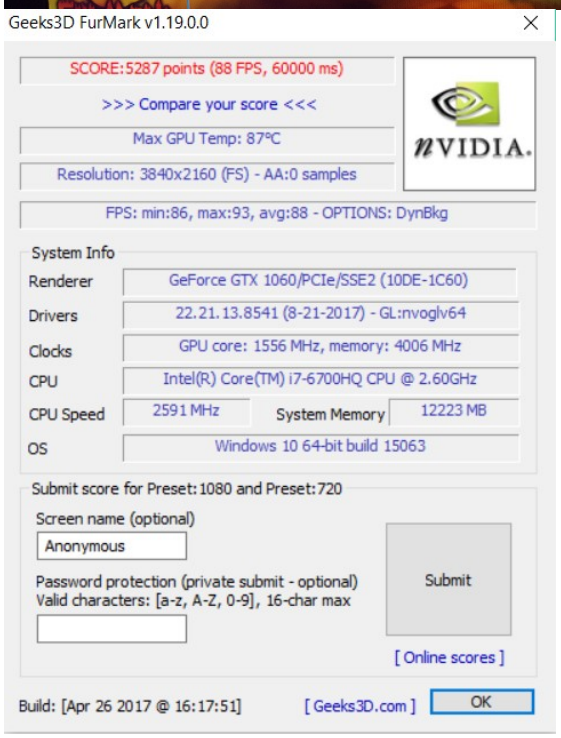
This software tests the graphic cards ability to produce graphics and handle the custom made lighting and custom graphic effects such as tessellation , anti-aliasing, anisotpic filtering etc.
It records the FPS and uses the FPS scored to compare.



2.Furmark

This produces highly complicated geometrical images with different lighting that puts intense load on the GPU and so then records the FPS (frames per second) and uses that for the comparing the performance or the benchmark.
Higher FPS will lead to better performance.
The result is as follows.





By this we can see that different benchmarking systems apply different load on the GPU and different results are produced. Hence the GPU handles different type of graphics differently.

CONCLUSIONS:

We performed tests on Windows 10, mac OS and Linux OS.
The scores obtained by these operating systems taking single core and multi cores as two parameters are as follows:

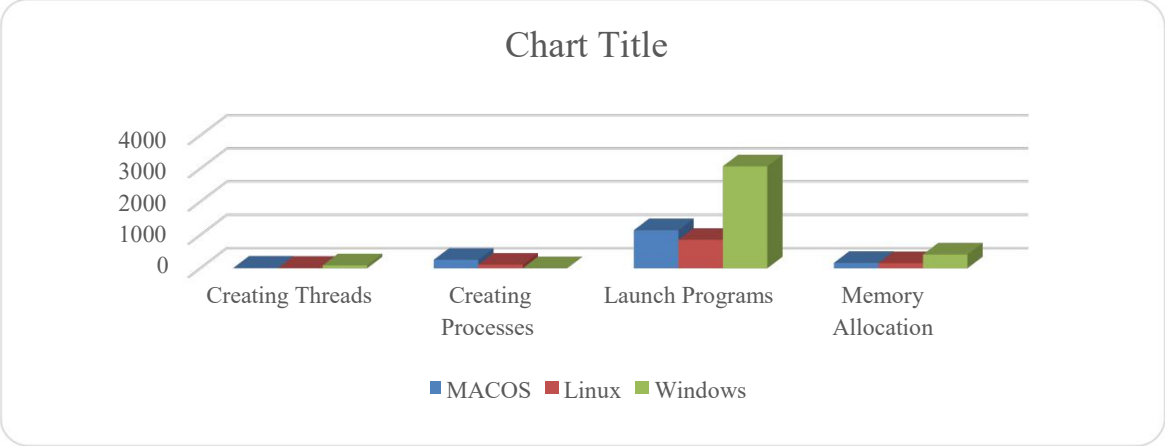
1) **SINGLE CORE:**

OPERATING SYSTEM	SCORE
WINDOWS	3338
macOS	3755
LINUX	3386

2) **MULTI CORE:**

OPERATING SYSTEM	SCORE
WINDOWS	6129
macOS	7276
LINUX	6639

3) **RAW KERNEL AND FILE SYSTEM PERFORMANCE:**



Clearly, Macintosh Operating system stands out in performance for both single and multi-core tasks.
Some of the differences between the operating systems are staggering. It is suspected that the poor process and file creation performance on Windows than macOS and Linux is to blame for the painfully slow git and CMake performance.
Obviously each operating system has its merits, but in general it seems that Linux=macOS > Windows when it comes to raw kernel and file system performance.

REFERENCES

1)
https://www.google.co.in/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&cad=rja&uact=8&ved=0ahUKEwjPjOaxyKXWAhXJipQKHU4rBcMQFggvMAA&url=http%3A%2F%2Fciteseerx.ist.psu.edu%2Fviewdoc%2Fdownload%3Fdoi%3D10.1.1.467.999%26rep%3Drep1%26type%3Dpdf&usg=AFQjCNESBKjnFOsK9m8ahUoa6wnHukzFFg

2)
https://www.google.co.in/url?sa=t&rct=j&q=&esrc=s&source=web&cd=6&cad=rja&uact=8&ved=0ahUKEwjPjOaxyKXWAhXJipQKHU4rBcMQFghXMAU&url=http%3A%2F%2Fwww.cs.ru.ac.za%2Fresearch%2Fg09k3351%2Flittreview.pdf&usg=AFQjCNHN84kTxPihba2SozC3dlL6bzCGIw

3)
https://www.google.co.in/url?sa=t&rct=j&q=&esrc=s&source=web&cd=7&cad=rja&uact=8&ved=0ahUKEwjPjOaxyKXWAhXJipQKHU4rBcMQFghfMAY&url=https%3A%2F%2Fwww.ncbi.nlm.nih.gov%2Fpmc%2Farticles%2FPMC3359088%2F&usg=AFQjCNGDHdOiYQIlA6HcIUl7YZLjZL6KJg

4)
https://www.google.co.in/url?sa=t&rct=j&q=&esrc=s&source=web&cd=9&cad=rja&uact=8&ved=0ahUKEwjPjOaxyKXWAhXJipQKHU4rBcMQFghsMAg&url=http%3A%2F%2Fwww.jimsjournal.org%2F16%2520Mohamed%2520S.%2520M.pdf&usg=AFQjCNELOjezYz8GJtnmVgU1fRH0n08ORg

5)
https://www.youtube.com/watch?v=-3OimssfaII

6)
https://www.google.co.in/url?sa=t&rct=j&q=&esrc=s&source=web&cd=3&cad=rja&uact=8&ved=0ahUKEwiy7sq9yaXWAhULzLwKHas2B0AQFgg5MAI&url=http%3A%2F%2Fwww.techrepublic.com%2Fresource-library%2Fwhitepapers%2Fa-survey-of-benchmarking-techniques-for-real-time-operating-system-performance-analysis%2F&usg=AFQjCNGOgnv3m1d29MyL3osp8IMjHg8s4g

7)
https://www.google.co.in/url?sa=t&rct=j&q=&esrc=s&source=web&cd=10&cad=rja&uact=8&ved=0ahUKEwiy7sq9yaXWAhULzLwKHas2B0AQFghuMAk&url=https%3A%2F%2Fwww.guru99.co m%2Fbenchmark-testing.html&usg=AFQjCNGFGpOfruY1QuzsZlhUzbCXE3ktVg

8)
https://www.google.co.in/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&cad=rja&uact=8&ved=0ahUKEwiy7sq9yaXWAhULzLwKHas2B0AQFggnMAA&url=https%3A%2F%2Fen.wikipedia.org%2Fwiki%2FBenchmark_(computing)&usg=AFQjCNHQ7R1A-HJE5sI0yu01n-jUVaLbqQ