Power Impact of Polyhedral Optimzations on Multicore Architecture

I. Introduction

Previously we have observed that tuning for execution can be used as a proxy to tuning for EDP (Energy-Delay Product). In this report, we look deeper into the power impact of individual loop optimizations and the interactive effect on power and execution time when multiple loop optimizations are applied. This report shed light on how, when and why do optimizations interact.

II. Benchmarks and Experimental Setup

In this work, we evaluate `covariance` Polybench program for understanding the power impact of different loop optimizations. The tests ran on a 2-socket 8-core Intel Xeon E5-2680 processor with 20MB (40MB total) L3 cache. PoCC v1.2 was used to generate program variants from Polybench v3.2. The extra large data set (specified in Polybench) was used. GCC v4.4.6 was the backend compiler. Every executable was compiled with -O3 optimization flag.

III. Experimental Results

Fig. 1 compares the execution time and the power consumption of `covariance` program variants with and without a good tiling size configuration ($32\times16\times1$). We can see from comparing the pink/purple line with the green line that proper tile size reduces execution time by up to $4\times$ (always at least $2\times$ performance increase) and results in minimum execution time.

We can also see from the smooth execution time line that with tiling there are 3 performance breaks and without tilting 1 performance break. The fastest executions of variants without tiling all have maxfuse set – approx 45% improvement of performance. In the case of program variants with tiling,

1) 1st break. Turning vectorization OFF had 45% improvement
2) 2nd break. Setting maxfuse on had 10% improvement
3) 3rd break. Set maxfuse on **and** turn vectorization ON had 100% improvement

It is interesting to note how one optimization (i.e. maxfuse) can impact the functionality of a second optimization (i.e. vectorization) so drastically. The increase of loop size and scope that maxfuse produces allows vectorization to go from a severe lost to a major win (thus turning vectorization on with maxfuse leads to minimum execution time).

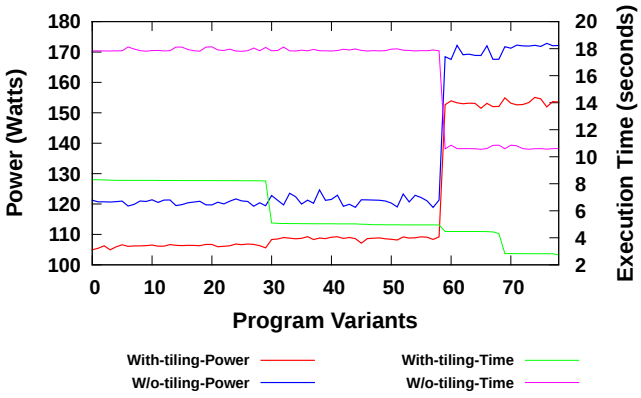

Fig. 1. Graph showing the execution time and the power of `covariance` Polybench program variants with and without loop tiling on SandyBridge Processor (sorted by execution time of program variants with loop tiling turned on). The jumps of power are caused by "maxfuse" loop transformation. The three breaks of the execution time line (with loop tiling) are caused by turning vectorization off, turning maxfuse on, and turning both maxfuse and vectorization on, respectively.

Optimizations can greatly change the power required by an application. E.g. maxfuse takes the tiled version from 108W to 150W (40% increase), the untiled from 120W to 170W (again about 40% increase). Also note that turning vectorization off increased power by a couple of Watts. For this application the power increases are more than made up by the execution time reductions – least energy is the quickest execution time.

Hypothesis: maxfuse allows more memory references to be simultaneously visible to the compiler and vectorization increase greatly – as does overlap of memory and ALU increasing power used considerably.

IV. Todo

Is the above assumption true? How does a machine learning function make sure that it does not give up on vectorization until everything is visible? Are there relationships between optimizations that can be applied across applications? Can this be generalized, i.e. more apps and more optimizations?