Power Tuning for High Performance Computing on GPGPU Clusters with CUDA/MPI

DaQi Ren[#†1], Reiji Suda[#†2]
[#]Department of Computer Science, University of Tokyo [†]JST, CREST, JAPAN
[1]dren@is.s.u-tokyo.ac.jp, [2]reiji@is.s.u-tokyo.ac.jp

The General Purpose Graphics Processing Unit (GPGPU) is now considered as a serious challenger of High Performance Computing (HPC) solutions because of its suitability for massively parallel processing and vector computation. However the energy usage of GPU has been continually increasing, high performance GPUs may become the largest power consumer in a HPC system. SIMD/SPMD (Single Instruction / Program Multiple Data) are parallel programming models that split up the tasks and run simultaneously on multiple processors with different input. CUDA Processing Element (PE) is a hardware unit composed by CPU and GPU that executes the streams of CUDA kernel instruction, several such PE can be bus-connected and one PE acts as a building block in the multiprocessing system. Multi-core and GPUs provide cooperative architectures in which both SIMD and SPMD programming models can co-exist and complement each other. MPI works as the data distributing mechanism between the GPU nodes and CUDA as the main computing engine. CUDA/MPI platform is becoming a important choice in data intensive HPC in varies of applications, however much less research has been carried out to tune the power performance of CPU-GPU PEs with integrated parallel programming paradigms.

The power of a CMOS chip can be estimated by dynamic power, short circuit power and leakage power [1]. Dynamic power is dominant that can be evaluated by the capacitance switches/clock, operating voltage, clock frequency and chip temperature [1]. If a processor's frequency and temperature are invariable, the power is only dependent on the number of executions. This means the varying amounts of computation energy depending on particular SIMD/SPMD task they perform at a given time. Therefore, the design of algorithms has impacts on the amount of power consumption and computer resources required for a given computing problem, and the existing performance tuning approaches such as code optimizations, cache strategy and workload parallelization can be also used for power tuning. For example in design level one can partition and map a problem to fit CUDA memory structure in order to well utilize fast shared memories, hiding data fetching latency and etc. we have shown in [2] that an enhanced CUDA kernel can save 91% of the energy by a simple kernel in matrix multiplications. Also by using parallel GPU [3], multi-threads in a CPU core collaborate with two GPUs can decreased 22% of the energy expense of one single GPU in computing the same problem. Other than performance optimization, traditional power tuning approaches manage power of computing systems by allowing manually/automatically adjust the frequency and voltage supplied to CPU, i.e. Dynamic Frequency Scaling (DFS) and Dynamic Voltage Scaling (DVS), which reduces both the amount of heat produced and electricity consumed.

The power dissipation of a workload on systems composed of CPU-GPU PEs is the sum of the power consumption of the various components. The power can be modeled as $P_{system}(w) = \sum_{i=1}^{N} P_{GPU}^{i}(w^i) + \sum_{j}^{M} P_{CPU}(w^j) + P_{memory}(w)$ where P_{system}, P_{GPU}, P_{CPU} and P_{memory} represent the power of the system, GPU, CPU and memory including data operation and transmission via PCI buses, respectively. N and M is the number of GPU and CPU that involved in the computing of workload w. w^i and w^j represent the workload assigned to GPU_i and CPU_j, respectively. P_{GPU} and P_{CPU} are related to each other when CUDA kernels performing on them. Above specifications show the fundamental difference between CPU-GPU architecture and the typical CPU environment. This difference determines the solutions in choosing a proper technique of performance tuning and power tuning cannot be the same for above two platforms.

The fact that there are different computing elements involved inside one PE (i.e. CPUs and GPUs, rather than only CPUs) does not change the nature of power optimization problems, however the typical performance tuning techniques for CPU machine such as data-level parallelism, cache optimization and optimal instruction scheduling have to be updated specifically for the requirements of GPGPU architectures. Design of performance tuning mechanism for GPGPU is critically platform-dependent, in detail it needs to solve the following problems:

1. The typical approaches of code optimization and cache strategy for CPU machine have to be redesigned based on GPGPU architecture. In CUDA memory model the GPU global memory is not cached; the frequently reused global data must be manually programmed to be loaded into the fast shared memory; a fast shared memory is shared only by threads in the same thread block. [4] This is a fundamental different with multi-core CPU [5] where the shared memory and the cache/memory hierarchies are dynamically partitioned among the active threads.
2. Design level optimization of GPGPU algorithms on computing parallelization and data parallelization requires additional synchronization mechanisms because CUDA model is lack of efficient global synchronization. Synchronization across thread blocks can be accomplished only by a CPU host after completing CUDA kernel calls. The cost of synchronization will become more expensive when using parallel GPUs. Global synchronization techniques for parallel CPUs cannot be directly applied on GPGPU because of above limiting factors.
3. A CUDA/MPI PE's computation capacity is dependent on each individual CPU and GPU component inside the PE, and their interconnection architectures. Load scheduling and balancing methods for GPGPU are different with that for parallel CPUs. MPI communication buffer and bandwidth optimization needs to refer the target GPGPU memory characters because a host CPU and a GPU device have separate address space.
4. The key design issues of DFS are performance prediction and frequency tuning by which a system can assign SPMD workloads to different processer to execute on a properly designed frequency. A CUDA kernel's frequency is determined by the GPU which is controlled through a CPU host. The power efficiency of a CUDA/MPI PE is dependent on the collaboration of its CPU and GPU components. Only increase/decrease a CPU frequency may not be able to change a CUDA kernel's speed. A list of CPU/GPU frequency combination along with its corresponding predicted performance is essential for DFS mechanism in choosing a preferable power and performance "step" option. Computation character is also important because the power in computing different workload may be different even on the same structure running in the same frequency.

Solution to problem 1 and 2 is suggested by using compiler design optimization to provide language extensions and command line options to CUDA kernel code and let CPU auto-tuning system to guide CUDA optimizations. We approach a solution to problem 3 and 4 in program design level by manually measure the computation powers in order to create a performance profile for each CUDA/MPI PE in the system. The profile will include the number of component inside each PE; CPU/GPU frequency alternatives; computation characters along with its corresponding performance evaluation and the available "steps" for power tuning. Therefore global power tuning mechanisms will use the profiles as guidelines to optimize a computation in achieving the best power performance.

References

1. J. M. Rabaey. Digital Integrated Circuits. Prentice Hall. (1996)
2. Da Qi Ren, Reiji Suda, " Power Model of Large-Scale Matrix Multiplication on Multi-core CPUs and GPUs Platform", Proceeding of PPAM 2009, Wroclaw. (2009)
3. Da Qi Ren and Reiji Suda, "Power Efficient Large Matrices Multiplication by Load Scheduling on Multi-core and GPU platform with CUDA", Proceedings of the CSE 09, pp. 424-429, Vancouver, Canada. (2009)
4. NVIDIA, "CUDA programming guide 2.3". (2009)
5. Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture. Intel Corporation. (2008)