基于 GPU 的并行 APSP 问题的研究

张凌洁，赵 英
（北京化工大学 北京 100029）

摘要： Floyd-Warshall 算法是图论中 APSP(All-Pair Shortest Paths)问题的经典算法，为了加快计算速度，提出使用 GPU 通用计算来实现。文章先从算法的原理入手，层层深入，提出了可以在 GPU 上运行的并行 F-W 算法。之后，又根据矩阵分块的原理和 GPU 共享存储器的使用，实现了改进的 GPU 并行 F-W 算法。通过大量测试实验，得到了该 GPU 并行程序相对于传统 CPU 并行程序产生超过百倍的加速比的结论。

关键词： Floyd-Warshall 算法；APSP；GPU；高性能计算；共享存储器
中图分类号： TP302 **文献标识码：** A **文章编号：** 1674-6236(2012)17-0015-04

Research of parallel APSP on GPU

ZHANG Ling-jie, ZHAO Ying
(*Beijing University of Chemical Technology*, *Beijing* 100029, *China*)

Abstract: How to use GPU-based Floyd-Warshall algorithm to deal with the APSP (All-Pair Shortest Paths) problem in graph theory is introduced. First, on the basis of the principle of F-W algorithm, its parallelized version is put forwards on GPU. Then, according to the matrix segmentation and the exploit of the shared memory in GPU, an improved parallel version of F-W algorithm on GPU is introduced. At last, we make a comprehensive comparison and analysis of these algorithms, the speedup is over 100x.

Key words: Floyd-Warshall algorithm; APSP; GPU; high performance computing; shared memory

最短路径问题是图论中的一个典型问题，已经得到广泛的研究和应用。此问题包含两类：1) 单源最短路径(SSSP, Single Source Shortest Paths)，求给定始末顶点之间的最短路径，如旅行商问题；2)所有顶点间最短路径(APSP, All Pair Shortest Paths)，求图中所有顶点对之间的全部最短路径。

目前针对此问题的相关研究成果已被广泛应用于众多领域，如城市物流规划、图形学图像分割等。由此可见最短路径问题不仅具有较高的科研意义，还具有相当高的实际应用价值。

另一方面，当今社会已步入信息化时代。海量数据以及复杂的科学计算都对计算机处理能力提出了更高的要求。HPC 已经成为热点研究问题。其中 GPU 作为低成本低功耗的新型高性能技术得到广泛关注和长足的发展。尤其是在 CUDA 平台出现后，打破了图形处理编程与普通编程之间的壁垒，对于 GPU 的发展可谓是如虎添翼，使得更多的程序员跨入到 GPU 通用计算的研究当中[1-2]。

在 GPU 成为研究热点的背景下，利用其高性能和低成本的优势，通过对 APSP 最短路径并行算法的研究，解决 Floyd 算法在 GPU 并行化中产生的问题。并根据 GPU 特点，提出相应改进策略，从而进一步加快计算速度。希望将来可以把研究成果运用到如物流配送中心的选址，城市规划等实践中。

1 Floyd-Warshall 算法简介

APSP 问题就是求解给定图中所有顶点对之间的全部最短路径。此问题有两种解法：

第 1 种解法是，每次以一个顶点作为起始顶点，依次将其他顶点作为终止顶点求它们之间的最短路径，然后更换起始顶点，依次再求其他顶点间的最短路径。如果对每对顶点之间都使用 Dijkstra 算法，则总的时间复杂度将是 $O(n^4)$。

第 2 种解法是 Floyd-Warshall 算法，也称传递闭包算法。假定一张带权的有向图 G，包含 n 个顶点$\{V_i, 1\leqslant i\leqslant n\}$，及 m 条有向边，有邻接矩阵 Arcs 和初始距离矩阵 $D^{(-1)}$。串行的 Floyd Warshall 算法的计算步骤如下：

1)初始化：初始化邻接矩阵 D，如果 V_i 和 V_j 之间存在直达路径，则将 $D^{(-1)}[i][j]$ 的值设为路径长度；如果不存在直接路径，则 $D^{(-1)}[i][j]$的值设为 ∞。注意，默认情况下，顶点到顶点自身的路径长度为 0；

2)依次计算 $D^{(k)}[i][j](k=0,1,\cdots n-1)$中 $D[i][j]$的值，如果经过中间顶点 V_k，$D[i][k]+D[k][j]$的值小于 $D[i][j]$，则表明经过顶点 k 存在更短路径，更新 $D[i][j]$值，直至图中的所有顶点都做过中间顶点得到最后的 D 矩阵。

使用 Floyd-Warshall 算法的串行版本去处理 ASAP 问题，在数据规模比较小的时候，由于逻辑简单，还是非常实用

收稿日期：2012-04-13 **稿件编号：**201204104
作者简介：张凌洁(1986—)，女，侗族，湖南怀化人，硕士。研究方向：GPU 高性能并行计算。

的。然而,算法主体是三重循环,使得当数据规模随着图中顶点数量增大的时候, 算法的执行时间也将呈现指数级的增加,在处理现实问题时效果并不理想。

2 基于 GPU 的 Floyd 算法的并行化

Floyd-Warshall 串行算法的核心是一个三重循环的迭代计算,算法的时间复杂度是 $O(n^3)$。如果将算法并行化,将极大的缩短计算时间[5]。对于三重循环体的处理,类似于 GPU 芯片对矩阵乘法的计算过程。所以,可以将循环体内的比较权值的操作改为 CUDA 多线程并行执行。实现多线程并行后可以大大提高计算速度,发挥 GPU 并行计算的优势。

该并行化算法的原理如下:首先选取距离矩阵中主对角线位置的元素作为 Key,在 $n\times n$ 距离矩阵中共有 n 个 Key,其坐标值为(0,0),(1,1)…(n,n)。每次迭代选取一个 Key 作为基准,选出与 Key 共行或共列的元素作为参考值,计算余下(n-1)×(n-1)个元素。每个元素 c 计算时,从参考值中找到和本元素共行的元素 a,和参考值中与本元素共列的元素 b,判断 $a+b$ 与 c 的大小关系。如 $a+b$ 更小,则更新该元素值为 $a+b$;否则,不更新。计算完本轮的(n-1)×(n-1)个元素后,Key 元素沿对角线向下移动,按上述步骤重新计算。图 1 展示了计算元素与参考元素的位置关系。

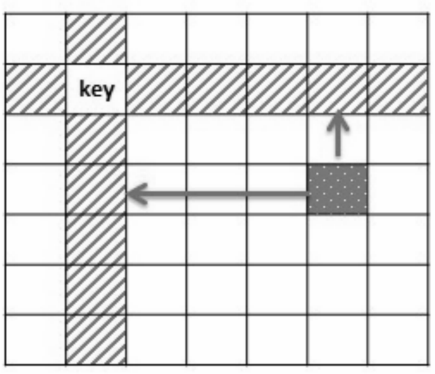


图 1 距离矩阵计算时元素对应关系

Fig. 1 Relationship of elements in distance matrix

GPU 并行程序的重点就是线程的分配和核函数(Kernel Function)的设计[3]。首先介绍如何通过核函数完成多线程的并行计算。并行化的 Floyd-Warshall 算法中使用到了以下几个核函数:

1) 初始化核函数:根据邻接矩阵的值初始化距离矩阵中的权值。这里需要注意线程的对应关系,将邻接矩阵等价为一维数组,依次复制其中的值到 grid 中,并赋值给距离矩阵。所以每个线程在循环处理中的增量大小为整个 grid 中线程的个数。

2)计算核函数:计算距离矩阵中每个元素位置对应的最短路径。距离矩阵中的每个元素都与主轴上对应的同行和同列的两个元素的权值之和做比较。如,D_{ij} 表示顶点 i 到顶点 j 的路径权值,需要和 $D_{ik}+D_{kj}$ 的权值作比较,其中 D_{ik} 与 D_{ij} 同行,D_{kj} 与 D_{ij} 同列,而 k 表示当前迭代时是以顶点 k 作为中间顶点。

下面介绍线程的分配。这时会遇到下面的两种情况:当要处理的矩阵小于或等于 grid 的大小,那么在线程在计算相应的顶点权值的时候, 需要首先对是否在边界当中进行判断。如果在边界当中,则继续进行处理和计算,反之,则放弃计算;当要处理的矩阵规模大于 grid 中线程个数时,需要计算其相对位置,分批次计算,如图 2 所示。

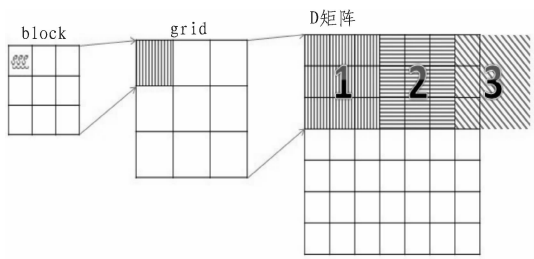


图 2 距离矩阵计算时线程对应关系

Fig. 2 Relationship of threads on grid, block and computing distance matrix

当距离矩阵规模较大时,根据线程数分块和相应的映射原则, 将数据交由 GPU 的 grid 计算。GPU 属于两层并行结构,其中每个 grid 中含有多个 block,每个 block 又含有多个线程,上图中左侧的两个图就表示了这个关系[4]。当 grid 中的线程数不足以为每个矩阵元素分配线程时,需要按照 grid 的大小将原距离矩阵分块,依次分配给 grid 计算。在上面最右侧的图片中,1 号子矩阵分配给 grid 计算完成后,再将 2 号子矩阵分配给 grid 计算,然后是 3 号子矩阵。注意,这里距离矩阵的大小并不能正好整除 grid 的大小(7%3! =0),所以 3 号子矩阵的元素个数比较少,可以分配给 grid 计算,剩余部分不做计算。以此类推,向左向下继续计算各个子矩阵。这里使用相应的行偏移和列偏移变量去表明 GPU 上 grid 应该对应的距离子矩阵的方位。

3 Floyd-Warshall 并行算法的改进

在上一节中, 已经实现了基于 GPU 的并行的 Floyd-Warshall 算法,但是为了进一步提高程序的计算效率,可以利用 GPU 当中的共享内存对程序进行改进。这里借用了 Venkataraman 提出了一个更复杂的、基于“块”的 Floyd 的 APSP 算法[7]。对于大规模的图而言,这个算法可以更好的利用缓存,并提供了更加具体的性能分析。在课题研究过程中,对这个的应用进行了扩展,使其通过 CUDA 平台运行在 GPU 上,并获得更高的效率。原始的 F-W 算法并不能够拆分成相互之间无关联的子矩阵,然后分给 GPU 上的每一个处理器来分开计算。其原因就在于每一个子矩阵需要和整个数据集产生对应关系。查看 F-W 算法串行,可以发现 D 矩阵中的每个值的更新都需要检查矩阵中的其他的每一个数据。所以想要简单地拆分矩阵并不是可行的。但是,通过仔细选择的一些子矩阵序列,就可以解决这个问题。而这个分块的 F-W 算法就是使用之前处理过的某些子矩阵去决定现在正需要处理

的子矩阵，这就使得问题并行化成为可能。

算法首先需要对距离矩阵进行分块，而划分完成后，就可以开始分阶段进行计算了。矩阵主模块个数等于迭代的次数[7]。以下以 6×6 的矩阵为例，对这个算法进行简要地介绍。首先如下图将这个矩阵分出 3 个主模块（主对角线上的 3 个子矩阵），所以就产生了 3 次迭代。每个阶段根据自己的主模块来运算，算法的每一阶段都分为同样的 3 步计算。

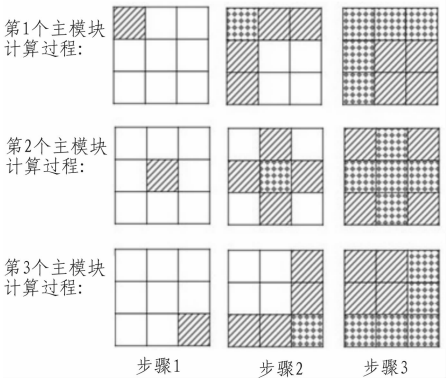


图 3　分块 F-W 算法过程演示

Fig. 3　Example of GPU-Tiled-FW parallel computing

步骤 1：只计算主模块中的最短路径，观察是否仅仅经由主模块所包含的顶点能够获得更短的路径。此时将主模块分配给 CUDA 中的一个 block 去计算，所以此步骤只需要用到了一个多处理器。主模块就是当前需要计算的模块，所以除了主模块本身，不在需要加载任何其他数据到共享存储器（Shared Memory）。每一个线程可以直接加载它们的线程编号对应的数据块，并在这一阶段的计算结束后，将结果保存到全局存储器（Global Memory）中。如图 4 所示，为当第 2 个主模块为当前主模块时进行步骤一时的情形。

步骤 2：计算那些只依靠自己和相应主模块来运算的子模块，即与主模块在同一行和同一列的模块。此时，主模块和当前计算模块都需要加载到共享存储器中，这就使得 GPU 中每一个线程都能从 block 的共享存储器中加载到一个对应的单元。同样，在此阶段完成后，需要每个线程将数据传回到全局存储器中。如图 4(b)所示，为当第 2 个主模块为当前主模块时进行步骤 2 时的情形。

步骤 3：计算剩余的子模块，如图 4 所示，当以 7 号模块为主模块的计算到达步骤 3 时，只有 1、2、3、4 号模块需继续计算。也就是说，当完成步骤 1 和步骤 2 时，还要处理剩下的 $(n-1)\times(n-1)$ 个子模块。此时需放入共享存储器中的模块除了自己之外，还需要将以当前主模块为轴的同行和同列对应的模块一并载入到共享存储器当中。如图 4(c)所示，为当第 2 个主模块为当前主模块时进行步骤 3 时的内存分配情形。

完成上面 3 个步骤，沿对角线向下移动主模块，重复上面的计算步骤。直至全部主模块都计算后，才算完成依次迭代。再进行第二次迭代，比较两次迭代后的距离矩阵：如果经过后一次迭代，距离矩阵并未发生变化，则迭代结束；否则，进行新一轮的迭代，直至距离矩阵不再发生改变。

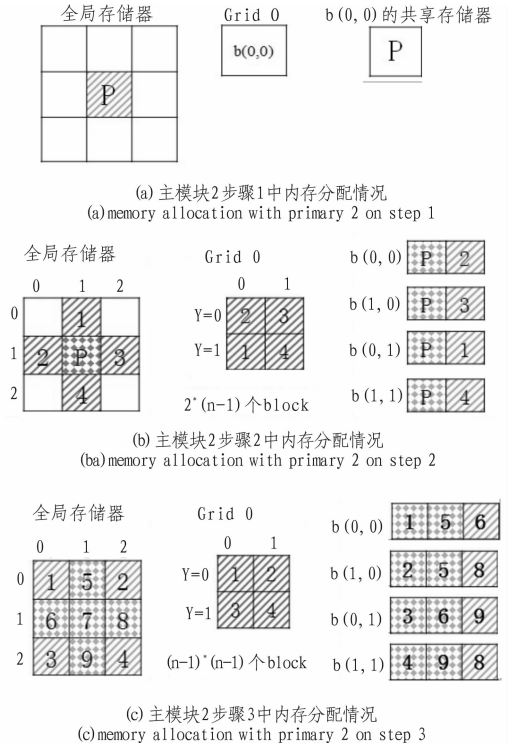


图 4　内存分配情况

Fig. 4　Memory allocation

4　实验结果比较与分析

文中所有的程序运行的硬件平台为惠普公司的 ProLiant 系列 SL390s G7 服务器。其中每个节点服务器都由两个 2.4-GHZ 的英特尔 Xeon 系列 E5620 处理器以及 24 GB 大小的内存和两块 NVIDIA 的 Tesla 系列 M2050 的 GPU 构成。本课题的软件平台为 64 位 Linux RedHat 6.0 操作系统，NVIDIA Tesla 驱动器，CUDA SDK 3.2，CUDA Toolkit 4.0。所以的程序运行数据都是在上述平台上运行 15 次取平均值获得的。

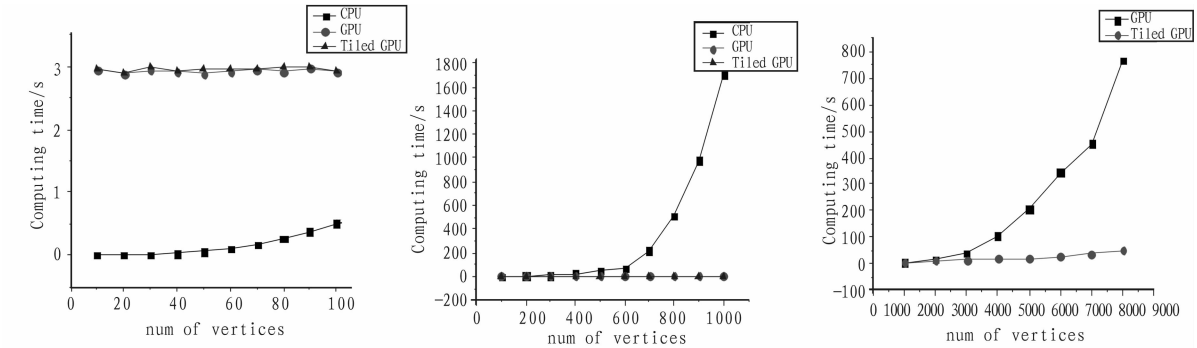
从表 1 的数据和性能比较图中可以看出，无论是原始的 GPU 并行 F-W 算法还是之后改进的分块的并行 F-W 算法，当数据规模较小时（≤100）时，算法执行时间维持在一个较低的水平，大概在 3 s 左右。而且从图中也无法看出改进的 GPU 并行程序优越于原始的并行程序，可以认为它们是相当的。而串行的 CPU 版本的 F-W 算法却极其出色的完成了任务，基本在 0.5 s 内就完成计算。这可能是由于在 GPU 并行算法当中，需要进行 GPU 设备与主机内存之间的数据传输，而这个传输是非常耗费时间的，且由于计算规模太小，使得这一瓶颈突显出来。

对于改进的 GPU 并行 F-W 算法来说，也是由于较小的计算量，使得算法中的分块以及加载使用共享存储器并没有显现出优势，所以使用两种并行算法在效率上没有什么分别。另外，值得注意的是，可以从图 5(a)中看到，串行版本的 F-W 算法在数据增加的后期，已经开始发生明显地时间增加，尽管增加的绝对值并不大。

表1　光顶点数≤100的Floyd算法串行程序和GPU并行程序耗时比较

Tab. 1　Cost time comparison with less than 100 vertices on CPU, GPU, and Tiled-GPU

顶点数	10	20	30	40	50	60	70	80	90	100
CPU 串行计时/s	0.00	0.00	0.01	0.03	0.06	0.10	0.17	0.27	0.38	0.51
GPU 并行计时/s	2.94	2.88	2.93	2.92	2.89	2.92	2.95	2.92	2.96	2.91
GPU 并行分块计时/s	2.94	2.88	3.00	2.93	2.94	2.94	2.95	2.98	2.98	2.91



(a) 100顶点内的3种Floyd算法计算时间 (a) computing time with less than 100 vertices

(b) 1000顶点内的3种Floyd算法计算时间 (b) computing time with less than 1000 vertices

(c) 8000顶点内的2种GPU上并行的Floyd算法计算时间比较 (c) computing time with less than 8000 vertices

图 5　算法计算时间

Fig. 5　Computing time

表2　光顶点数≤1000的Floyd算法串行程序和GPU并行程序耗时比较

Tab. 2　Cost time comparison with less than 1000 vertices on CPU, GPU, and Tiled-GPU

顶点数	100	200	300	400	500	600	700	800	900	1 000
CPU 串行	0.51	3.48	9.92	22.77	47.24	66.51	219.87	510.42	982	1 711
GPU 并行	2.91	3.04	3.03	3.17	3.41	3.54	3.86	4.28	4.61	5.27
GPU 分块并行	2.91	3.02	3.12	3.19	3.33	3.66	4.00	4.19	4.43	5.20

表 2 的数据表示了在顶点数为 100~1 000，间隔为 100 的输入数据下，各种 F-W 算法计算时间的比较。首先从表 2 中，可以看出两种 GPU 并行 F-W 算法在计算时间上只有相对较小幅度的增加，即从 3 s 左右的计算时间增加到了 5 s 左右。而且在顶点数大于等于 700 时，改进后的分块 GPU 并行 F-W 算法才稳定地表现出了一些优势，尽管这个优势并不大。而 CPU 版本的串行程序的运行时间在这个阶段却产生了指数级的增长，这使得在图 5(b)中，GPU 并行算法和 GPU 分块并行算法的计算时间几乎和坐标轴平行。

表 3 的数据可以看出，在这个阶段 CPU 对 F-W 算法的计算时间已经超出需求范围，这使得在图 5(c)中，并没有出现串行版本的执行时间曲线。很显然，由于算法中的一个二维循环和一个三维循环，使得在这个阶段，每一次的数据增加都会带来迭代次数的“暴增”。而原始的 GPU 并行 F-W 算法也因为顶点数的增加，使其自身的迭代次数和循环处理次数增加。而且这一算法是直接读取全局内存中的数据，这种延迟也会累积到一个可观的数值的。相对而言，改进的分块 GPU 并行 F-W 算法的计算时间增加就小得多。尽管这种算法当然也会因为顶点数的增加，使其迭代次数增加，但是由于使用了共享存储器的优化，且访问共享存储器是访问全局内存的若干倍，这使得迭代次数的增加被访问的加速而掩盖，进而达到了令人满意的并行效果。

表3　光顶点数≤8000 的 Floyd 算法串行程序和 GPU 并行程序耗时比较

Tab. 3　Cost time comparison with less than 8000 vertices on CPU, GPU, and Tiled-GPU

顶点数	1 000	2 000	3 000	4 000	5 000	6 000	7 000	8 000
CPU 串行	1 710.9	2 713.9	4 599.2	8 324.6	16 493	34 920	74 032	N/A
GPU 并行	5.27	15.96	40.98	105.11	207.17	344.01	453.14	769.05
GPU 分块并行	5.20	9.24	14.40	18.29	20.39	28.40	37.73	50.47

5　结　论

文中针对 APSP 问题中的 Floyd-Warshall 算法的并行化进行了较为全面的研究和分析。实现了基于 GPU 的并行的 Floyd 算法，在数据规模超过 100 个顶点后，加速比平均达到 100 倍以上。然后又利用 GPU 架构中的共享储存器，根据 Venkataraman 提出的分块思想，将上述 GPU 并行算法进行优化。进一步优化之后，加速比达到 8 倍以上，计算速度明显加快。

（下转第 22 页）

索方法，介绍了语义检索的基本理论和五种语义推理工具，并选择出 Jena 作为航空武器装备 IETM 语义检索的工具，详细研究了 Jena 语义推理的功能结构、模块架构、以及其 Model API、推理系统、数据存储和检索语法 4 个子模块等关键技术。提出了基于 Jena 推理的 IETM 智能语义检索方法。该方法先检索语义相关集，然后根据语义集合，选择合适的节点获取详细信息。该方法能有效克服传统的关键字检索模型存在的语义缺失问题，且能有效地提高检索的查全率和查准率。

参考文献：

[1] Fuller J J. Plan for DoD wide demonstrations of a DoD improved interactive electronic technical manual (IETM) architecture[R]. West Bethesda:CDNSWC,1998.

[2] Su L P,Nolan M,deMare G,et al. Prognostics frame-work software design tool [C]//Aerospace Conference Proceedings, 2000:18-25.

[3] Mathur A,Ghoshal S,Haste D,et al. An integrated support system for rotorcraft health management and maintenance[C]// Aerospace Conference Proceedings,2010:18-25.

[4] Koh J J,Kwon S D,Kim B U,et al. Implementation of an interactive electronic technical manual based on web-multimedia technology[C]//The 4th Korea-Russia International Symposium on Volume 2,2007:21-24.

[5] Sarma S,Brock D,Ashton K. The Networked Physical World [R].White paper MIT,MIT Auto-ID Center,2001.

[6] Scholer F,Williams H E,Yiannis J,et al. Compression of inverted indexes for fast query evaluation[C]//In Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval,2002: 222-229.

(上接第 18 页)

在后续的研究中，希望可以将 GPU 版本的并行 F-W 算法应用在多 GPU 上，得到更高的性能加速比。并将这一并行算法应用到实际中，如物流中心的选址、城市规划等问题当中。

参考文献：

[1] 许桢. 关于CPU+GPU异构计算的研究与分析[J].科技信息，2010(17):22-28.
XU Zhen. Research and analysis of CPU+GPU heterogeneous computing[J]. Technology Information,2010(17):22-28.

[2] 张舒，褚艳丽，赵开勇，等. GPU高性能运算之CUDA[M]. 北京：中国水利出版社，2009.

[3] Harish P,Narayanan P J. Accelerating large graph algorithms on the GPU using CUDA [J]. High Performance Computing-HiPC,2007:197-208.

[4] 程豪，张云泉，张先轶. CPU-GPU并行矩阵乘法的实现与性能分析[J]. 计算机工程，2010(7):24-25.
CHENG Hao,ZHANG Yun-quan,ZHANG Xian-yi. Implementation and performance analysis of CPU-GPU parallel matrix multiplication[J]. Computer Engineering,2010(7):24-25.

[5] Nvidia: Cuda 3.0.Technical report,Nvidia Corp.(2010) Available at[EB/OL]. http://developer.nvidia.com/object/cuda_3_ 0_ downloads.html.

[6] 卢照，师军. 并行最短路径搜索算法的设计与实现[J]. 计算机工程与应用，2010(46):69-71.
LU Zhao,SHI Jun. Design and implement of parallel computing of shortest paths[J]. Computer Engineering and Applications,2010(46):69-71.

[7] Venkataraman G,Sahni S,Mukhopad H S. A blocked all-pairs shortest-paths algorithm[M]. J. Exp. Algorithmics ,2003.

Marmalade 的跨平台移动应用开发 SDK 支持 MIPS™ 架构

为数字家庭、网络、移动和嵌入式应用提供业界标准处理器架构与内核的领导厂商美普思科技公司(MIPS Technologies, Inc)以及全球领先的跨平台移动应用程序 SDK 供应商 Marmalade 宣布，两家公司正合作将 Marmalade SDK 支持带到 MIPS™ 架构。许多全球畅销的 2 D 和 3 D 游戏，包括 Draw Something(画东西)、Call of Duty(决胜时刻)、Cut the Rope(切绳子)、Lara Croft (古墓奇兵)、Need for Speed (极速快感) 以及 BackBreaker (美式足球) 等均是通过 Marmalade 平台开发的。通过在 Marmalade SDK 中支持 MIPS 架构，这些游戏开发厂商以及未来的游戏与应用程序都能轻松地支持 MIPS-Based™ Android 设备。

凭借 Marmalade SDK，开发人员可采用 C/C++、HTML5、CSS3 和 JavaScript 等快速、容易地创建与发布跨平台应用程序，然后以本地应用的形式支持 Android 和其他众多移动和智能电视设备。全球游戏出版商和开发厂商，包括 Activision、EA Mobile、PopCap、Konami、Square Enix、Namco Bandai 等等都已采用 Marmalade 作为他们跨平台开发的基础平台，包括智能手机、平板电脑和智能电视等。

咨询编号：2012171001