Journal of Universal Computer Science, vol. 14, no. 14 (2008), 2416-2427
submitted: 30/9/07, accepted: 30/4/08, appeared: 28/7/08 © *J.UCS*

Tabu Search on GPU

Adam Janiak
(Institute of Computer Engineering Control and Robotics
Wroclaw University of Technology, Poland
adam.janiak@pwr.wroc.pl)

Władysław Janiak
(Institute of Industrial Engineering and Management
Wroclaw University of Technology, Poland
wladyslaw.janiak@interia.pl)

Maciej Lichtenstein
(Institute of Computer Engineering Control and Robotics
Wroclaw University of Technology, Poland
maciej.lichtenstein@pwr.wroc.pl)

Abstract: Nowadays Personal Computers (PCs) are often equipped with powerful, multi-core CPU. However, the processing power of the modern PC does not depend only of the processing power of the CPU and can be increased by proper use of the GPGPU, i.e. General-Purpose Computation Using Graphics Hardware. Modern graphics hardware, initially developed for computer graphics generation, appeared to be flexible enough for general-purpose computations. In this paper we present the implementation of two optimization algorithms based on the tabu search technique, namely for the traveling salsesman problem and the flow shop scheduling problem. Both algorithms are implemented in two versions and utilize, respectively, multi-core CPU, and GPU. The extensive numerical experiments confirm the high computation power of GPU and show that tabu search algorithm run on modern GPU can be even 16 times faster than run on modern CPU.
Key Words: graphics hardware, tabu search, traveling salesman, flow shop
Category: I.3.6, I.3.m

1 Introduction

Increasing demands of products quality and increasing costs of production requires intelligent systems that are used for decision support in production management to deliver good decisions in a fast manner. Those systems, however, are faced with the optimization problems that are difficult to solve (NP-hard). Moreover, those problems are usually of discrete nature, and thus, cannot be solved optimally in reasonable time. On the other hand, they can be solved by various heuristics, but obtaining good solution (close to the optimal one) requires much computational effort and significant amount of time. An example of the method that is applied for optimization, and delivers good solutions is tabu search.

On the other hand the computational power of modern personal computers increases rapidly due to increasing demands of their users. This computational power follows not only from faster Central Processing Unit (CPU), or multicore CPUs, but also from the Graphics Processing Unit (GPU). The evolution of GPUs over several recent years turned them from simple text-mode screen generators to powerful, programable units intended for movie-like real-time graphics generation. Evolution of those chips was even more rapid than CPUs and their computational power is theoretically greater. Moreover, the introduction of programable vector units (called shaders) into GPUs allowed them to be used not only for graphic generation, but also for other, not graphics-related, purposes.

In this paper, we present some results regarding tabu search implementation using GPU. We propose two tabu search algorithms that utilize graphics hardware: for the traveling salesman problem, and for the flow shop scheduling problem. Performed computational experiments show that the usage of GPU in the tabu search algorithm can decrease its running time almost 16 times in comparison to single core CPU.

The remainder of the paper is organized as follows. The next section is devoted to some basic information of GPGPU (General-Purpose Computation Using Graphics Hardware). We present the methodology of GPGPU and its advantages and disadvantages. Next, in Section 3, we describe the idea of tabu search algorithm and its GPGPU implementation for the well known traveling salesman problem and flow shop scheduling problem. Section 4 presents the results of performed computational experiments that compare in terms of the running time the CPU and GPU implementations of tabu search for above mentioned problems. We conclude the paper by pointing out some directions for future research.

2 General-Purpose Computation Using Graphics Hardware

The General-Purpose Computation Using Graphics Hardware is a quite new programming technique, that utilizes graphics hardware for the computations that are not related to real-time graphics generation. Such computations are possible, since the modern GPUs are usually equipped with some number of programmable units called shaders. The programming of early GPUs equipped with shaders was very restrictive (limited number of instructions, small set of instructions, etc.), however, its high potential was noted by the programmers and caused fast evolution to modern unified shader units. Those unified shader units may function either as vertex shaders, as well as pixel shaders (or geometry shaders, which were introduced recently together with shader model 4.0) and its function may change dynamically according to application demands. The processing pipeline of graphics hardware (limited to the scope of this paper) is depicted in Figure 1.

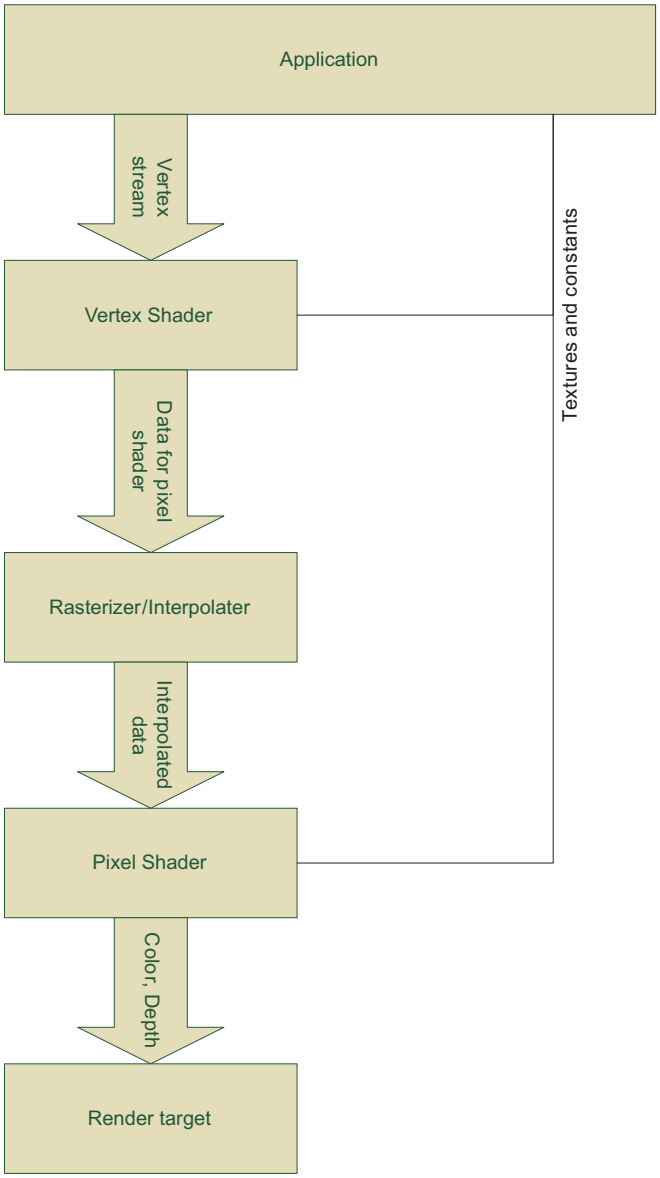

Figure 1: The pipeline of the graphics hardware (limited for the purpose of this paper)

As it can be seen, the vertex stream from application (each vertex is described by the set that can contain its position, normal vector, texture coordinates, etc.) is processed by the vertex shader. The output from the vertex shader contains the data for pixel shader (position, texture coordinates, etc.) and are interpolated across the drawn primitive (triangle) by rasterizer/interpolater. Data from rasterizer/interpolater are processed by pixel shader which outputs the color of the pixel to draw (and optionally the depth in the z-buffer) on the render target. The render target can be either computer screen, as well as the texture in the GPU memory. The application also sends to the GPU textures and some constants that are used by vertex and pixel shader programs, however, those data cannot be changed during the processing of a single vertex stream.

The general idea of GPGPU is to partition the computations into mutually independent (preferably small) programs for shaders (usually pixel shader) called kernels. Since kernels are mutually independent, and there is more than one shader units working in parallel, each of them can process some part of the vertex stream. The goal of the GPGPU is to develop the kernels in such a way, that they will process the data stored in the vertex stream according to our purpose, and store the results in the render target (usually texture), and since we deal with several shader units working in parallel we should be able to perform the computations faster than on a single CPU.

As it can be seen from the description above, the idea of the GPGPU is quite simple, and should speed-up the application significantly (modern GPUs are equipped with 240 shader units), however, the programmers are faced with some limitations. First limitation is the lack of dynamic data structures in kernel programs (even dynamic arrays). This limitation can be overcome by using textures instead of arrays. Those textures, however, have to be prepared separately and may lead to significant data-preparation overhead. Similarly, the only type of data that can be handled by kernels are 32-bit floating point numbers (modern GPUs can handle 64-bit floats). Other restriction is that the number of nested loops in the kernel program may be limited. Finally, the number of instructions per kernel is limited. All those limitations may be overcome in some way. On the other hand, the possible complexity of shader programs allowed to develop high-level programming languages for shader programming, such as NVidia Cg [Fernando and Kilgard 2003] or Microsoft HLSL [St-Laurent 2005].

This brief description of the GPGPU is only an introduction for the purpose of this paper. We refer the reader interested in more details of GPGPU, to the website: `http://www.gpgpu.org`. Moreover, to the best of our knowledge, there are no publications that combine the GPGPU and optimization, so the reader interested in other applications of GPGPU is referred to the survey paper [Owens et. al. 2005].

In the following we present some details of tabu search algorithm for well

known traveling salesman problem and flow shop scheduling problem. Both algorithms, were implemented in two versions:

- version that utilizes only CPU,
- version that utilizes either CPU, as well as GPU.

Then, both version of the algorithms are compared to each other in terms of the running time, showing that the GPGPU can bring its significant shortage.

3 Tabu search

The tabu search (TS) procedure, originally proposed by F.Glover, [Glover 1989, Glover 1990], is a neghbourhood-based search method with deterministic mechanism of avoiding local minima. The general idea of TS is to start from some initial solution, and iteratively move among neighbouring solutions. At each iteration, a move to the best solution in the neighbourgood of the current one is performed. To avoid local minima, the memory of already visited solutions is introduced. Most frequently used type of that memory is the tabu list. The tabu list stores some number of already visited solutions, its attributes, or moves leading to them. During the search process, the move that leads to the solution that is stored in the tabu list is forbidden. Many implementations of tabu search method for various optimization problems (see, for example: [Taillard 1983, Nowicki and Smutnicki 1998, Oguz et. al. 2004]) shows, that tabu search can deliver optimal or near-optimal solutions for NP-hard problems in reasonable time. The efficiency of TS method is strongly dependent on the proper selection of its attributes, i.e:

1. initial solution,
2. neighbourhood,
3. tabu list,
4. stoping condition.

In the following two sections we present some details of our implementation of tabu search for traveling salesman problem, and for flow shop scheduling problem, respectively. Moreover, we present how we ported these methods to take advantage of GPGPU.

3.1 Tabu search for traveling salesman problem

The traveling salesman problem can be defined as follows. There is a given set $N = \{1, ..., n\}$ of n cities. The distance from city i to city j is denoted by d_{ij}

and is given for each $i \in N$, $j \in N$, $j \neq i$. Note that in, general $d_{ij} \neq d_{ji}$. A salesman route can be defined by a permutation $\pi = (\pi(1), \pi(2), ..., \pi(n))$ of the set N, where $\pi(j)$ is the number of the city visited as jth. The problem is to find such permutation π^* that minimizes the total route length $D(\pi^*)$, where $D(\pi)$ is defined as follows:

$$D(\pi) = \sum_{j=1}^{n-1} d_{\pi(j)\pi(j+1)} + d_{\pi(n)\pi(1)}.$$

Our tabu search method for the TS problem was a straightforward implementation. In the following we present some details.

As a initial solution random permutation of cities was chosen.

The neighbourhood was defined by swap moves, i.e., a neighbouring permutations were generated by swapping positions of two cities in the base permutation. More precisely, a swap move is defined by two parameters $i \in N$, and $k \in N$, $i \neq k$, and exchange the positions of elements $\pi(i)$ and $\pi(k)$ in permutation π.

The tabu list stores the parameters of moves (pairs $< \pi(k), \pi(i) >$) and forbids all the solutions π in which element $\pi(k)$ is before the element $\pi(i)$. The length (number of stored move parameters) of the tabu list is constant and equals 7.

Our tabu search implementation stops after 1000 iterations.

It can be easy observed, that over 90% of tabu search running time is occupied by neighbourhood evaluation (i.e., computing the salesman route length for each solution in the neighbourhood). On the other hand this part of the algorithm can be easy parallelized.

In our GPGPU implementation of tabu search method, we prepare two textures that are send to the GPU only once at the beginning of the algorithm. One of these textures (data texture) is square of $n \times n$ pixels, and in each pixel at coordinates (i, j) the distance d_{ij} is stored. The texture has $R32F$ format, i.e., each pixel is a 32-bit floating point number that represents the only the R (red) component of the color. The second texture (neighbourhood texture) is also $n \times n$ pixels square, however, in this texture the neighbourhood is stored. More precisely, each pixel stores a single swap move. Texture has $R16G16$ format (i.e., stores only R and G components of the color, each on 16 bits) and stores the values of parameters i and k of the move, on R and G color components, respectively. Beside these two textures, at each iteration the texture in $R16G16$ format of $n \times 1$ pixels is prepared, and send to the GPU. This texture (permutation texture) represents the current permutation π (only R color component is used).

After sending the permutation texture, the GPU is programmed to draw a quad (a square composed of two triangles) of $n \times n$ pixels, however, as a render target additional texture in $R32F$ format is used. The vertex shader program is a

very simple and only passes the vertex data to pixel shader, whose contains only the neighbourhood texture coordinates (its corners). Those data are interpolated by resterizer/interpolater so entire neighbourhood texture is nicely covered. The pixel shader calculates the route length for each solution in the neighbourhood, by sampling neighbourhood texture, permutation texture, and data texture and stores this value in output render target.

Thus, the briefly explained above program evaluates the neighbourghood of the current solution and stores all the route lengths in the output render target texture. Then, then this texture is fetch from the GPU memory, and the best, non-tabu, neighbour is selected (this part is computed on CPU). After updating tabu list, and permutation texture, all the computations are repeated. This stops after 1000 repetitions.

It is easy to notice, that output texture is symmetrical (i.e, the value of the pixel at (i, j) coordinates is the same as at (j, i) coordinates. This is caused by the fact that swapping the element i with the element k in π leads to the same permutation as swapping the element k with the element i. So, it is enough to draw not the entire quad, but only the upper, or lower triangle. However, the textures that are sent to the GPU have to be rectangular.

3.2 Tabu search for flow shop scheduling problem

The flow shop scheduling problem (FS) can be defined as follows. There is a given set $J = \{1, ..., n\}$ of n jobs to be processed on m processors. Each job $j \in J$ is comprised of m operations. The ith operation of the job j has to be processed on ith processor and requires p_{ij} time units, moreover, it cannot be started before operation $(i-1)$ completes. We deal with the permutation flow shop, in which the sequence of operations at each processors is the same. The schedule for the FS can be precisely defined by the permutation of jobs $\sigma = (\sigma(1), ..., \sigma(n))$, where $\sigma(j)$ is the number of job processed as jth.

Denote by $C_{ij}(\sigma)$ the completion time of the ith operation of job j in the schedule defined by σ. For a given σ we can determine the completion times of each operation from the following formulae calculated for $i = 1, .., m$, $j = 1, ..., n$

$$C_{ij}(\sigma) = p_{i\sigma(j)} + \max\{C_{(i-1)\sigma(j)}(\sigma), C_{i\sigma(j-1)}(\sigma)\},$$

where $C_{0j}(\sigma) = 0$ for $j = 1, ..., n$, $C_{i0}(\sigma) = 0$ for $i = 1, ..., m$, and $\sigma(0) = 0$.

The problem is to find such a schedule σ^* that minimizes the makespan (total schedule length) $C_{max}(\sigma^*)$, where makespan is defined as

$$C_{max}(\sigma) = \max_{j \in J}\{C_{mj}(\sigma)\}.$$

The general idea of the tabu search for flow shop problem is the same as for the traveling salesman problem. There are only two differences. First the length

of tabu list is extended to store 10 last moves. Second is that the neighbourhood is based in insertion-type moves not swap-type moves. The insertion-type move is also defined by two parameters i and k, however, the change in the permutation σ is different. The move removes the element from position i ($\sigma(i)$) and inserts it into position k. Thus, the move with parameters (i, k) in general produces different permutation than move with parameters (k, i). This results in non-symmetrical output texture, so the entire quad (two triangles) has to be drawn.

We do not present the source code of the pixel shaders of our methods because it is quite long and not self explaining. The reader interested in implementation details is asked to send an e-mail to one of the authors and we send back the full source code of our algorithms.

4 Experimental results

In the following two sections we compare the running time of the tabu search that utilizes CPU only, and that utilizes CPU an GPU as well, separately for the traveling salesman problem, and the flow shop scheduling problem. The algorithms that utilize CPU only were implemented in C#. The algorithms that utilizes both CPU and GPU were implemented in C# using Microsoft XNA framework (see `http://www.xna.com`) and the HLSL for shaders programming (see [St-Laurent 2005]). We used two testing platforms. The first platform was based on Intel Celeron 2.9 GHz CPU, and 1.5 GB of DDR2 RAM. We used two following graphics hardware on the first platform:

G8: GeForce 7300 GT (8 pixel shader units, 4 vertex shader units)

G32: GeForce 8600 GT (32 unified shader units).

The second platform was based on four core Intel Xeon X3220 2,4 GHz CPU, 4GB DDR2 RAM and was equipped with GeForce 8800 GT graphics hardware (112 unified shader units), denoted by **G112** in the following .

The CPU versions of the TS algorithm for FS problem run on second platform was implemented is such a way, to take advantage of all the CPU processing power, i.e., at each iteration of TS the neighbourhood was partitioned into 4 parts, and each part was evaluated by separate thread to utilize all four cores of the CPU.

4.1 Results for traveling salesman problem

The tabu search algorithm for traveling salesman problem was tested on the first platform equipped with G32 graphics hardware. In the test instances the distances d_{ij} were uniformly distributed in the range [1,1000]. For each number of cities 10 instances were generated and average running time was considered.

The results of comparison of algorithms (1000 iterations) run on CPU and GPU are summarized in Table 1. From the presented results it can be easy observed

Table 1: The comparison of average running time (in seconds) of tabu search for traveling salesman problem run on CPU and on G32 GPU, where n is the number of cities.

n	5	10	20	30	40	50	60	70	80	90	100
CPU	0.019	0.116	0.652	2.400	4.847	9.803	14.979	21.044	37.104	45.570	78.970
G32	6.759	6.886	6.698	7.422	8.734	10.570	13.491	18.510	21.581	30.597	69.312

that the advantage of GPGPU in this case is not high (12% maximum for 100 cities). This is caused by the large overhead of data preparation which can be seen for small instance size. Moreover, the amount of computations transferred to the GPU is not high, and those computation are not very demanding for the CPU (single loop of n iterations). On the other hand, even in this simple case the computation power of the GPU can be seen, since the relative shortage of the running time increases with the increase of the problem instance size.

4.2 Results for flow shop scheduling problem

The tabu search algorithm for the flow shop scheduling problem was tested on all the platforms indicated in the beginning of the Section 4. Test instances were randomly generated, with operations processing times, p_{ij}, uniformly distributed from the range [1,100]. For each combination of the number of jobs $n \in \{10, 40, 60, 100, 120\}$ and the number of processors $m \in \{5, 10, 15\}$ ten instances were generated and average running time was considered. The results for the first and the second testing platform are summarized in Table 2.

First observation that follows from the presented results is that the G8 GPU is not capable to overcome the CPU. However, G32 and G112 GPUs overcome the CPU 2, and 4 times, respectively. This results hold for large problem instances ($m = 15$ processors), and the ratio (GPU time)/(CPU time) is depicted in Figure 2. For smaller problem instances the overhead in data preparation is still so significant, that the GPU utilization is meaningless. Note that the increase of the ratio depicted in Figure 2 is not linear. This is caused by limited number of shader units and it seams that it has expected stepwise shape.

Note that the implementation for the second testing platform was multithreaded, so the G112 is expected to be up to 16 times faster than single core CPU.

Table 2: The comparison of average running time (in seconds) of tabu search for flow shop scheduling problem run on the first and second testing platform (CPU1, and CPU2) and G8, G32, and G112 GPU, where n is the number of jobs and m is the number of processors.

n	m	CPU1	G8	G32	CPU2	G112
10	5	0.107	7.379	7.842	0.069	3.032
40	5	4.326	10.006	11.051	2.972	4.523
60	5	13.024	21.267	18.239	9.703	7.974
100	5	56.540	102.662	77.284	42.430	33.866
120	5	106.593	163.957	99.139	70.578	45.945
10	10	0.148	7.377	7.695	0.109	3.008
40	10	7.181	12.643	11.165	5.739	4.546
60	10	23.603	30.926	19.583	18.935	8.504
100	10	100.558	154.646	87.120	76.946	36.396
120	10	185.793	248.394	110.650	141.637	47.955
10	15	0.196	7.225	7.658	0.149	3.025
40	15	10.541	15.072	11.633	8.496	4.657
60	15	33.466	38.959	21.459	27.012	9.210
100	15	148.513	201.023	97.471	118.326	38.401
120	15	262.274	324.112	124.568	208.990	53.483

5 Conclusions

In this paper we presented some results regarding application of GPGPU to tabu search optimization method. To the best of our knowledge the presented results are the first ones that combine the GPGPU and discrete optimization. Performed computational experiments indicate that the modern Graphics Processing Unit is more powerful than modern multicore Central Processing Unit. However, there are many difficulties in porting the optimization methods into GPGPU. Those difficulties follows from the fact that the splitting the computations into mutually independent kernels to be processed in parallel by GPU may be difficult or even impossible. Moreover, the limitations of shader programming may cause the impossibility of porting some methods to GPGPU. On the other hand, optimization methods that can be easily parallelized can be implemented according to GPGPU principle and utilize all the power of modern PCs. The demanding market of GPUs (especially market of video games) causes the rapid evolution of the GPUs, so mentioned above limitations in shader programming may not exist in the near future.

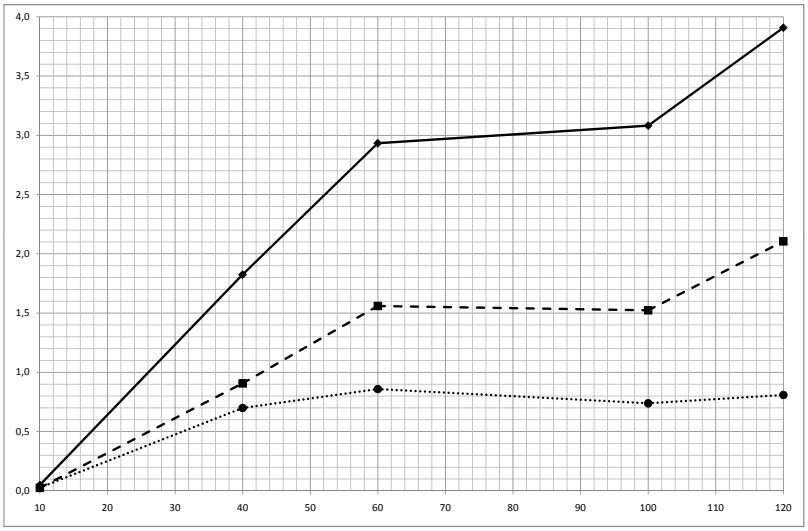

Figure 2: The (GPU time)/(CPU time) ratio for $m = 15$ processors. Values greater than 1 indicate that GPU implementation is faster than CPU implementation. G8 – dotted line, G32 – dashed line, G112 - solid line

References

[Fernando and Kilgard 2003] Fernando, R., Kilgard, M. J.: "The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics"; Addison-Wesley, 2003

[Glover 1989] Glover, F.: "Tabu search. Part I"; ORSA Journal on Computing, 1 (1989), 190-206

[Glover 1990] Glover, F.: "Tabu search. Part II"; ORSA Journal on Computing, 2 (1990), 4-32

[Nowicki and Smutnicki 1998] Nowicki, E., Smutnicki, C.:"The flow shop with parallel machines: A tabu search approach"; European Journal of Opemtional Research, 106 (1998), 226253

[Oguz et. al. 2004] Oguz, C., Zinder, Y., Do, V. H., Janiak, A., and Lichtenstein, M.:"Hybrid flow-shop scheduling problems with multiprocessor task systems" ;European Journal of Operational Research, 152 (2004), 115131

[Owens et. al. 2005] Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krger, J., Lefohn, A. E., and Purcell, T. J: "A Survey of General-Purpose Computation on Graphics Hardware", In Eurographics 2005, State of the Art Reports, August 2005, 21-51.

[St-Laurent 2005] St-Laurent, S.: "The COMPLETE Effect and HLSL Guide "; Paradoxal press, 2005

[Taillard 1983] Taillard, E.: "Some efficient heuristic methods for the flow shop sequencing problem"; European Journal of Operational Research, 47 (1983), 6474